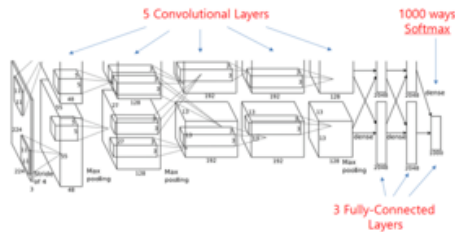


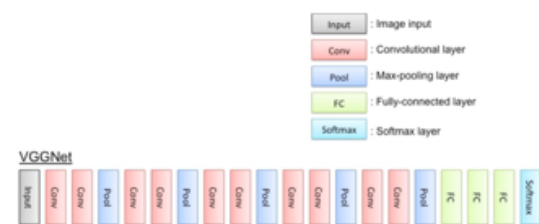
Review of Previous lecture

CNN Architectures

AlexNet



VGG



GoogLeNet

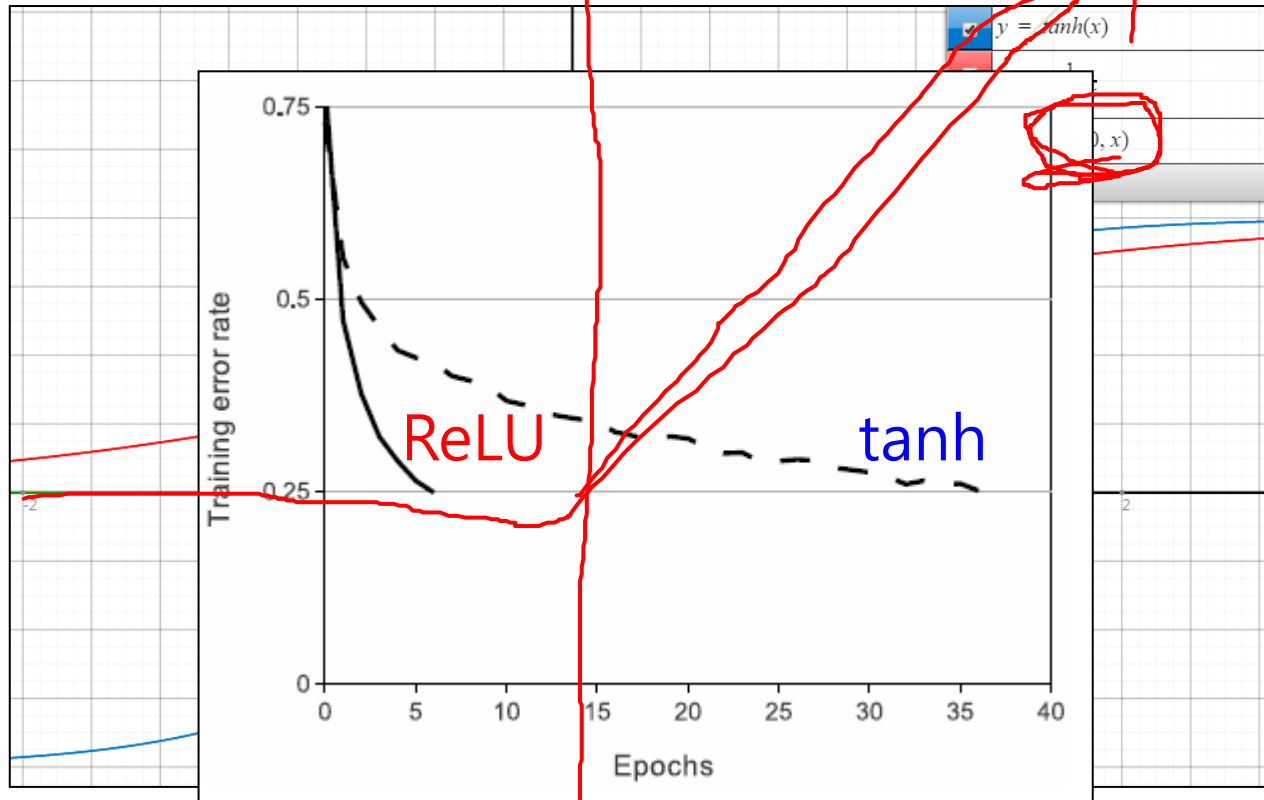


ResNet



ReLU

Rectified Linear Unit

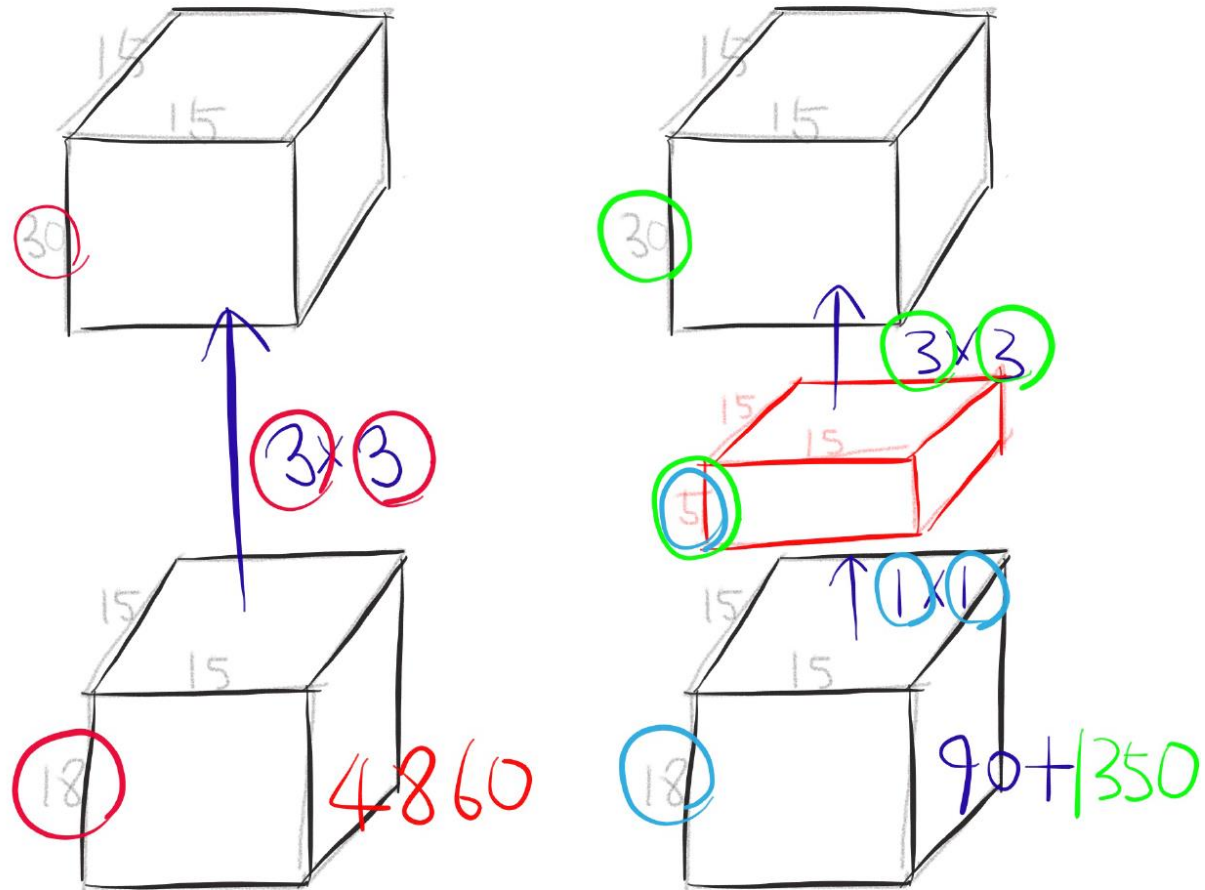


Faster Convergence!

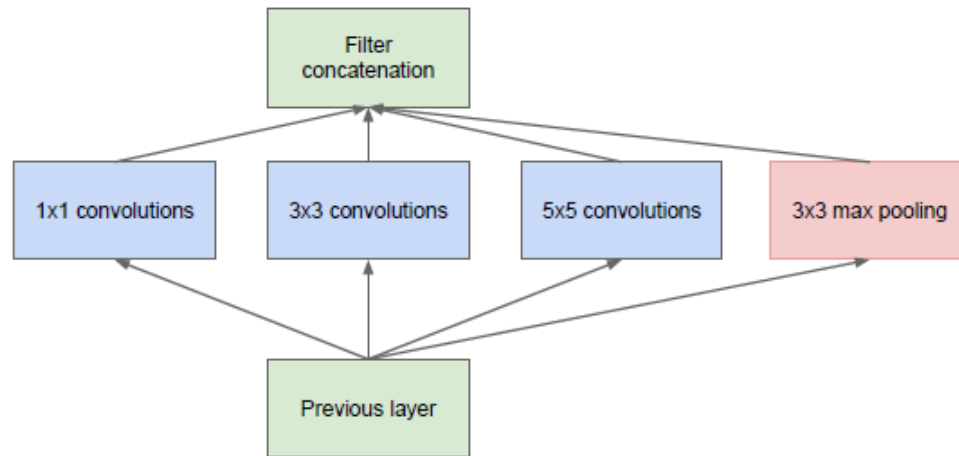
VGG?

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

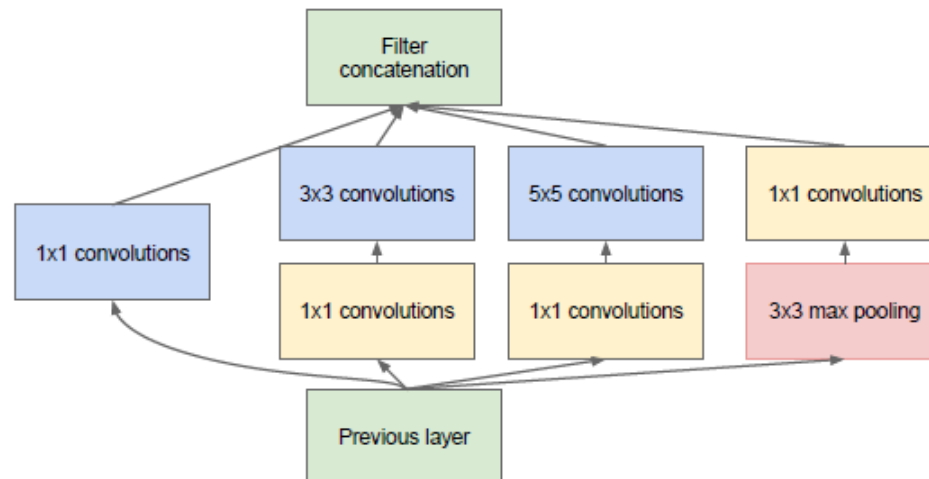
One by one convolution



Inception module

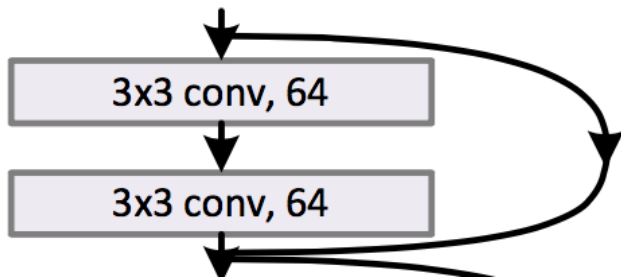
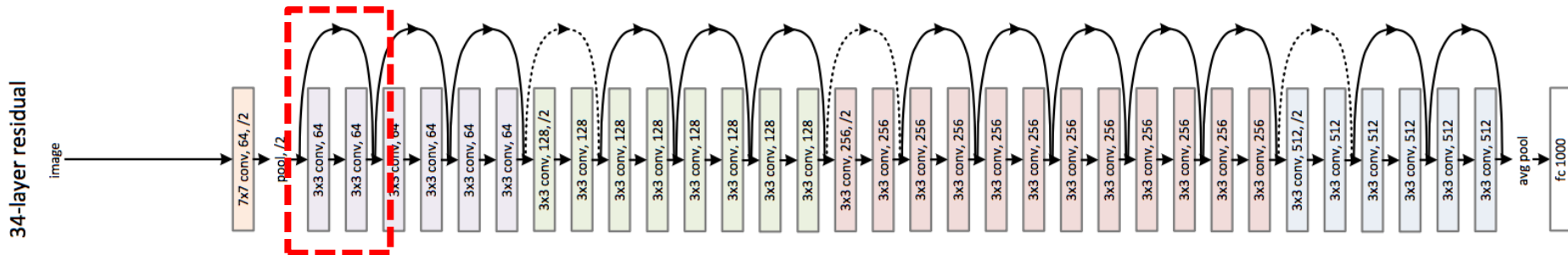


(a) Inception module, naïve version



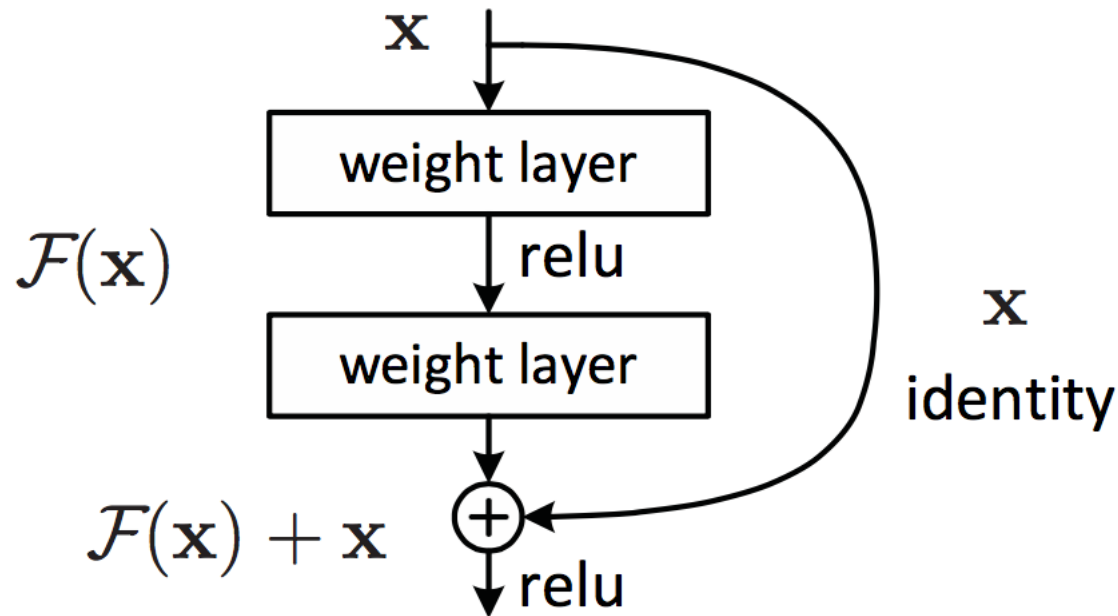
(b) Inception module with dimensionality reduction

Deep residual network



```
def residual_block(x, n_in, n_out, subsample, phase_train, scope='res_block'):
    with tf.variable_scope(scope):
        if subsample:
            y = conv2d(x, n_in, n_out, 3, 2, 'SAME', False, scope='conv_1')
            shortcut = conv2d(x, n_in, n_out, 3, 2, 'SAME',
                             False, scope='shortcut')
        else:
            y = conv2d(x, n_in, n_out, 3, 1, 'SAME', False, scope='conv_1')
            shortcut = tf.identity(x, name='shortcut')
        y = batch_norm(y, n_out, phase_train, scope='bn_1')
        y = tf.nn.relu(y, name='relu_1')
        y = conv2d(y, n_out, n_out, 3, 1, 'SAME', True, scope='conv_2')
        y = batch_norm(y, n_out, phase_train, scope='bn_2')
        y = y + shortcut
        y = tf.nn.relu(y, name='relu_2')
    return y
```

Residual learning building block



Regularization

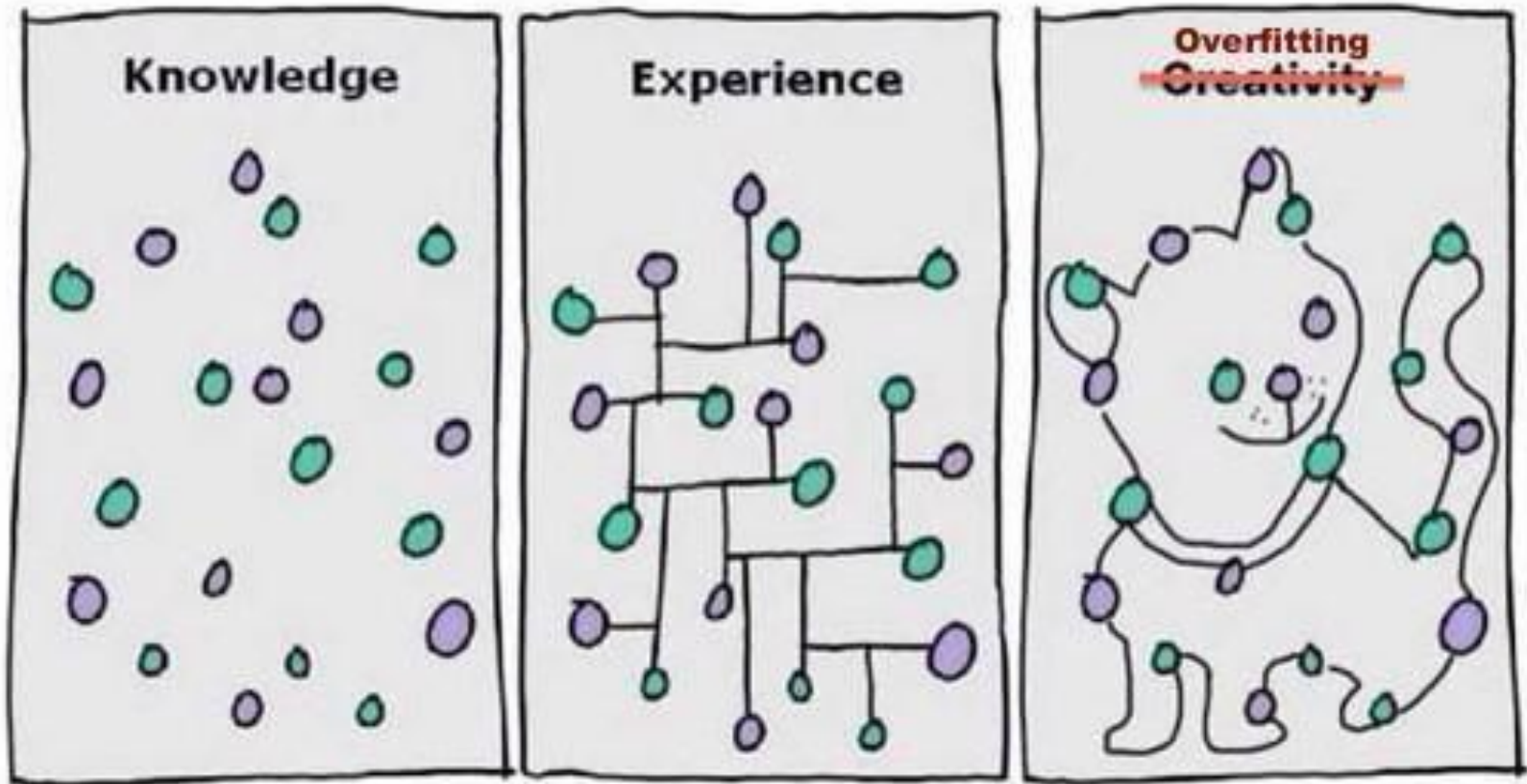
Sungjoon Choi
(sungjoon.choi@cpslab.snu.ac.kr)

Regularization?



Main purpose is to avoid "**OverFitting**"

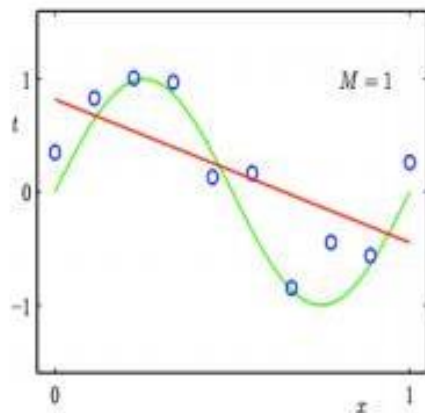
OverFitting?



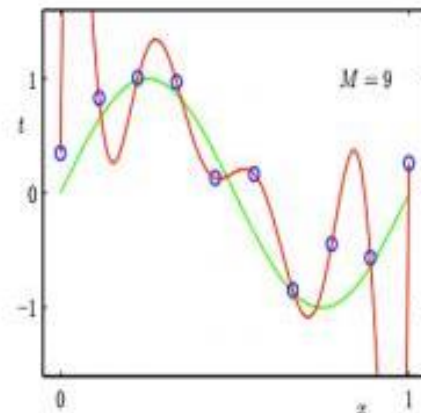
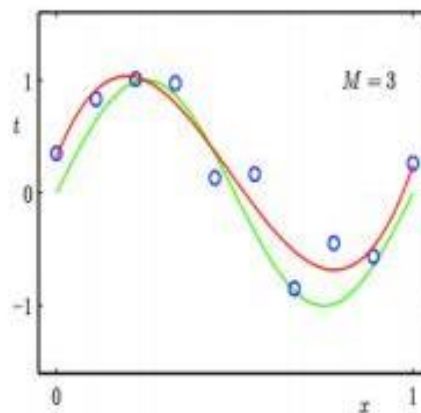
This actually happens **all the time!**

Examples

Regression:

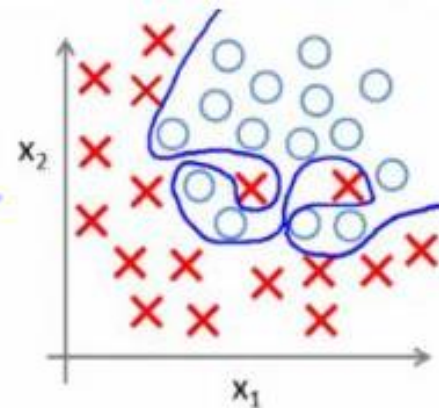
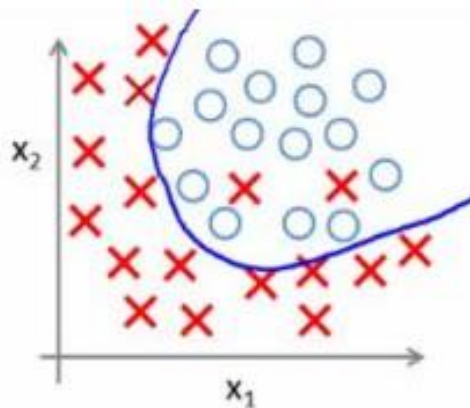
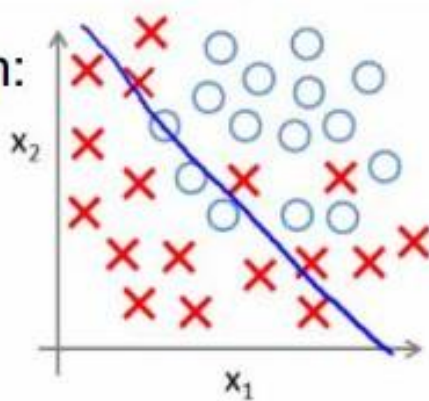


predictor too inflexible:
cannot capture pattern



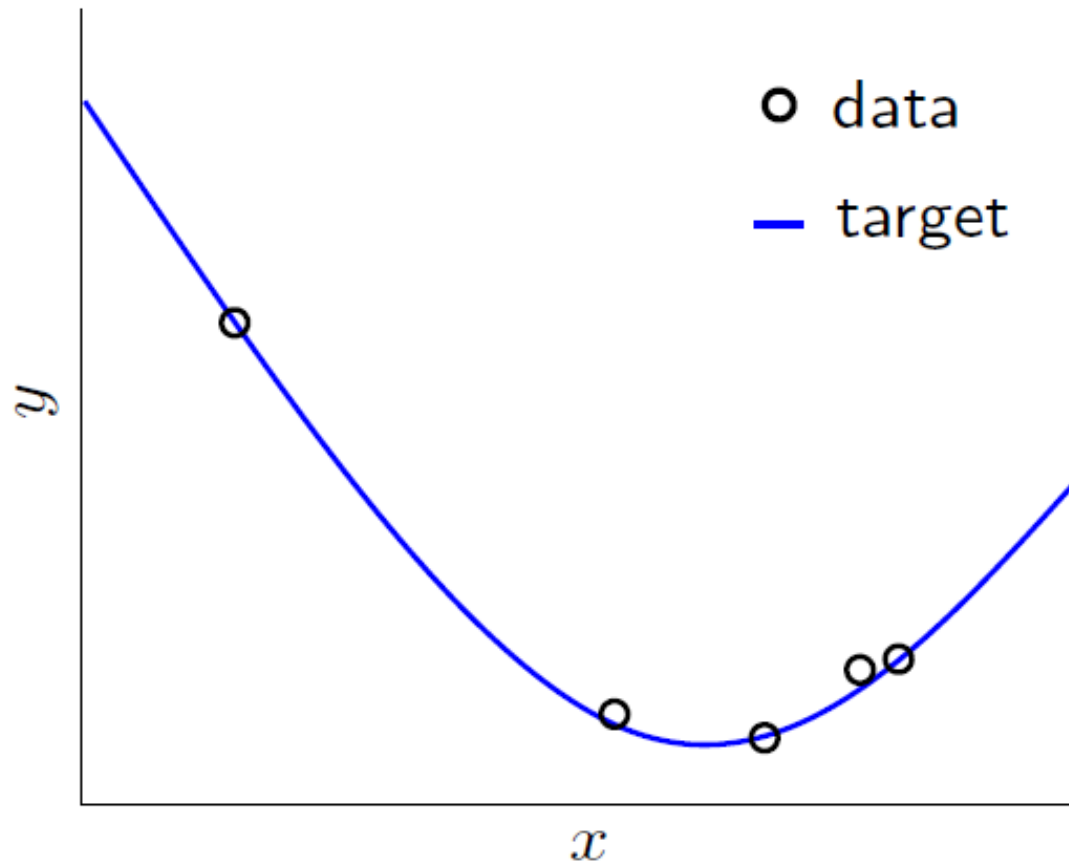
predictor too flexible:
fits noise in the data

Classification:



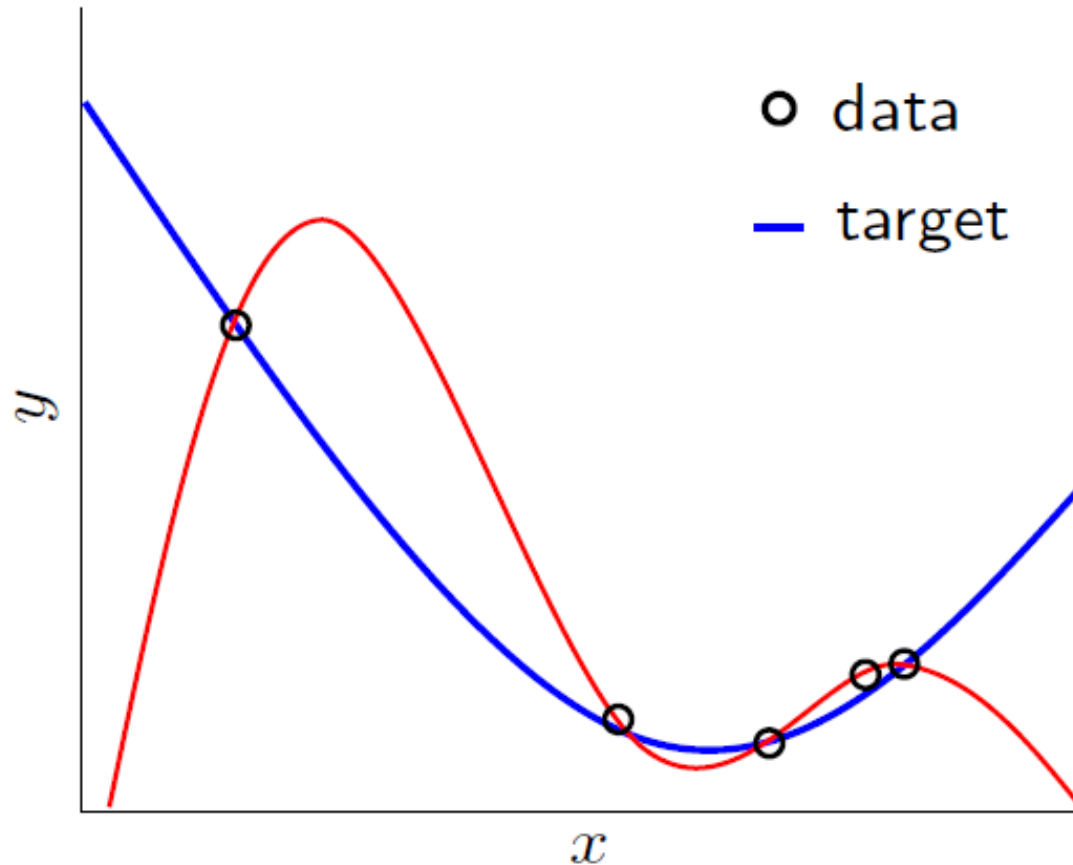
OverFitting

Literally, **Fitting** the data more than is warranted



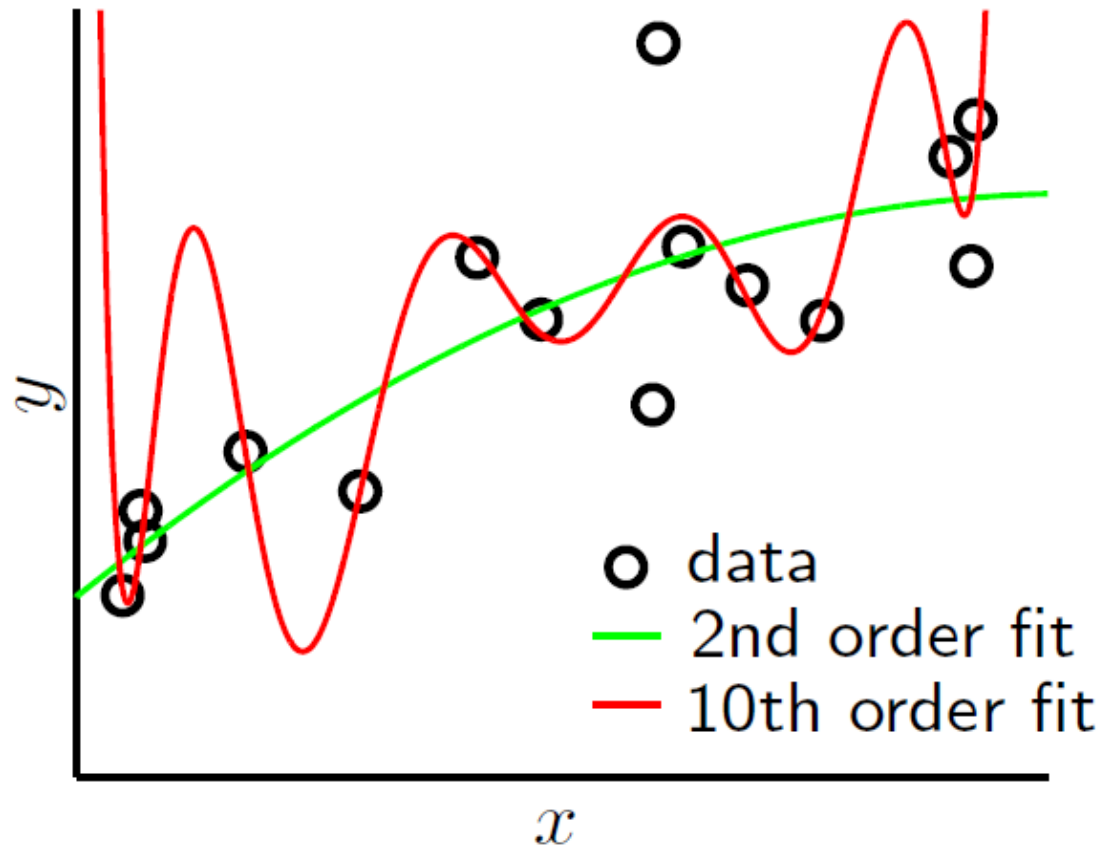
OverFitting

Literally, **Fitting** the data more than is warranted



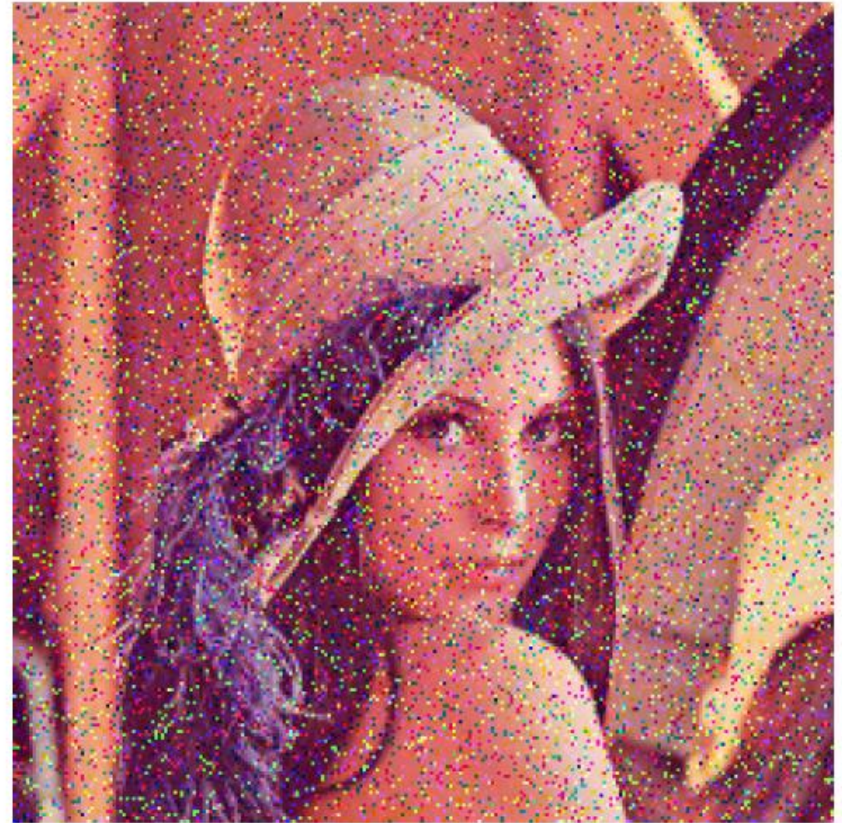
OverFitting

Things get worse with **noise**!



Stochastic Noise

Comes from **random measurement error**

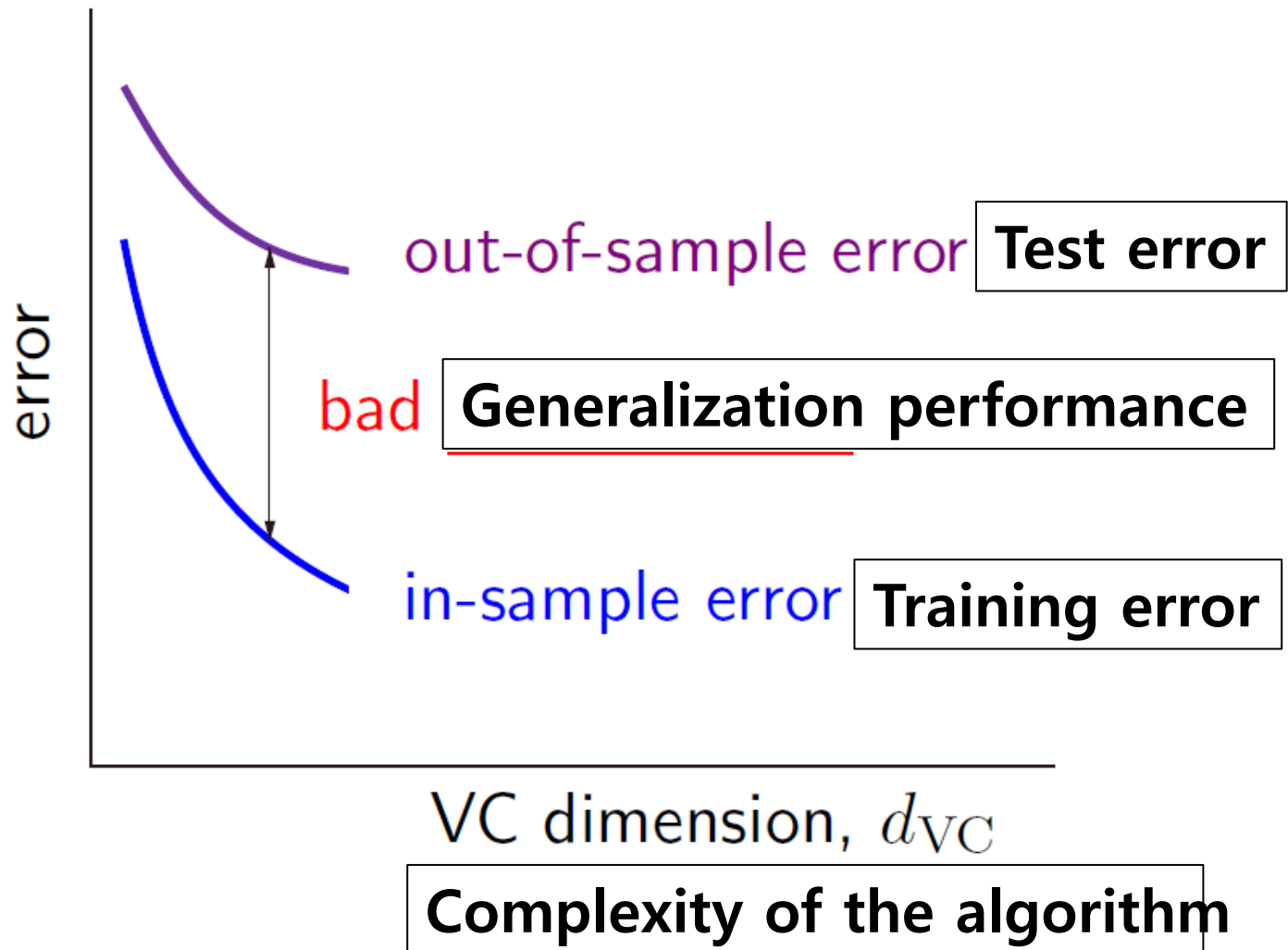


Deterministic Noise

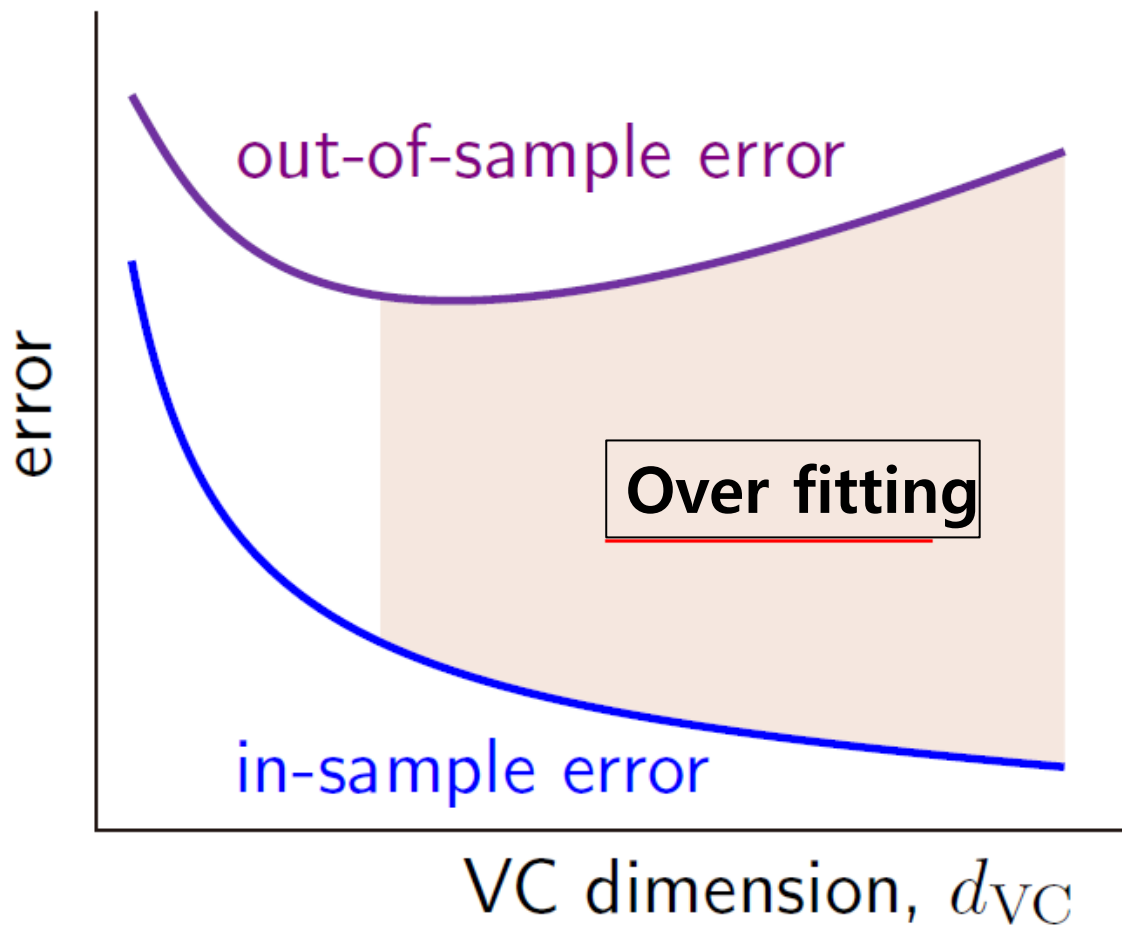
Cannot model this type of error



Mathematically,



Mathematically,



Preventing OverFitting?

Approach 1: Get **more** data



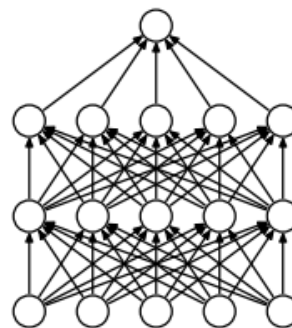
Approach 2: Use a model with the right **capacity**



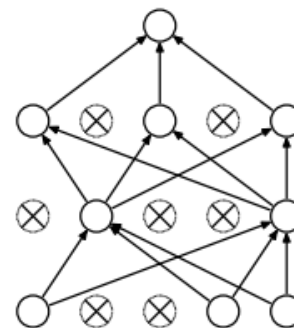
Approach 3: **Average** many different models (Ensemble)



Approach 4: Use DropOut, DropConnect, or BatchNorm



(a) Standard Neural Net



(b) After applying dropout.

Limiting the Capacity



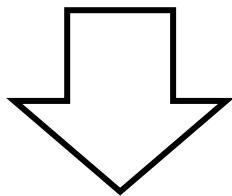
Architecture: Limit the number of hidden layers and units per layer

Early stopping: Stop the learning before it overfits using validation sets

Weight-decay: Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty)

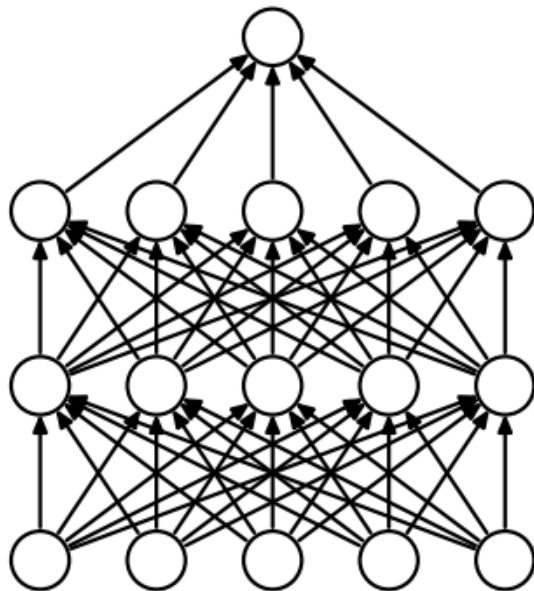
Weight Decay

$$E(w) = E_0(w) + \frac{1}{2} \lambda \sum_i w_i^2$$

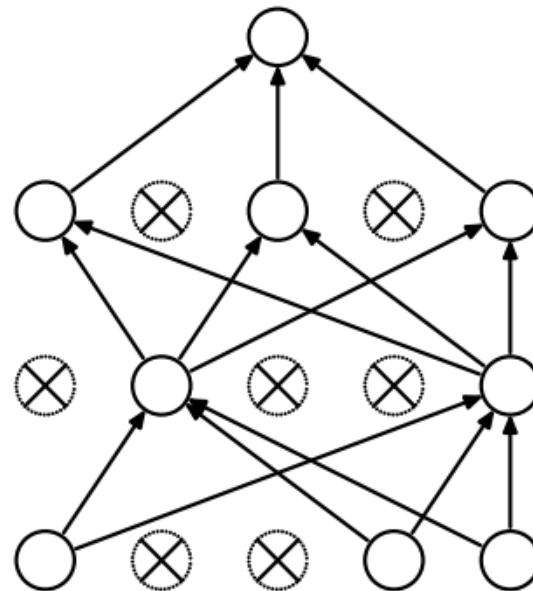


$$w_i \propto -\frac{\partial E_0}{\partial w_i} - \lambda w_i$$

DropOut



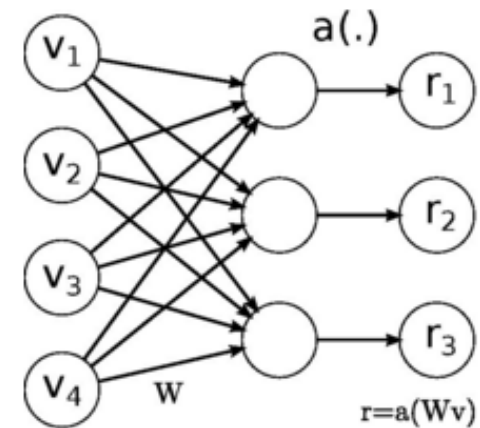
(a) Standard Neural Net



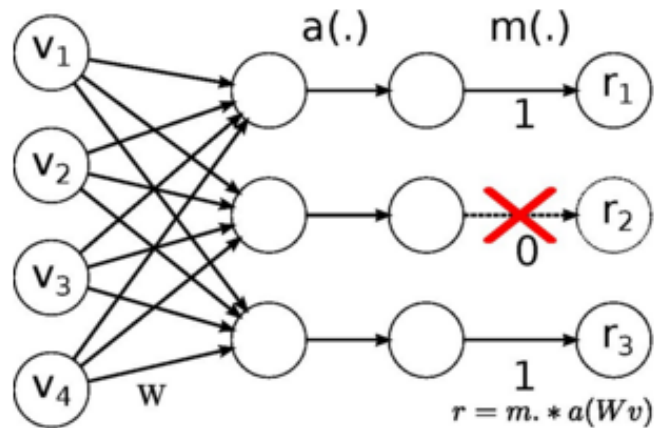
(b) After applying dropout.

DropOut increases the generalization performance of the neural network by **restricting** the model capacity!

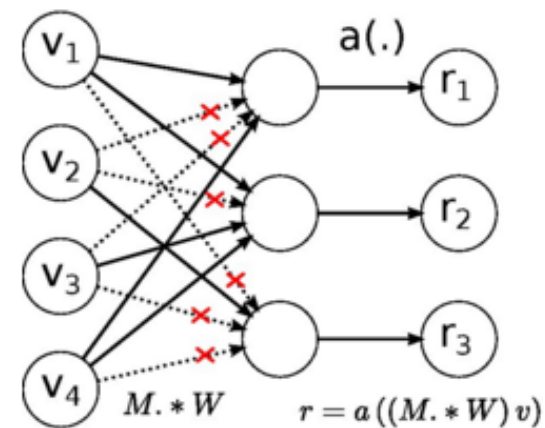
DropConnect



No-Drop Network



DropOut Network



DropConnect Network

Instead of turning the neurons off (DropOut), DropConnect **disconnects** the connections between neurons.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;

Parameters to be learned: γ, β

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

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

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

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

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

Benefits of BN

1. Increase learning rate
2. Remove Dropout
3. Reduce L2 weight decay
4. Accelerate learning rate decay
5. Remove Local Response Normalization

Book review

1. Parameter Norm Penalties
2. Dataset Augmentation
3. Noise Robustness: to input, weights, and output
4. Semi-Supervised Learning = learning a **representation**
5. Multitask learning
6. Early Stopping
7. Parameter Tying and Parameter Sharing
8. Sparse representation
9. Bagging and Other Ensemble Methods
10. Dropout
11. Adversarial Training

Parameter Norm Penalties

- Many regularization approaches are based on limiting the **capacity** of models by adding a parameter norm penalty to the objective function.

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\boldsymbol{\theta})$$

- L2 parameter regularization:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^\top \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

- L1 parameter regularization:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \|\mathbf{w}\|_1 + J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

Dataset Augmentation

- The best way to make a machine learning model generalize better is to train it on **more data**.
- Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create **fake data** and add it to the training set.
- Label preserving transformation is used.
- Injecting noise in the input to a neural network can also be seen as a form of data augmentation.
- Often, hand-designed dataset augmentation schemes can dramatically reduce the generalization error of a machine learning technique.

Noise Robustness

- In the general case, it is important to remember that noise injection can be much **more powerful** than simply shrinking the parameters, especially when the noise is added to the hidden units.

$$\begin{aligned}\tilde{J}_{\mathbf{W}} &= \mathbb{E}_{p(\mathbf{x}, y, \epsilon_{\mathbf{W}})} \left[(\hat{y}_{\epsilon_{\mathbf{W}}}(\mathbf{x}) - y)^2 \right] \\ &= \mathbb{E}_{p(\mathbf{x}, y, \epsilon_{\mathbf{W}})} \left[\hat{y}_{\epsilon_{\mathbf{W}}}^2(\mathbf{x}) - 2y\hat{y}_{\epsilon_{\mathbf{W}}}(\mathbf{x}) + y^2 \right].\end{aligned}$$

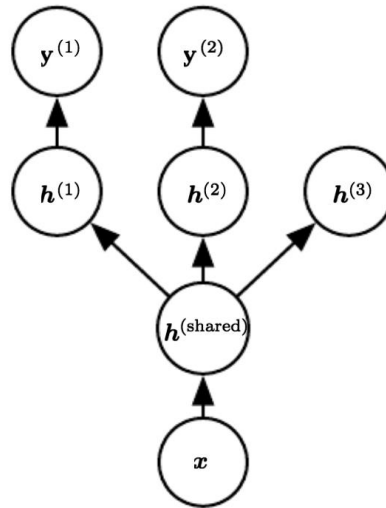
- Another way that noise has been used is by adding it to the **weights**.
- Most datasets have some amount of mistakes in the y labels. **Label-smoothing** can be used in this regard.

Semi-Supervised Learning

- In the paradigm of semi-supervised learning, both **unlabeled examples** and **labeled examples** are used to.
- In the context of deep learning, semi-supervised learning usually refers to learning a **representation**.
 - The goal is to learn a **representation** so that examples from the same class have similar representations.
 - Unsupervised learning can provide useful cues for how to group examples in representation space.

Multi-Task Learning

- Multi-task learning is a way to improve generalization by pooling the examples arising out of several tasks.

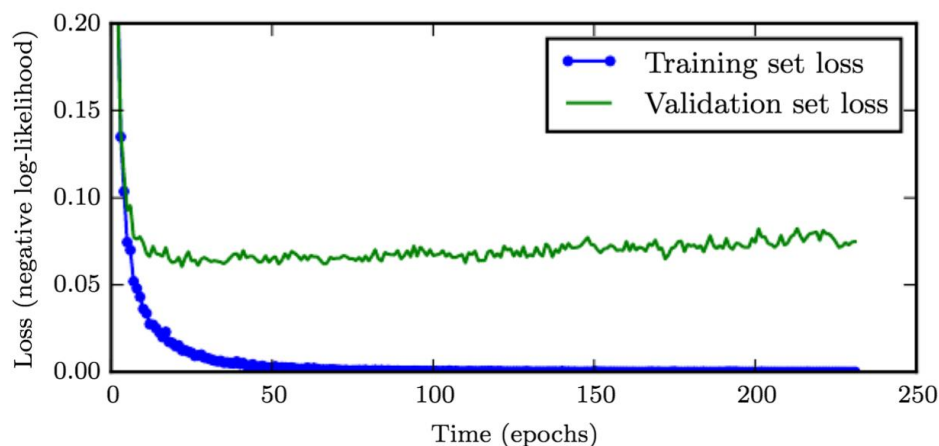


- Learning shared representation is a key point in multi-task learning.
- From the point of view of deep learning, the underlying prior belief is the following:

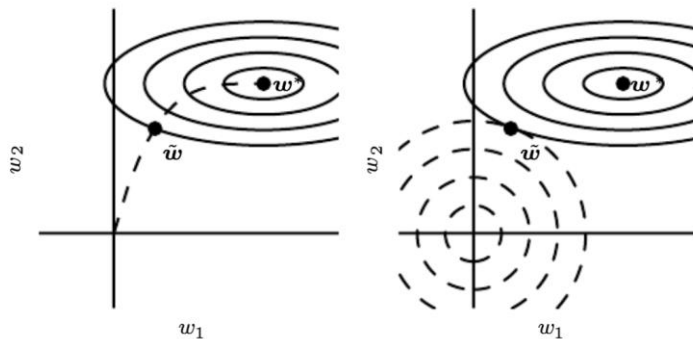
"Among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks."

Early Stopping

- When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but **validation set error** begins to rise again.



- Early stopping acts as a regularizer:



Parameter Tying and Parameter Sharing

- A common type of dependency that we often want to express is that certain parameters should be close to one another.
- Parameter Tying
 - Consider the following scenario: we have two models performing the same classification task (with the same set of classes) but with **different input distributions**.
- Parameter Sharing
 - By far the most popular and extensive used of parameter sharing occurs in **convolutional neural networks** (CNNs).

Sparse Representations

- **Representational sparsity** describes a representation where many of elements are zero.

- Sparse weights (L1 decay):

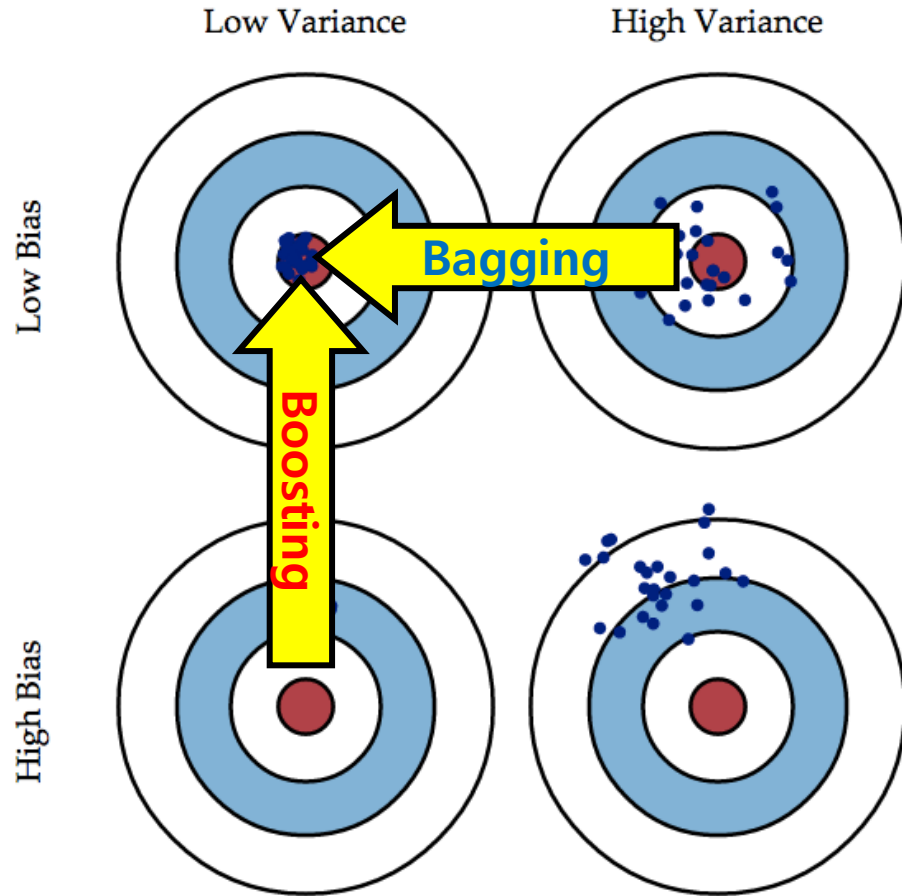
$$\begin{array}{ccc} \begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} & = & \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix} \\ \mathbf{y} \in \mathbb{R}^m & & \mathbf{A} \in \mathbb{R}^{m \times n} \quad \mathbf{x} \in \mathbb{R}^n \end{array}$$

- Sparse activations

$$\begin{array}{ccc} \begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} & = & \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \\ \mathbf{y} \in \mathbb{R}^m & & \mathbf{B} \in \mathbb{R}^{m \times n} \quad \mathbf{h} \in \mathbb{R}^n \end{array}$$

- Norm penalty regularization of representations is performed by adding to the loss function a norm penalty on the **representations**.

Bagging and Other Ensemble Methods



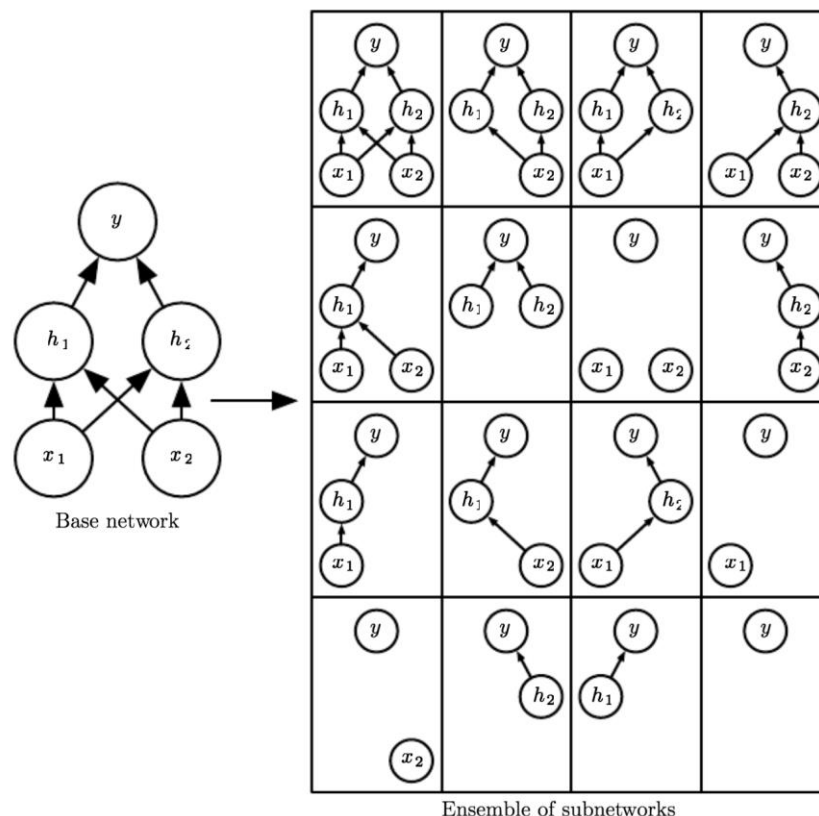
Bagging

- Generate **multiple sets of training data** (with bootstrapping), multiple predictions and combine them with averaging prediction
- **Reduce overall variance** using multiple low-bias models
- Example: Random Forest

Boosting

- Generate **multiple weak learners** and combine them to make a strong learner
- **Reduce overall bias** using multiple weak learners
- Example: Boosting, AdaBoost




Dropout



One of the key insights of dropout is that training a network with stochastic behavior and making predictions by averaging over multiple stochastic decisions implements a form of bagging with parameter sharing.

Adversarial Training

- In many cases, neural networks have begun to reach human performance.
- Examples that a human cannot tell the difference results in highly different predictions. These examples are called the **adversarial examples**.

	$+ .007 \times$		$=$	
x		$\text{sign}(\nabla_x J(\theta, x, y))$		$x + \epsilon \text{sign}(\nabla_x J(\theta, x, y))$
$y = \text{"panda"}$		"nematode"		"gibbon"
w/ 57.7%		w/ 8.2%		w/ 99.3 %
confidence		confidence		confidence

- Adversarial training is done via **virtual adversarial examples**.

Optimization methods

Sungjoon Choi
(sungjoon.choi@cpslab.snu.ac.kr)

Gradient descent?



Gradient descent?

There are **three** variants of gradient descent

Differ in **how much** data we use to compute gradient

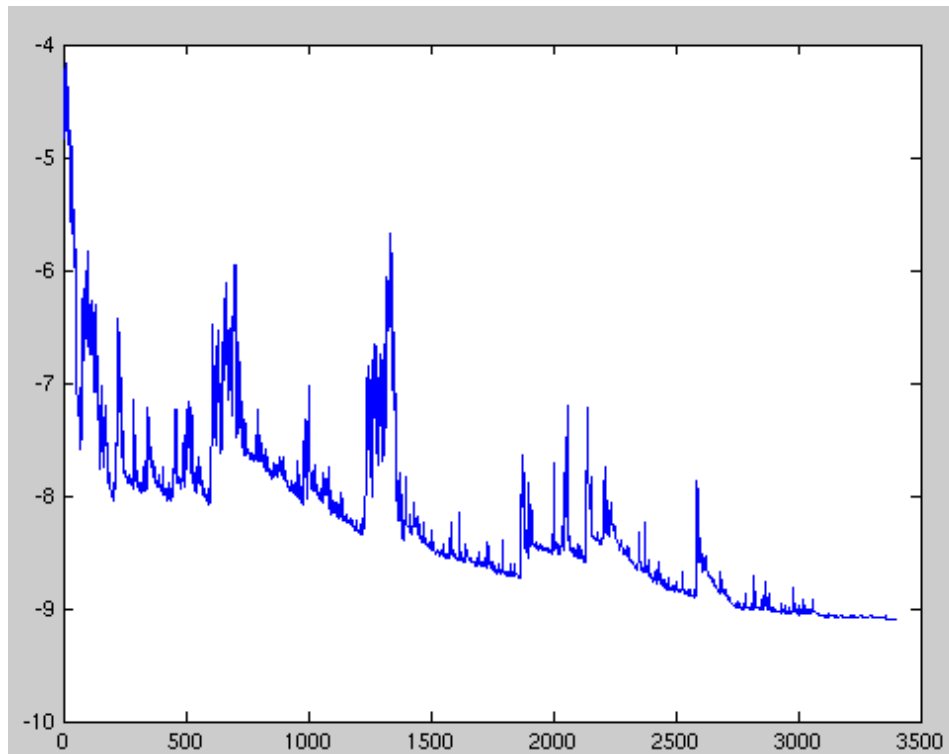
We make a **trade-off** between the accuracy and computing time

Batch gradient descent

In batch gradient descent, we use the entire training dataset to compute the gradient.

Stochastic gradient descent

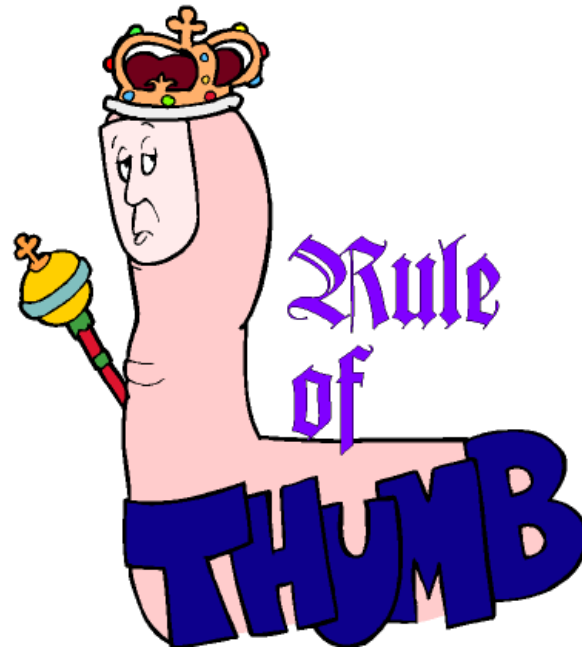
In stochastic gradient descent (SGD), the gradient is computed from each training sample, one by one.



Mini-batch gradient decent

In mini-batch gradient decent, we take the best of both worlds.

Common mini-batch sizes range between 50 and 256 (but can vary).



Challenges

Choosing a proper learning rate is cumbersome.

→ Learning rate schedule

Avoiding getting trapped in suboptimal local minima

Momentum

```
tf.train.MomentumOptimizer.__init__(learning_rate,  
momentum, use_locking=False, name='Momentum')
```

Construct a new Momentum optimizer.

Args:

- `learning_rate`: A `Tensor` or a floating point value. The learning rate.
- `momentum`: A `Tensor` or a floating point value. The momentum.
- `use_locking`: If `True` use locks for update operations.
- `name`: Optional name prefix for the operations created when applying gradients. Defaults to "Momentum".

Nesterov accelerated gradient



qixianbiao commented on 23 Dec 2015



Does anybody have experience about "momentum" and "nesterov momentum"?

does the "nesterov momentum" perform much better than than "momentum"?



eduardo-elizondo commented on 8 Feb



@qixianbiao working on this right now.

Adagrad

It adapts the learning rate to the parameters,

```
tf.train.AdagradOptimizer.__init__(learning_rate,  
initial_accumulator_value=0.1, use_locking=False, name='Adagrad')
```

Construct a new Adagrad optimizer.

Args:

- `learning_rate`: A Tensor or a floating point value. The learning rate.
- `initial_accumulator_value`: A floating point value. Starting value for the accumulators, must be positive.
- `use_locking`: If True use locks for update operations.
- `name`: Optional name prefix for the operations created when applying gradients. Defaults to "Adagrad".

Adadelta

Adadelta is an extension of Adagrad that seeks to reduce its **monotonically decreasing** learning rate.

It restricts the window of accumulated past gradients to some fixed size w .

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

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\Delta\theta_t^2$$

No learning rate!

$$\theta_{t+1} = \theta_t - \frac{\sqrt{E[\Delta\theta^2]_t + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Exponential moving average

Exponential Moving Average (EMA)

Daily Chart - eBay (EBAY)



www.OnlineTradingConcepts.com - All Rights Reserved

Created with TradeStation

RMSprop

```
tf.train.RMSPropOptimizer.__init__(learning_rate, decay=0.9,  
momentum=0.0, epsilon=1e-10, use_locking=False,  
name='RMSProp')
```

Construct a new RMSProp optimizer.

Args:

- `learning_rate`: A Tensor or a floating point value. The learning rate.
- `decay`: Discounting factor for the history/coming gradient
- `momentum`: A scalar tensor.
- `epsilon`: Small value to avoid zero denominator.
- `use_locking`: If True use locks for update operation.
- `name`: Optional name prefix for the operations created when applying gradients. Defaults to "RMSProp".

Adam

Adaptive Moment Estimation (Adam) stores both exponentially decaying average of past gradients and squared gradients.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Momentum

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

**Running average of
gradient squares**

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} m_t$$

Adam

Ada
exp
and

both
clients

```
tf.train.AdamOptimizer.__init__(learning_rate=0.001,  
beta1=0.9, beta2=0.999, epsilon=1e-08, use_locking=False,  
name='Adam')
```

Construct a new Adam optimizer.

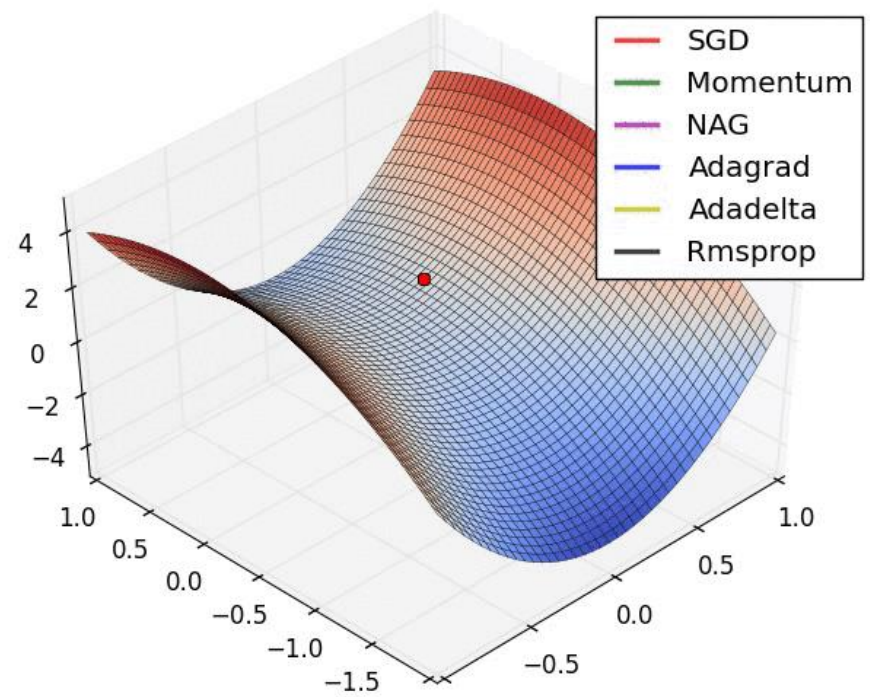
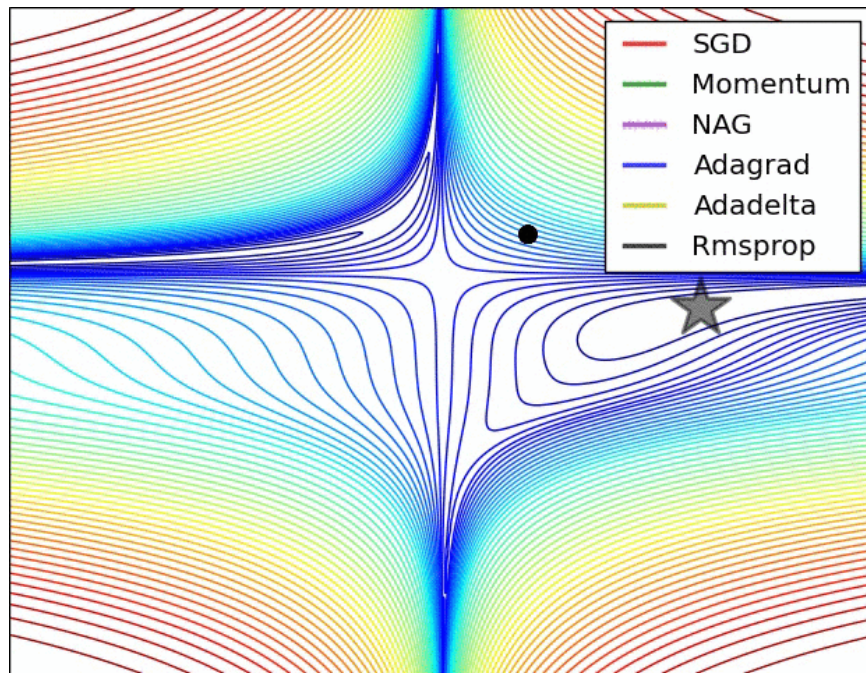
Initialization:

```
m_0 <- 0 (Initialize initial 1st moment vector)  
v_0 <- 0 (Initialize initial 2nd moment vector)  
t <- 0 (Initialize timestep)
```

The update rule for **variable** with gradient **g** uses an optimization described at the end of section 2 of the paper:

```
t <- t + 1  
lr_t <- learning_rate * sqrt(1 - beta2^t) / (1 - beta1^t)  
  
m_t <- beta1 * m_{t-1} + (1 - beta1) * g  
v_t <- beta2 * v_{t-1} + (1 - beta2) * g * g  
variable <- variable - lr_t * m_t / (sqrt(v_t) + epsilon)
```

Visualization



Which optimizer to use?

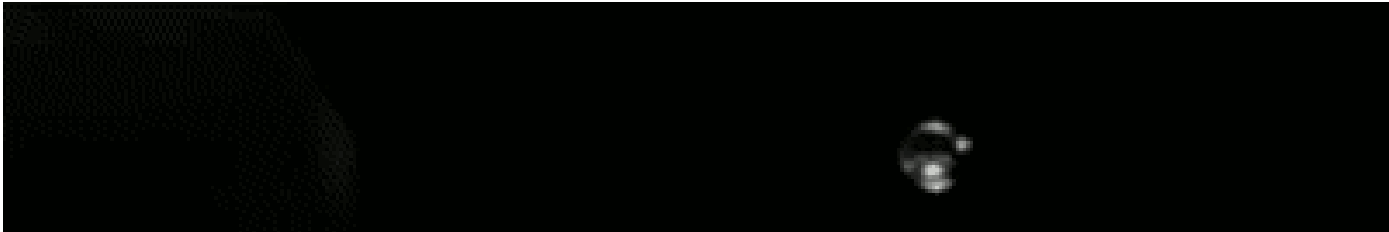
Use adaptive learning rate methods (Adagrad, Adadelata, RMSprop, Adam)

Adagrad decreases the learning rate of more activated features.

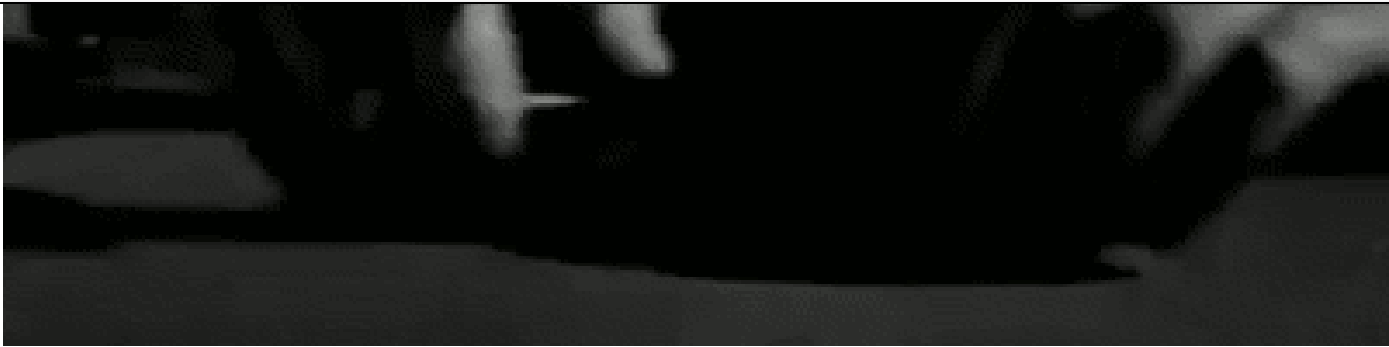
RMSprop extends Adagrad by dealing with radically diminishing learning rates (=Adadelata).

Adam adds bias-correction and momentum to RMSprop.

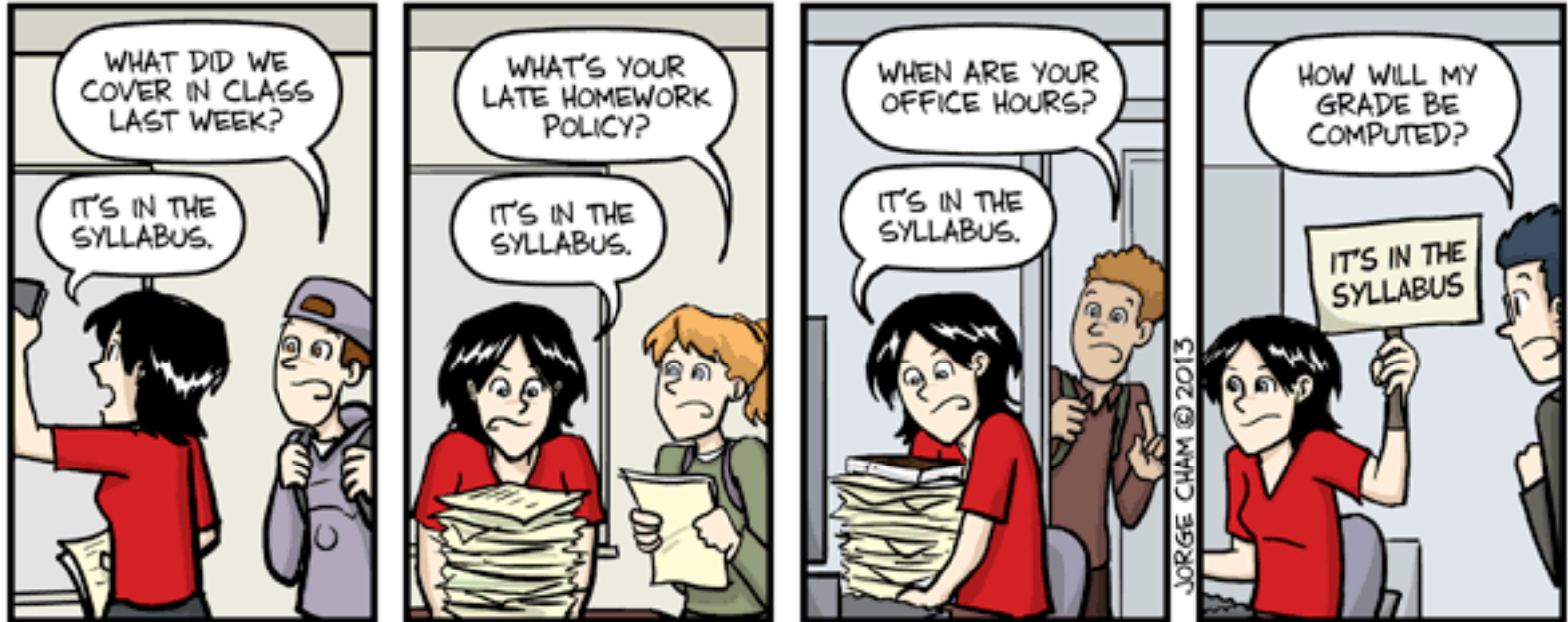
Shuffling



```
for i in range(num_batch):  
    if 0: # Using tensorflow API  
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)  
    else: # Random batch sampling  
        randidx = np.random.randint(training.shape[0], size=batch_size)  
        batch_xs = training[randidx, :]  
        batch_ys = trainlabel[randidx, :]  
  
    # Fit training using batch data  
    sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys})
```



Curriculum Learning



IT'S IN THE SYLLABUS

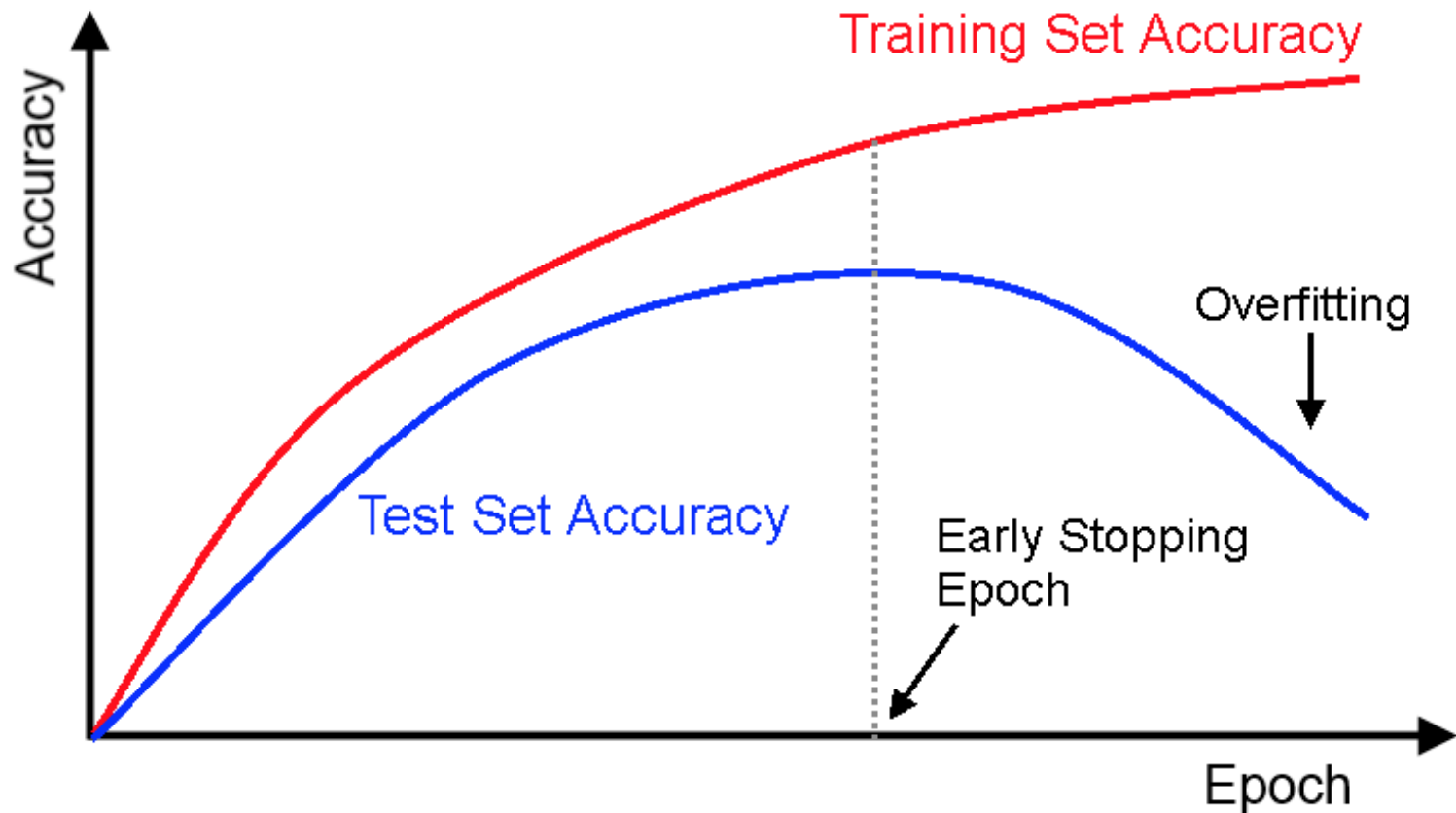
This message brought to you by every instructor that ever lived.

WWW.PHDCOMICS.COM

Batch normalization



Early stopping



Gradient noise



Learning rate

