Training neural networks

Our main tool for training neural networks is ***backpropagation*** (i.e., a clever implementation of ***gradient descent***)

However, the loss functions we are aiming to minimize are particularly challenging

- Nonconvex
  - local minima
  - saddle points
- vanishing gradients
  - particularly at initial layers
- sensitive to initialization

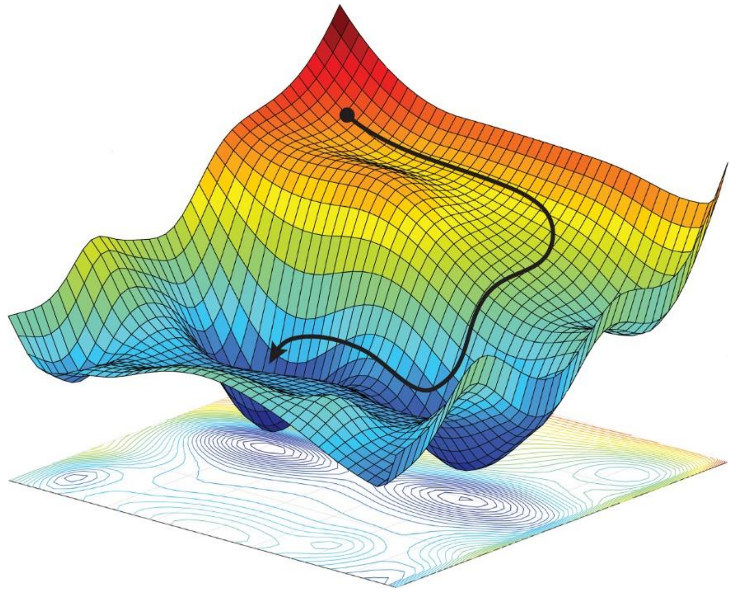


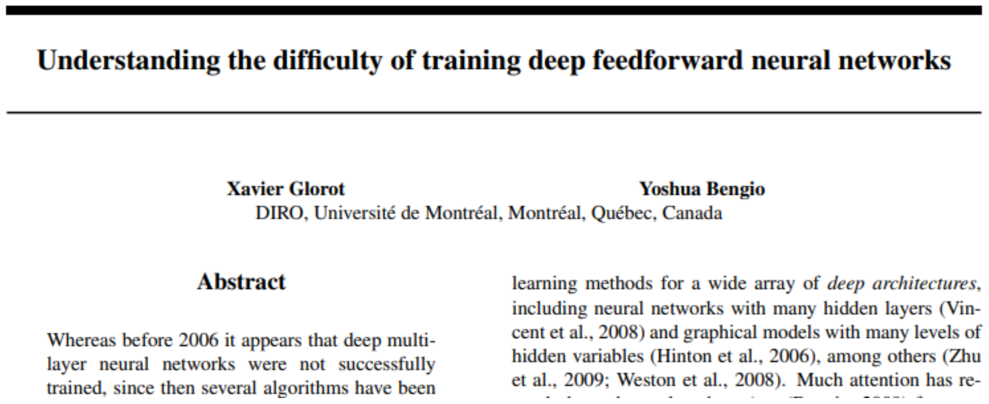
Image Source: A. Amini et al. "Spatial Uncertainty Sampling for End-to-End Control". NeurIPS Bayesian Deep Learning 2018

Initialization

Classical rule of thumb: Initialize by setting

$$w_{ij}^{(\ell)} \sim U\left[-\frac{1}{d^{(\ell-1)}}, \frac{1}{d^{(\ell-1)}}\right]$$

2010



Understanding the difficulty of training deep feedforward neural networks

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

Abstract

Whereas before 2006 it appears that deep multi-layer neural networks were not successfully trained, since then several algorithms have been

learning methods for a wide array of *deep architectures*, including neural networks with many hidden layers (Vincent et al., 2008) and graphical models with many levels of hidden variables (Hinton et al., 2006), among others (Zhu et al., 2009; Weston et al., 2008). Much attention has re-

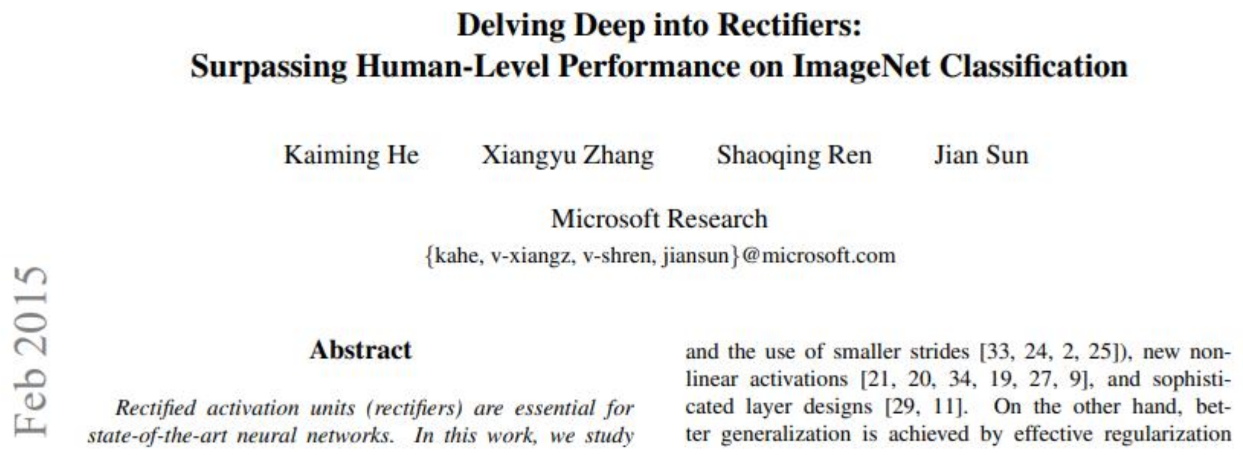
Initialization

The classical rule of thumb was designed for a sigmoid/tanh nonlinearity

As we will soon see, modern architectures are more likely to use a "rectified linear unit" or ReLU nonlinearity

"He" initialization:

$$w_{ij}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{\sqrt{d^{(\ell-1)}}}\right)$$



Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun

Microsoft Research

{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Feb 2015

Abstract

Rectified activation units (rectifiers) are essential for state-of-the-art neural networks. In this work, we study

and the use of smaller strides [33, 24, 2, 25]), new non-linear activations [21, 20, 34, 19, 27, 9], and sophisticated layer designs [29, 11]. On the other hand, better generalization is achieved by effective regularization

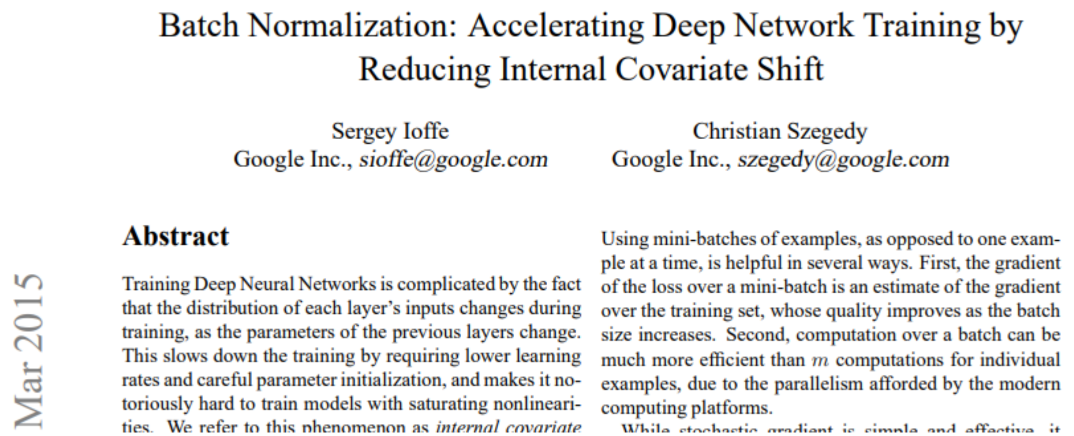
Feature normalization

Feature normalization is a ***critical*** preprocessing step

Standard normalization: For each feature we compute

$$x'[j] = \frac{x[j] - \mu}{\sigma} \qquad \mu = \frac{1}{n}\sum_{i=1}^{n} x_i[j]$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i[j] - \mu)^2$$



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Mar 2015

Abstract

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

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

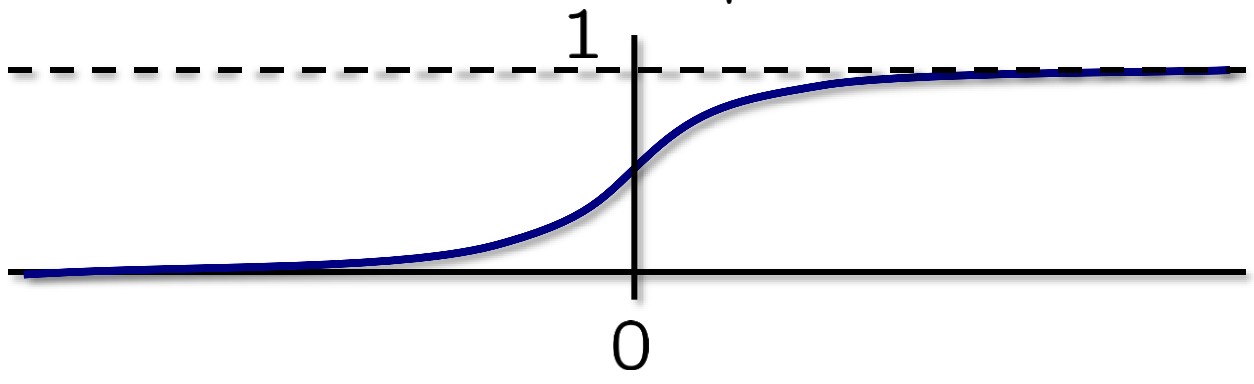
While stochastic gradient is simple and effective, it

Architectural improvements

Can we change the structure of our network to address the “vanishing gradient” problem?

- choose activation functions and a loss function that make optimization easier
- want to avoid situations where the gradient is tiny, but you are far from the true solution
  - “saturation”

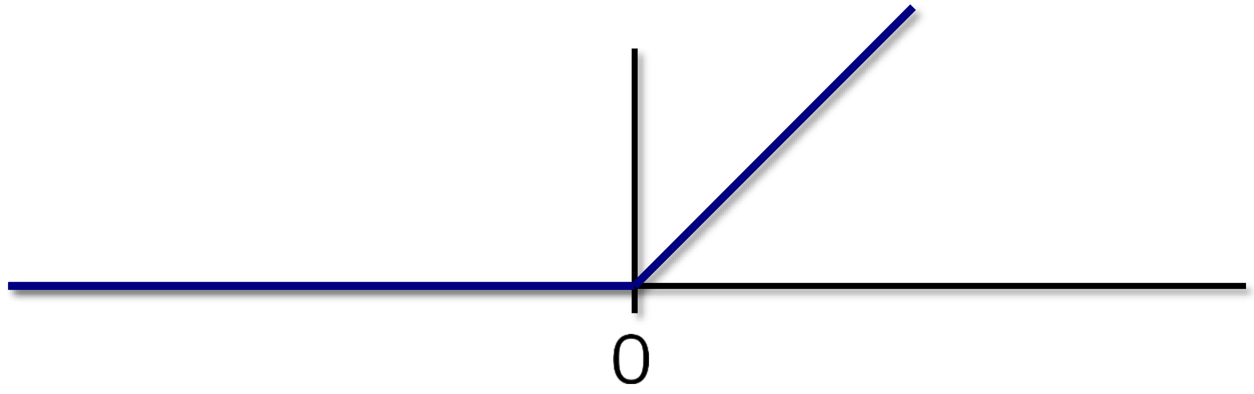
$$g(t) = \frac{1}{1+e^{-t}}$$



Choice of hidden units

For hidden units in a deep architecture the most common choice is the ***rectified linear unit (ReLU)***

ReLUs use the activation function $g(t) = \max\{0, t\}$



Note that the derivative is (usually) very easy to calculate

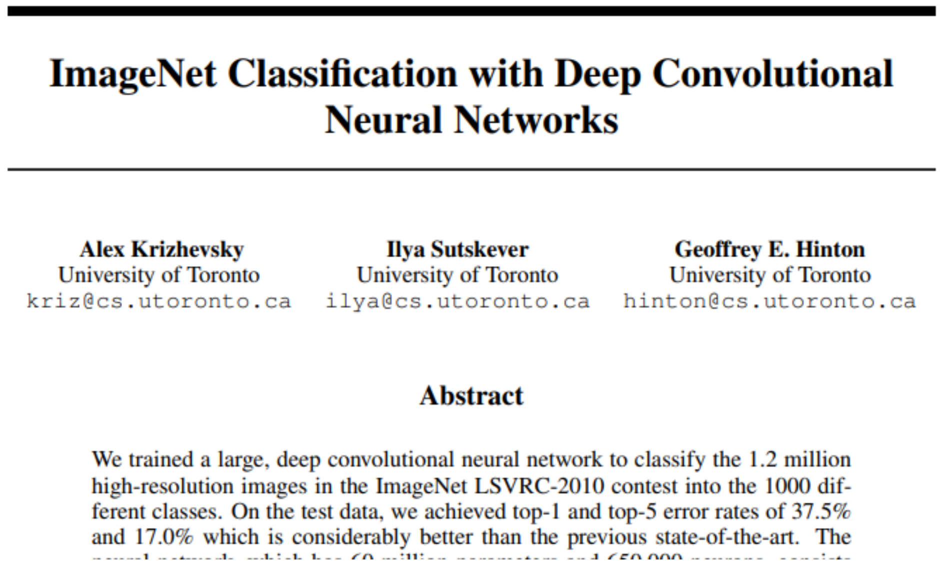
Not differentiable at zero, but generalizations exist, or we can use "subgradient descent"

AlexNet

2012 ImageNet Large Scale Visual Recognition Challenge

- AlexNet: Top-5 error rate @15.3% (next closest @ 26.2%!)
- ReLU was one of the key innovations
- Subsequent variations (e.g., He et al., 2015)

$$g(t) = \max\{0, t\} + \alpha \min\{0, t\}$$



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The

How many layers/units?

- Choosing the number of hidden layers and number of units in each layer is more art than science
- In general, it seems to be better to have too many parameters rather than too few, and rely on regularization to avoid overfitting
- Additional strategies to control overfitting include
  - dropout
  - early stopping
  - dataset augmentation

Regularization

To avoid overfitting the data, regularization is, again, ***crucial***

Choose $\boldsymbol{\theta}$ to minimize

$$L(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2 \quad \text{or} \quad L(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_1$$

This encourages small (or zero) weights

Note that when the weights are small, the effect of the nonlinear functions can go away g, h

If we constrain all weights to be small, the entire network can approximately reduce to a simple linear classifier

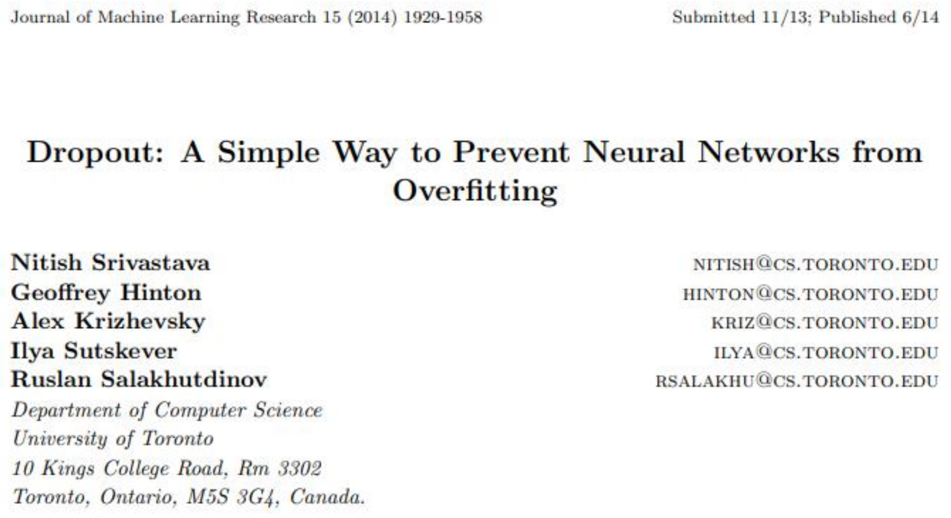
Encouraging small weights in the network reduces the effective number of degrees of freedom

Noise injection and dropout

In 1995, Bishop showed that “noise injection” (adding noise to the training data) is approximately equivalent to Tikhonov regularization

Dropout is a way to add noise into each layer

Set ${x'_j}^{(\ell)} = 0$ with prob p and ${x'_j}^{(\ell)} = \frac{{x_j}^{(\ell)}}{1-p}$ with prob $1 - p$

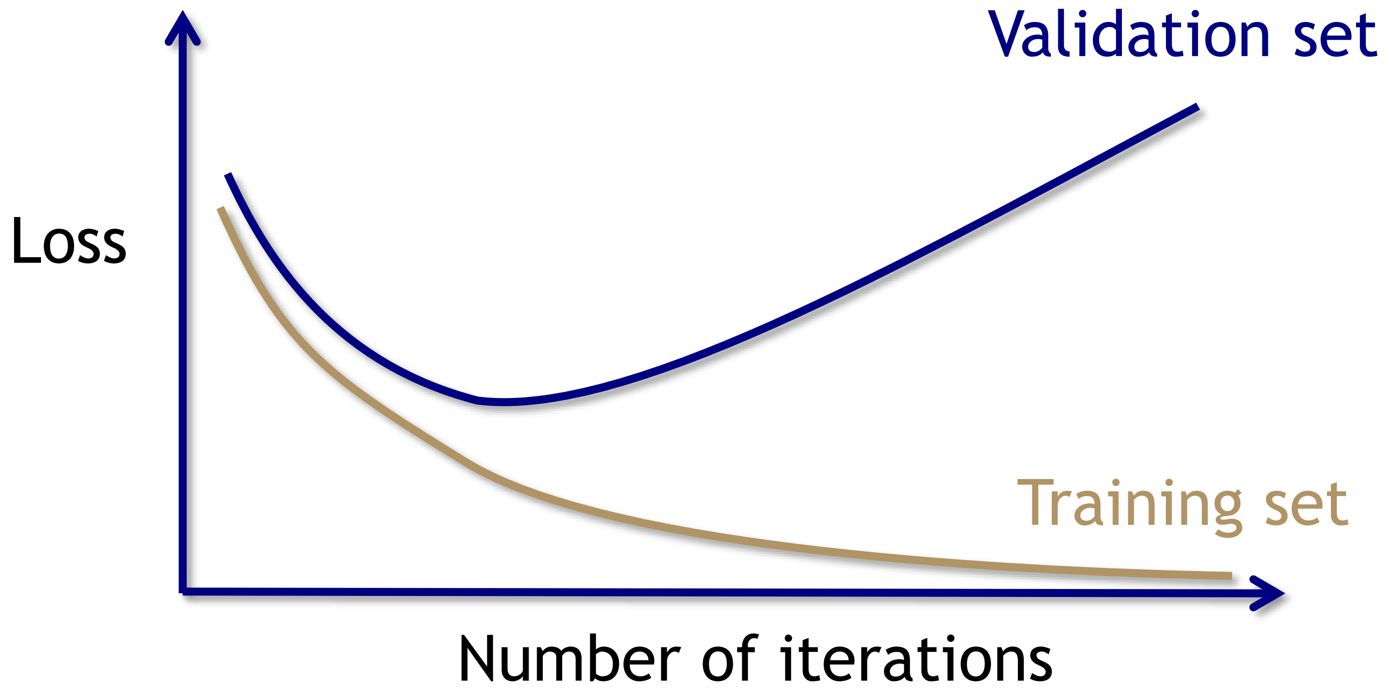

Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

Early stopping

When running gradient descent, you would think that more iterations would always be better…



Early stopping serves as another form of regularization

Dataset augmentation

Overfitting is fundamentally a problem of having too many parameters and not enough data

In many application areas, it can be possible to generate more training data on demand

For example, images can be

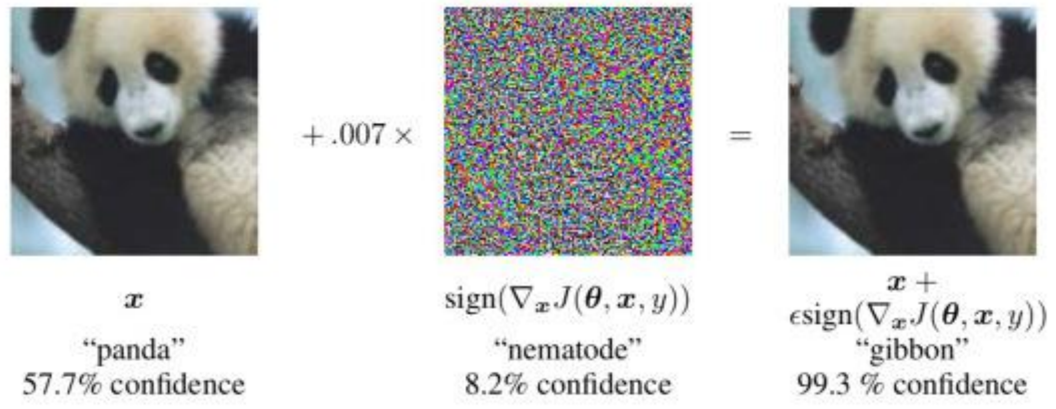
- translated
- rotated
- zoomed
- warped, occluded, and subjected to other distortions

without changing the correct class label

It may be possible to generate lots of extra training data…

Adversarial training

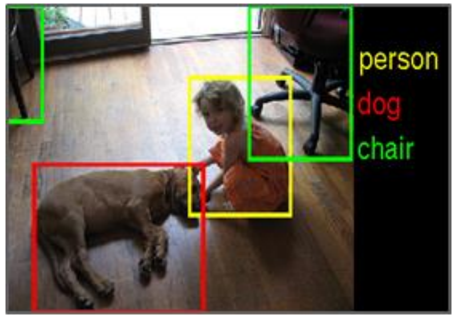
If you work at it, it is also possible to generate some rather strange training data…


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

This highlights some strange behavior, but also provides a way to generate augmented data that can help make deep networks much more robust

Large-scale gradient descent

Some of the most impressive results achieved by deep architectures have been on large-scale image datasets



Consider the ImageNet dataset

- 14 million images
- 1000 classes

Each gradient step is incredibly expensive…

- make each step faster
  - stochastic gradient descent, minibatch, rely on parallelism
- try to take fewer steps
  - use smarter optimization algorithms

Stochastic gradient descent

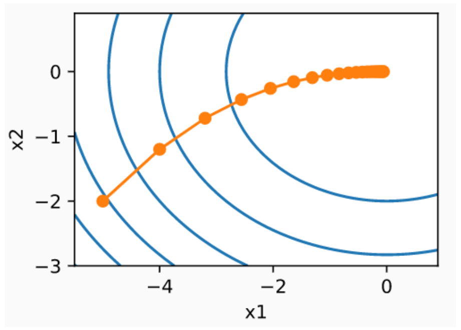
The best way to make gradient descent faster on large data sets is something we have already seen…

Stochastic gradient descent

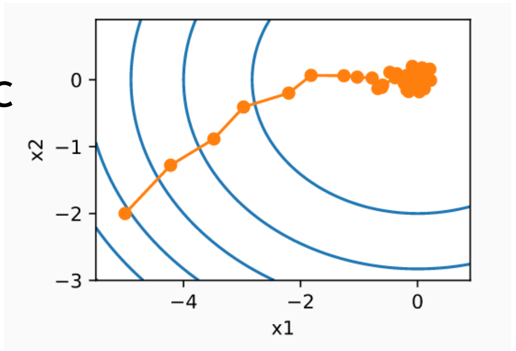
Estimate the gradient at each iteration by randomly sampling an element from the training data

This might result in an increase in the total number of iterations required to converge, but on big datasets simply computing the full gradient is wasteful/impossible

gradient descent



stochastic gradient descent



Minibatch

Vanilla stochastic gradient takes too many iterations and also fails to exploit the full power of vectorization/parallelism in modern computational architectures

The standard approach when working with large datasets is to form ***minibatches***

Stochastic gradient descent, but with a bigger subset of the data

Optimal size of minibatch will depend on problem and your computational resources

Momentum

We have talked a lot about gradient descent, but there has been a lot of research recently in other first-order variants that can often result in much faster convergence

Heavy ball method

(Gradient descent with momentum)

$$\boldsymbol{\theta}^{(r+1)} = \underbrace{\boldsymbol{\theta}^{(r)} - \alpha_r \nabla L(\boldsymbol{\theta}^{(r)})}_{\text{gradient update}} + \underbrace{\beta_r(\boldsymbol{\theta}^{(r)} - \boldsymbol{\theta}^{(r-1)})}_{\text{momentum}}$$

Momentum term can cancel out “oscillations” in certain dimensions, resulting in convergence in fewer iterations

See also ***Nesterov’s accelerated method***

Adaptive stepsizes

Another problem that can occur in stochastic gradient descent is that the gradient updates can be relatively ***sparse***

If you have to wait a long time for an update, you want to make the most of it

In AdaGrad (2011), you have a separate stepsize for each parameter and set

$$\alpha_r \propto \frac{1}{\sqrt{\epsilon + \sum g_r^2}}$$

The popular ADAM optimizer (2015) combines this idea with momentum

Structural constraints

In practice, another common strategy is to place constraints on the parameters

Force sets of parameters to be equal

This is the underlying idea of ***convolutional neural networks***, where the linear mapping to the hidden layers is of the form of a convolution

Weights can be modeled via circulant matrices
(same parameters for each row)

Convolutional neural networks

The neural net architecture you’ve most likely heard of in the news is the ***convolutional neural network (CNN)***

Makes the explicit assumption that the input is an image

Constrains the structure of the network in a sensible way to make learning the weights scalable to high-resolution images

Main insight: Translation invariance

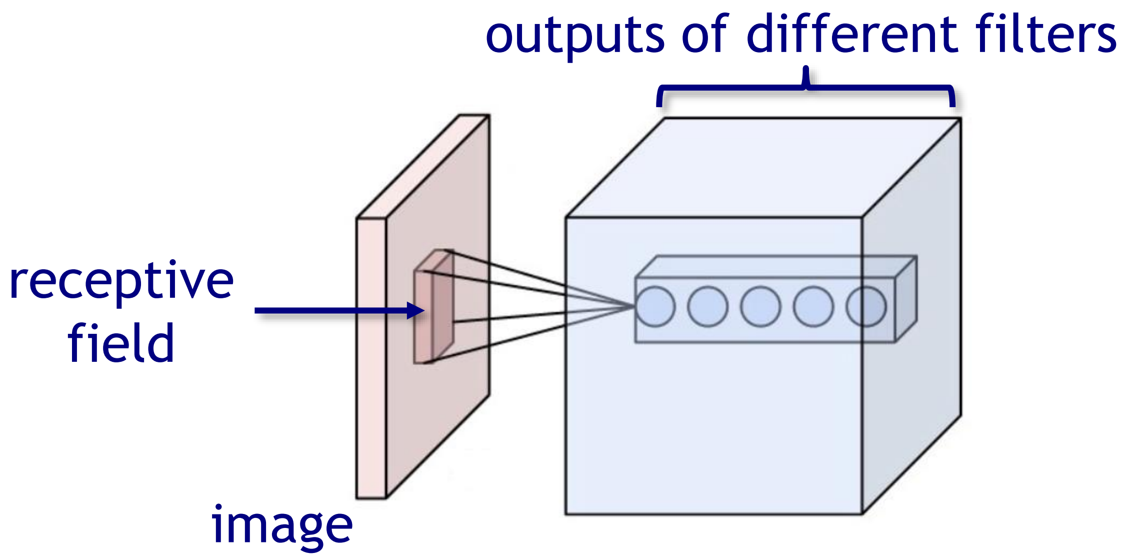
Convolutional layer

The weights we apply to raw pixels should be the same, no matter where the object is located in the image

We begin by convolving the image with M_1 different filters

- filters are localized, the filter weights represent the parameters to be learned

The output of these filters are then passed through a ReLU

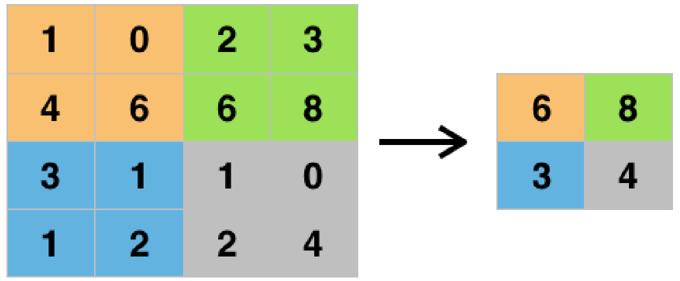


Pooling layer

The other key concept in CNNs is ***pooling*** (downsampling)

Given the output of a filter, we can downsample the output to produce a smaller number of coefficients for the next layer

Most common choice is known as ***max pooling***

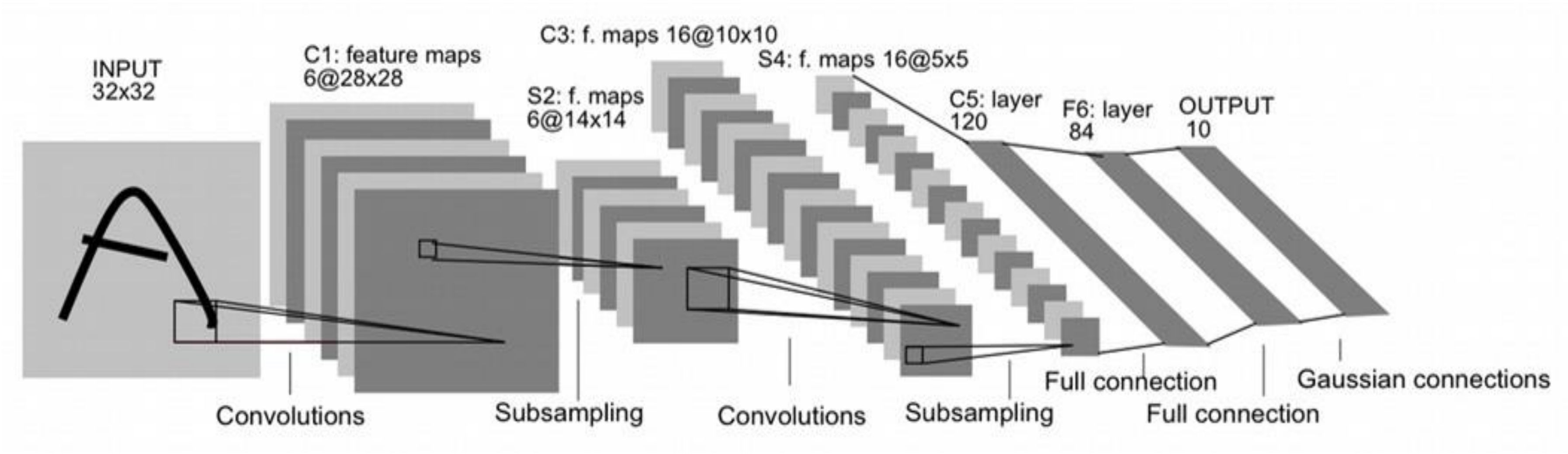


Intuition: the precise location of a feature is not important

A CNN will typically have a pooling layer after each convolution layer

Full CNN

Yann LeCun (1998)



Note that as you go deeper and subsample more, there are more filters/feature maps available

After a certain depth, the network simply consists of fully connected layers

Deep Learning: Theory

Overparameterization

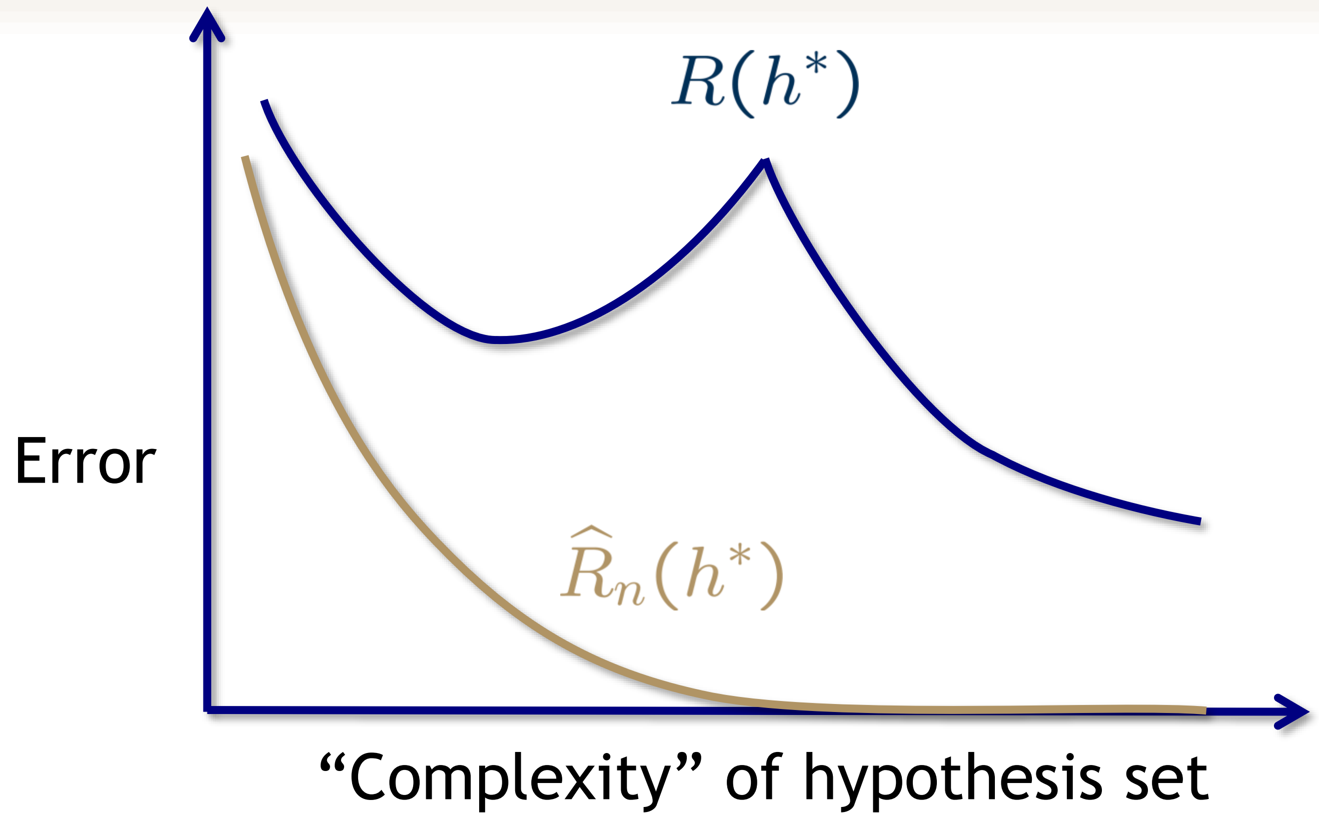
It is common to have many more parameters in a deep neural network than number of examples in the training set

How can this possibly avoid overfitting?

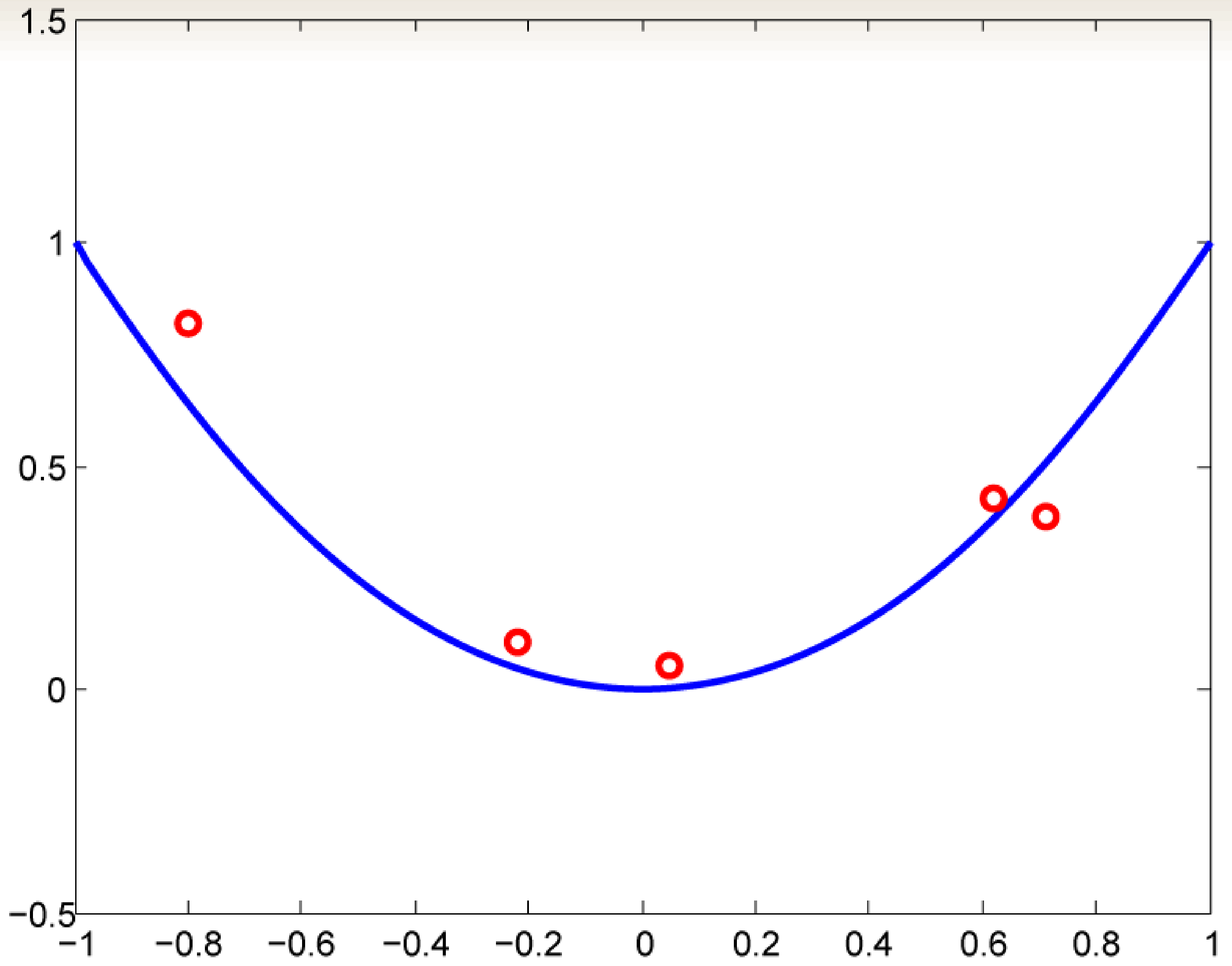
Regularization only tells part of the story

These networks ***do*** often achieve zero error on the training data, so they ***are*** overfitting in at least some capacity

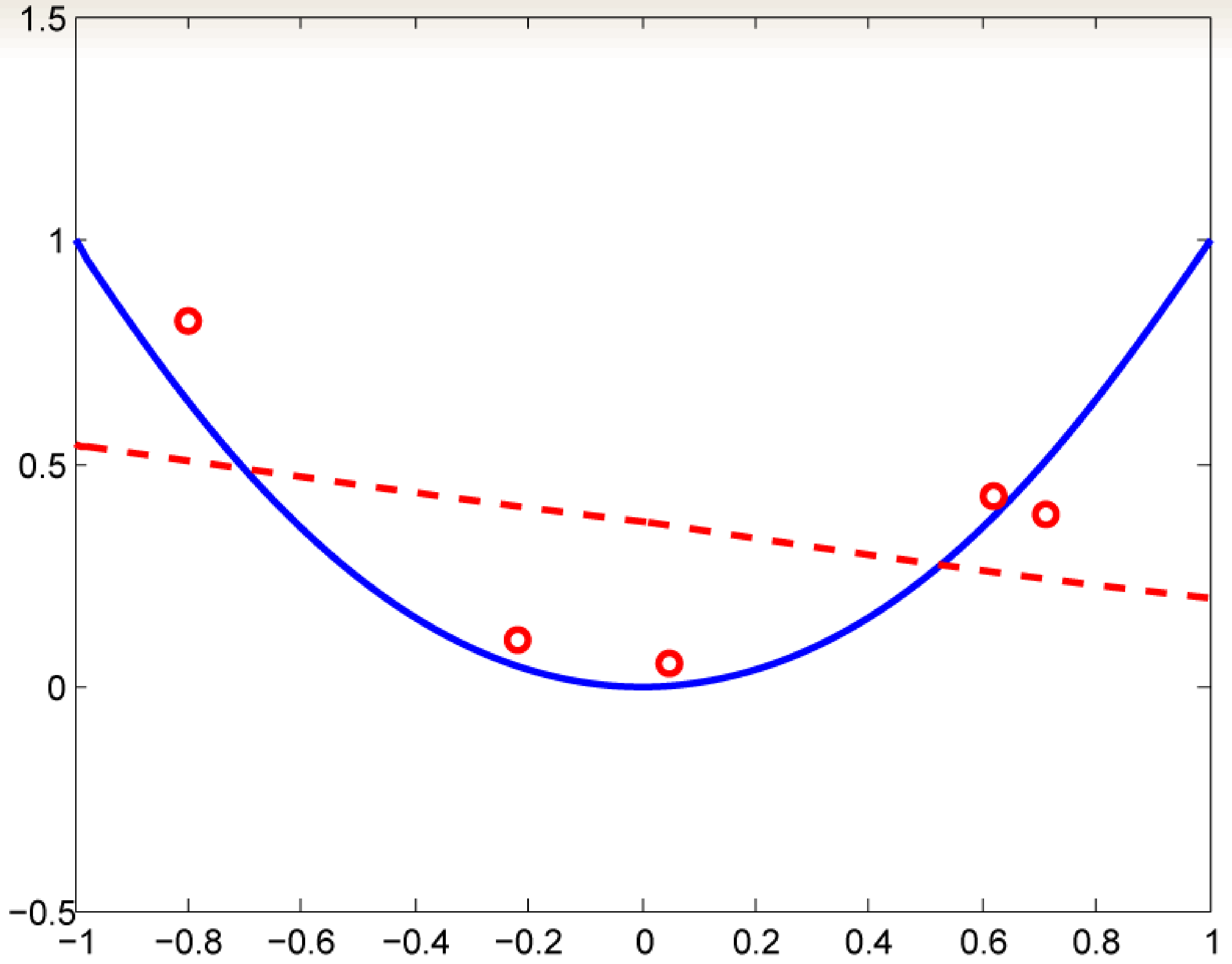
Double descent phenomenon



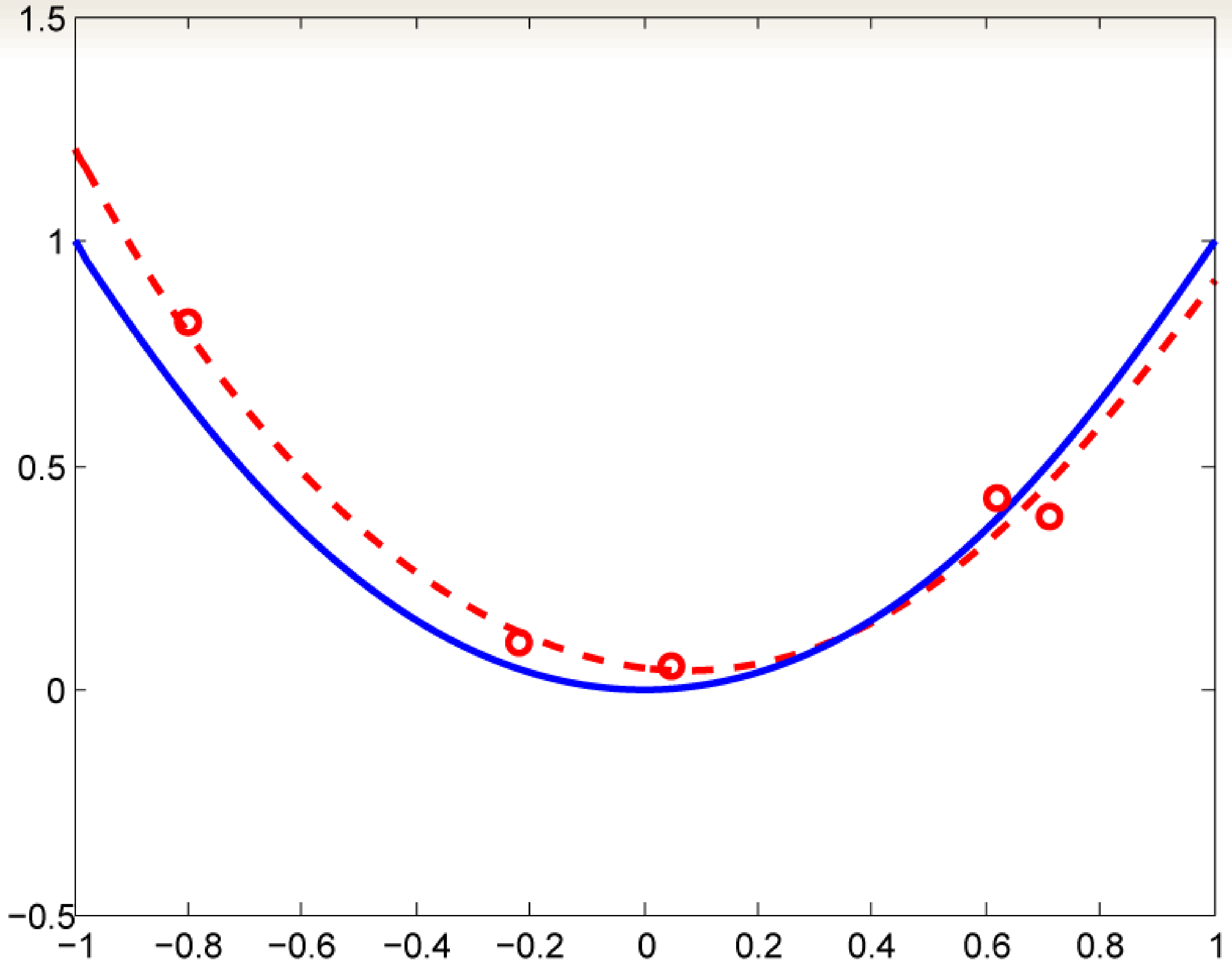
Overfitting



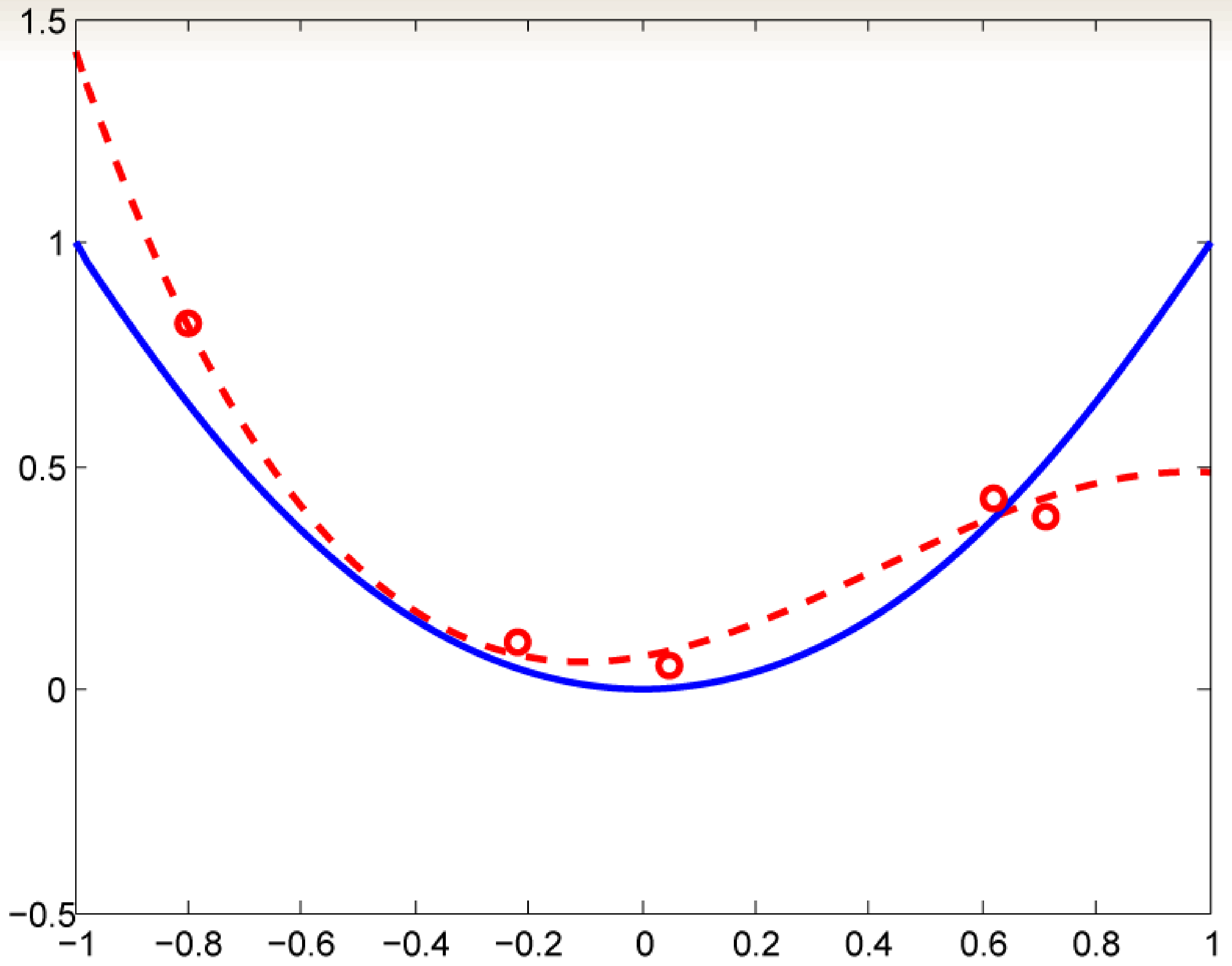
Overfitting



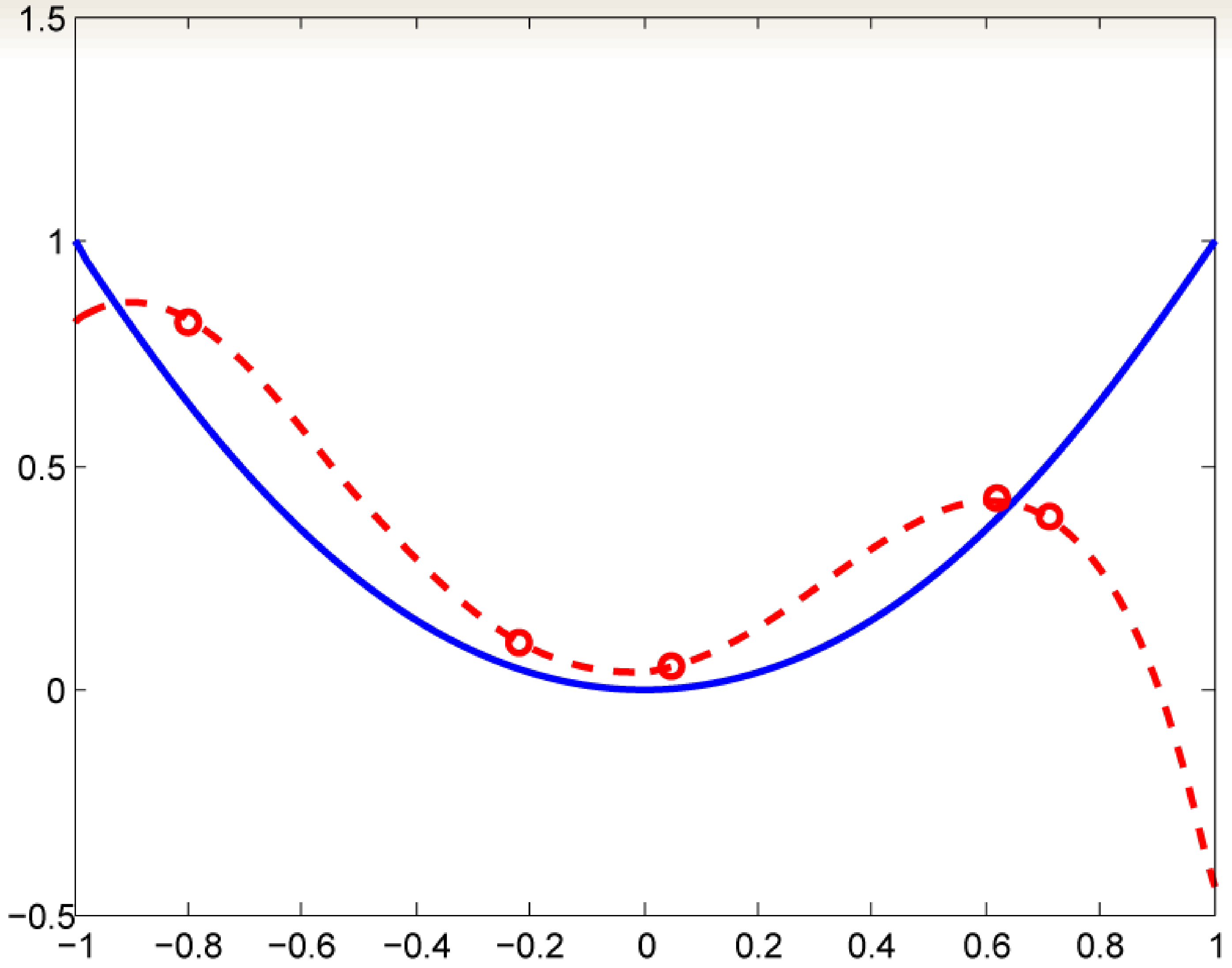
Overfitting



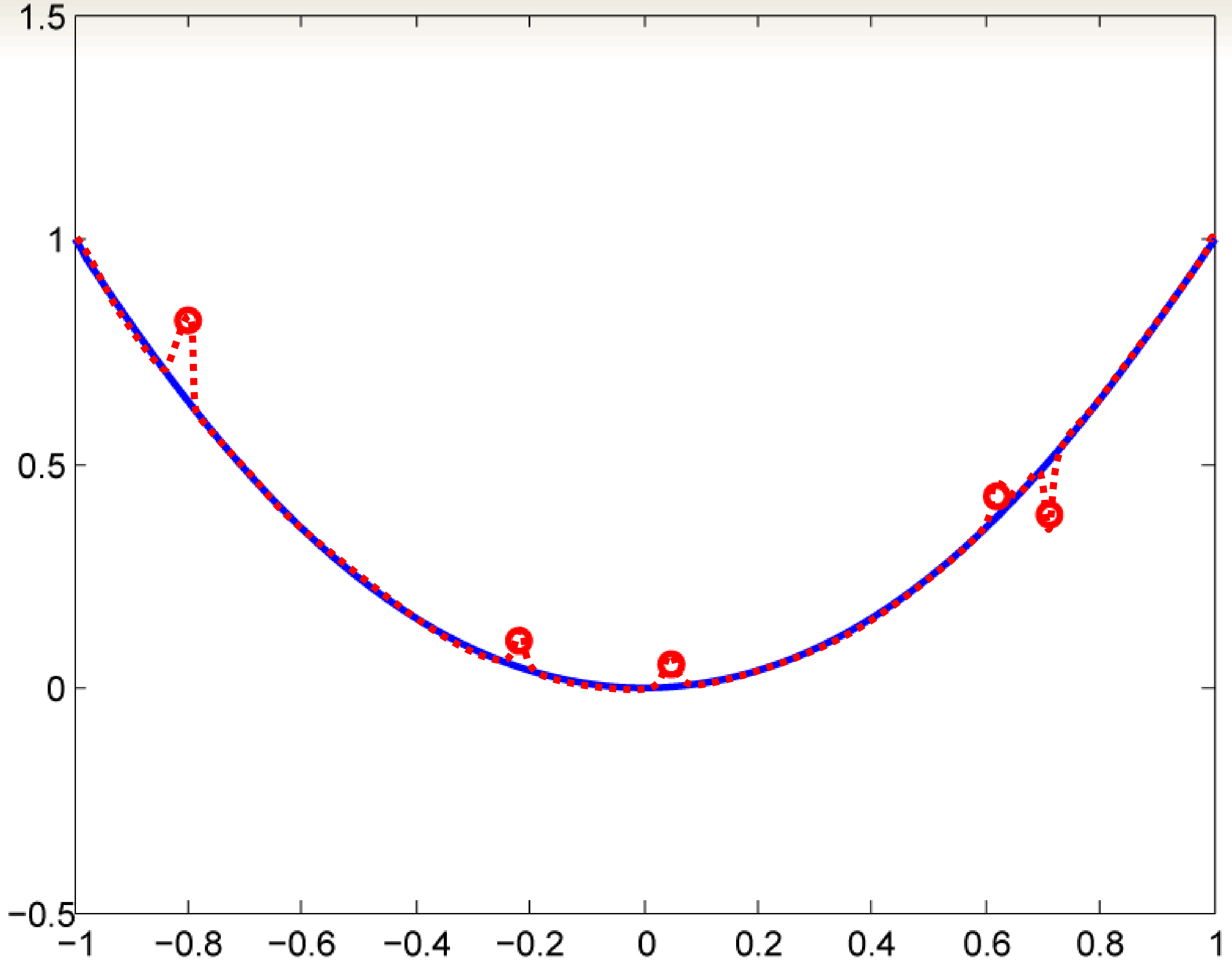
Overfitting



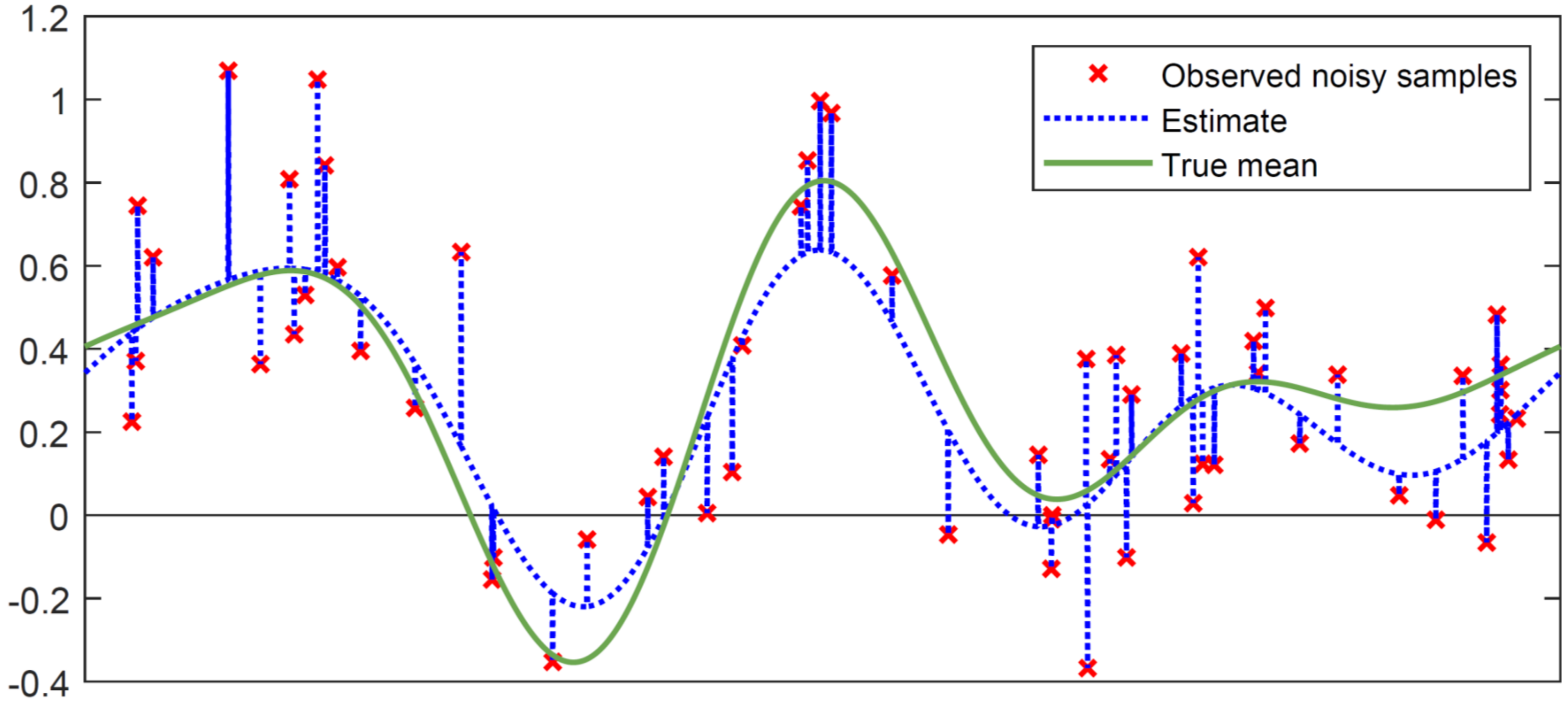
Overfitting



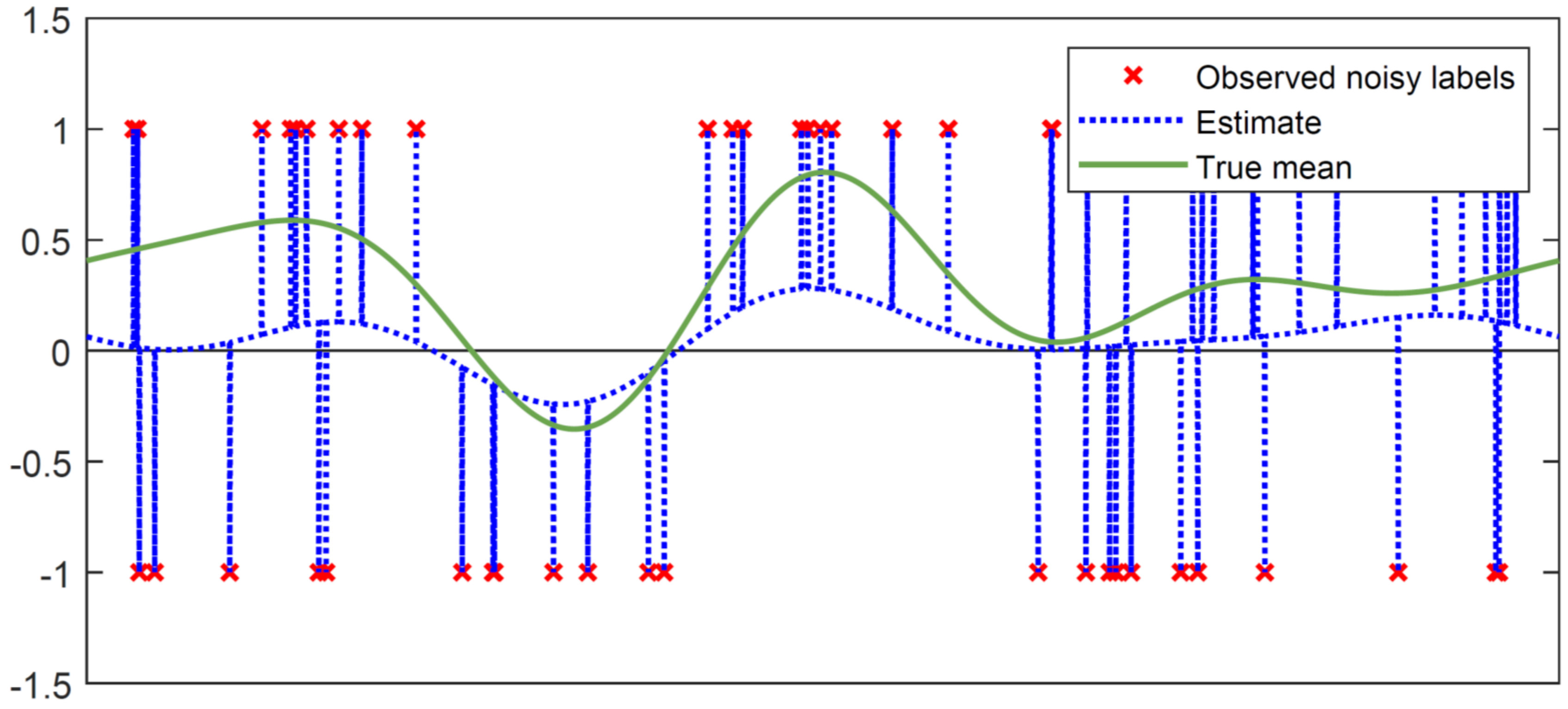
“Benign overfitting”



Real example: Sinusoidal features



Real example: Sinusoidal features


Observed noisy labels
Estimate
True mean