Generalization in Neural Networks

Alexander Trofimov
PhD, professor, NRNU MEPHI

lab@neuroinfo.ru
http://datalearning.ru

Course "Neural Networks"

April 2020

Neural Network Training Problem

Given:

$\mathscr{D} = \{(x^{(1)}, \sigma^{(1)}), ..., (x^{(n)}, \sigma^{(n)})\}$ — available data sample

$x^{(i)} = \left(x_1^{(i)}, ..., x_M^{(i)}\right)^T$ — i-th input vector, $i = 1, ..., n$

$\sigma^{(i)} = \left(\sigma_1^{(i)}, ..., \sigma_K^{(i)}\right)^T$ — i-th target vector, $i = 1, ..., n$

Problem:

The training of neural network F is the minimization of mean loss L over data sample $\mathscr{D}$:

$$E(w) = \frac{1}{n} \sum_{i=1}^{n} L\left(F, (x^{(i)}, \sigma^{(i)})\right) \to \min_w$$

If the neural network has acceptable error on training sample $\mathscr{D}$, does it mean acceptable error on some other data sample?

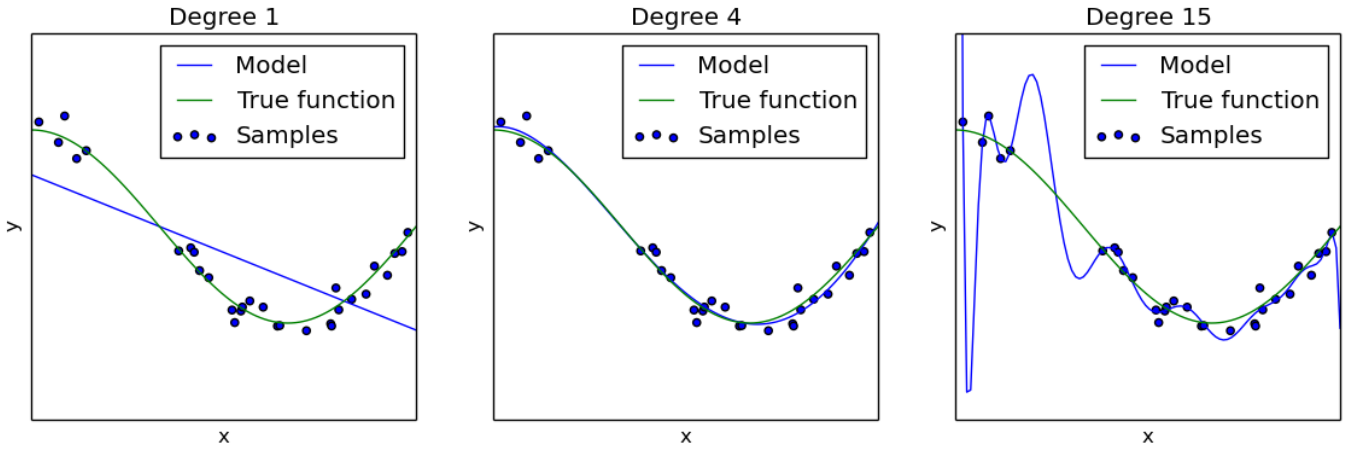
Generalization, Underfitting and Overfitting

Definition

Generalization of the model is its ability to accurately predict responses for previously unseen data

Underfitting: model cannot capture the underlying trend or patterns in the data

Overfitting: model describes random error or noise instead of the underlying relationship



Estimating the Generalization

How to measure generalization of a model?

To measure the generalization we need unseen data

Available data sample should be partitioned into "seen" and "unseen" parts

Training data is used to fit the model

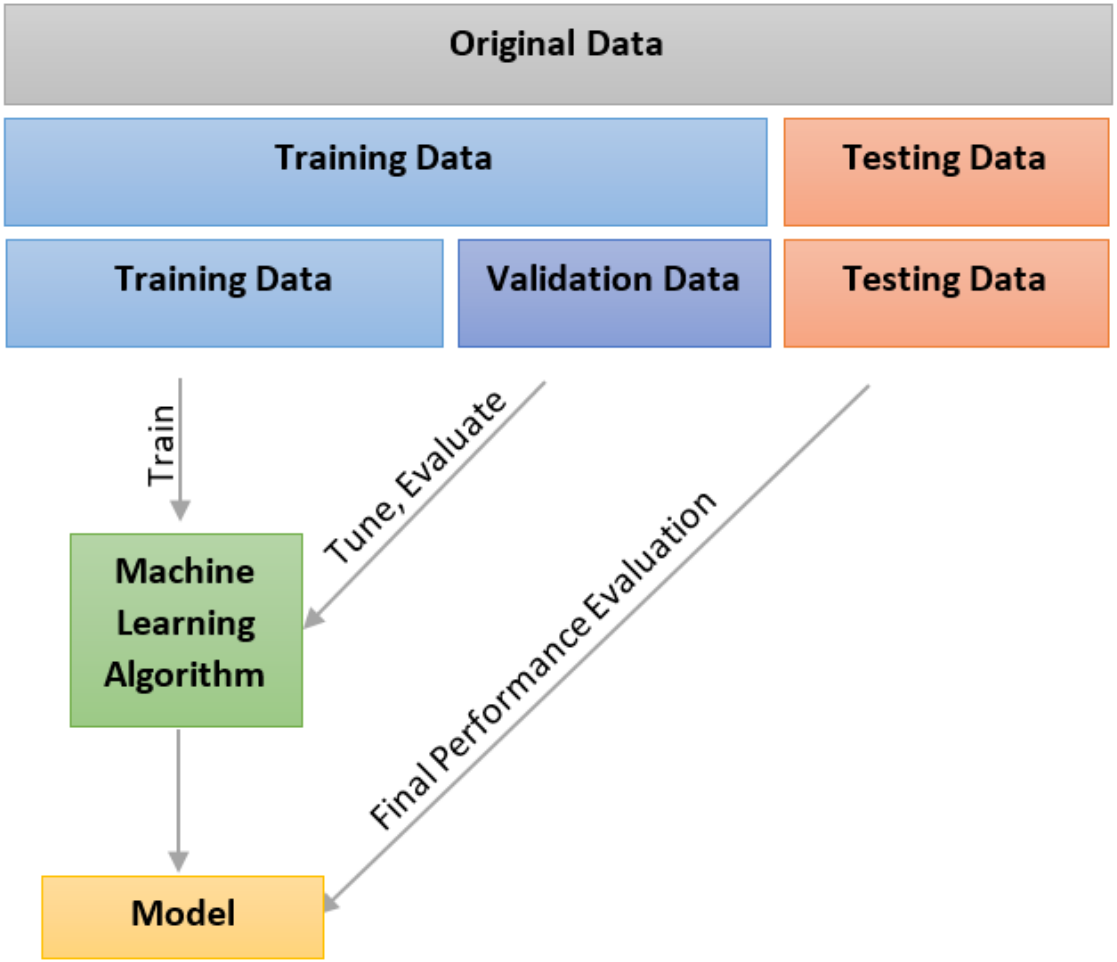
Validation data is used to test the generalization ability of the trained models and select the best one

Test data is used to final accuracy estimation of the model

$$\mathscr{D} = (\mathscr{D}_T \cup \mathscr{D}_V) \cup \mathscr{D}_{Tst}$$

$$\mathscr{D}_{Tr} = \mathscr{D}_T \cup \mathscr{D}_V$$

Training, Validation and Test Samples



Validation sample is considered to be a part of training process

Expectation of Quadratic Loss

Suppose that σ is an observation of random variable S at given x

$F(x)$ — response of the model F at given x (determined)
$S = f(x)$ — value of target function f at given x (random)

Expectation of quadratic loss function $L(F, (x, S))$ at given x:

$$\begin{aligned} \mathrm{M}[L(F,(x,S))|x] &= \mathrm{M}\left[(F(x)-S)^2|x\right] \\ &= F^2(x) - 2F(x)\mathrm{M}[S|x] + \mathrm{M}[S^2|x] \end{aligned}$$

Expectation of Quadratic Loss

Suppose that σ is an observation of random variable S at given x

$F(x)$ — response of the model F at given x (determined)
$S = f(x)$ — value of target function f at given x (random)

Expectation of quadratic loss function $L(F, (x, S))$ at given x:

$$\begin{aligned}\mathrm{M}[L(F,(x,S))|x] &= \mathrm{M}\left[(F(x)-S)^2|x\right] \\ &= F^2(x) - 2F(x)\mathrm{M}[S|x] + \mathrm{M}[S^2|x] \\ &= F^2(x) - 2F(x)\mathrm{M}[S|x] + \mathrm{D}[S|x] + (\mathrm{M}[S|x])^2\end{aligned}$$

Expectation of Quadratic Loss

Suppose that σ is an observation of random variable S at given x

$F(x)$ — response of the model F at given x (determined)
$S = f(x)$ — value of target function f at given x (random)

Expectation of quadratic loss function $L(F, (x, S))$ at given x:

$$\begin{aligned}\mathrm{M}[L(F,(x,S))|x] &= \mathrm{M}\left[(F(x)-S)^2|x\right] \\ &= F^2(x) - 2F(x)\mathrm{M}[S|x] + \mathrm{M}[S^2|x] \\ &= F^2(x) - 2F(x)\mathrm{M}[S|x] + \mathrm{D}[S|x] + (\mathrm{M}[S|x])^2 \\ &= (F(x)-\mathrm{M}[S|x])^2 + \sigma_x^2\end{aligned}$$

$(F(x) - \mathrm{M}[S|x])^2$ — error of model F at given x

$\sigma_x^2 = \mathrm{D}[S|x]$ — noise, doesn't depend on $\mathscr{D}$ or F

$F(x) = \mathrm{M}[S|x] \Leftrightarrow F(x)$ is a regression function S on x

Bias-Variance Decomposition

Expectation of quadratic loss at given x:

$$\mathrm{M}[L(F,(x,S))|x] = (F(x) - \mathrm{M}[S|x])^2 + \sigma_x^2$$

The neural network F trained on training data $\mathscr{D}_T$ depends on $\mathscr{D}_T$: $F(x, \mathscr{D}_T)$ — response of the neural network F trained on random sample $\mathscr{D}_T$ at given x

Expectation over all random samples $\mathscr{D}_T$:

$$\mathrm{M}\left[(F(x,\mathscr{D}_T) - \mathrm{M}[S|x])^2\right]$$
$$= \mathrm{M}\left[F(x,\mathscr{D}_T)^2\right] - 2\mathrm{M}[F(x,\mathscr{D}_T)]\mathrm{M}[S|x] + \mathrm{M}[S|x]^2$$

Bias-Variance Decomposition

Expectation of quadratic loss at given x:

$$\mathrm{M}[L(F,(x,S))|x] = (F(x) - \mathrm{M}[S|x])^2 + \sigma_x^2$$

The neural network F trained on training data $\mathscr{D}_T$ depends on $\mathscr{D}_T$: $F(x, \mathscr{D}_T)$ — response of the neural network F trained on random sample $\mathscr{D}_T$ at given x

Expectation over all random samples $\mathscr{D}_T$:

$$\begin{aligned}
&\mathrm{M}\left[(F(x,\mathscr{D}_T) - \mathrm{M}[S|x])^2\right] \\
&= \mathrm{M}\left[F(x,\mathscr{D}_T)^2\right] - 2\mathrm{M}[F(x,\mathscr{D}_T)]\mathrm{M}[S|x] + \mathrm{M}[S|x]^2 \\
&= \mathrm{D}\left[F(x,\mathscr{D}_T)\right] + \mathrm{M}[F(x,\mathscr{D}_T)]^2 - 2\mathrm{M}[F(x,\mathscr{D}_T)]\mathrm{M}[S|x] + \mathrm{M}[S|x]^2
\end{aligned}$$

Bias-Variance Decomposition

Expectation of quadratic loss at given x:

$$\mathrm{M}[L(F,(x,S))|x] = (F(x) - \mathrm{M}[S|x])^2 + \sigma_x^2$$

The neural network F trained on training data $\mathscr{D}_T$ depends on $\mathscr{D}_T$: $F(x, \mathscr{D}_T)$ — response of the neural network F trained on random sample $\mathscr{D}_T$ at given x

Expectation over all random samples $\mathscr{D}_T$:

$$\begin{aligned}
&\mathrm{M}\left[(F(x,\mathscr{D}_T) - \mathrm{M}[S|x])^2\right] \\
&= \mathrm{M}\left[F(x,\mathscr{D}_T)^2\right] - 2\mathrm{M}[F(x,\mathscr{D}_T)]\mathrm{M}[S|x] + \mathrm{M}[S|x]^2 \\
&= \mathrm{D}\left[F(x,\mathscr{D}_T)\right] + \mathrm{M}[F(x,\mathscr{D}_T)]^2 - 2\mathrm{M}[F(x,\mathscr{D}_T)]\mathrm{M}[S|x] + \mathrm{M}[S|x]^2 \\
&= (\mathrm{M}[F(x,\mathscr{D}_T)] - \mathrm{M}[S|x])^2 + \mathrm{D}\left[F(x,\mathscr{D}_T)\right]
\end{aligned}$$

$(\mathrm{M}[F(x,\mathscr{D}_T)] - \mathrm{M}[S|x])$ — statistical bias of model F at given x
$\mathrm{D}\left[F(x,\mathscr{D}_T)\right]$ — variance of model F over training samples $\mathscr{D}_T$

Bias-Variance Trade-off

Expectation of loss L at given x over training samples $\mathscr{D}_T$:

$$\begin{aligned}\mathrm{M}[L(F,(x,S))|x] &= (\mathrm{M}[F(x,\mathscr{D}_T)] - \mathrm{M}[S|x])^2 + \mathrm{D}\left[F(x,\mathscr{D}_T)\right] + \sigma_x^2 \\ &= Bias^2[F(x)] + Var[F(x)] + \sigma_x^2\end{aligned}$$

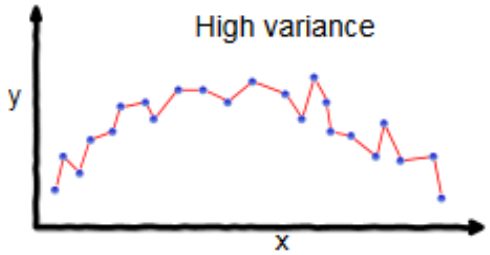
Three sources of error:

- $Bias^2[F]$ — error due to incorrect neural network architecture or its complexity
- $Var[F]$ — error due to variance of training samples (inability to perfectly estimate neural network's parameters from limited and noisy data)
- σ_x^2 — unavoidable error (doesn't depend on neural network)

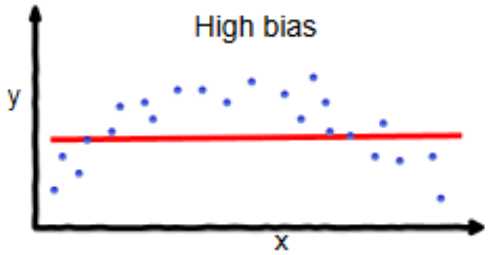
The architecture and complexity of neural network determines trade-off between bias and variance

Bias-Variance Trade-off. Illustration 1

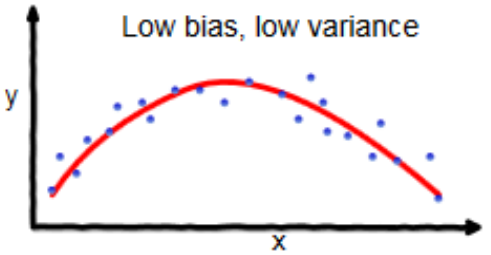
Overfitting



Underfitting



Good balance



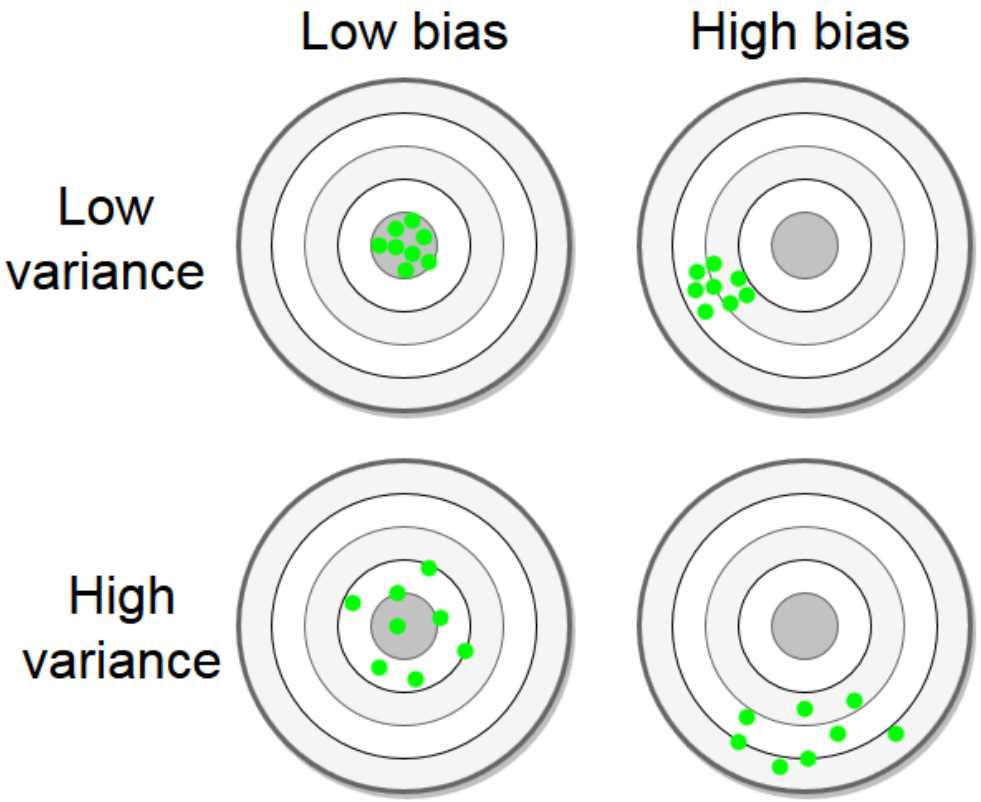
High variance leads to overfitting
High bias leads to underfitting

Low neural network complexity $\Rightarrow$ high bias, low variance
High neural network complexity $\Rightarrow$ low or high bias, high variance

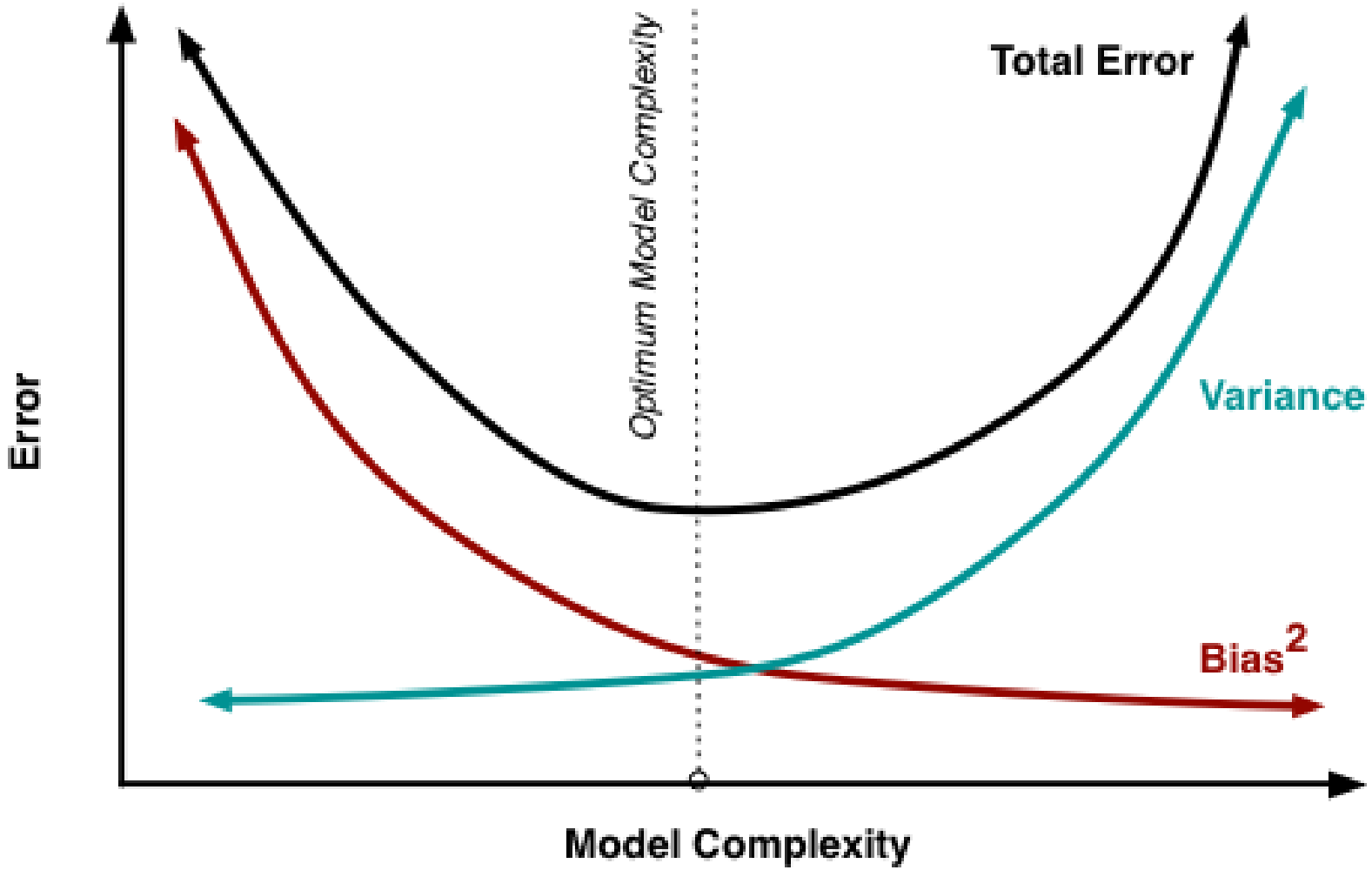
Appropriate neural network complexity leads to low bias, low variance

Bias-Variance Trade-off. Illustration 2

Dartboard = space of models
Bullseye = target function
Darts = learned models

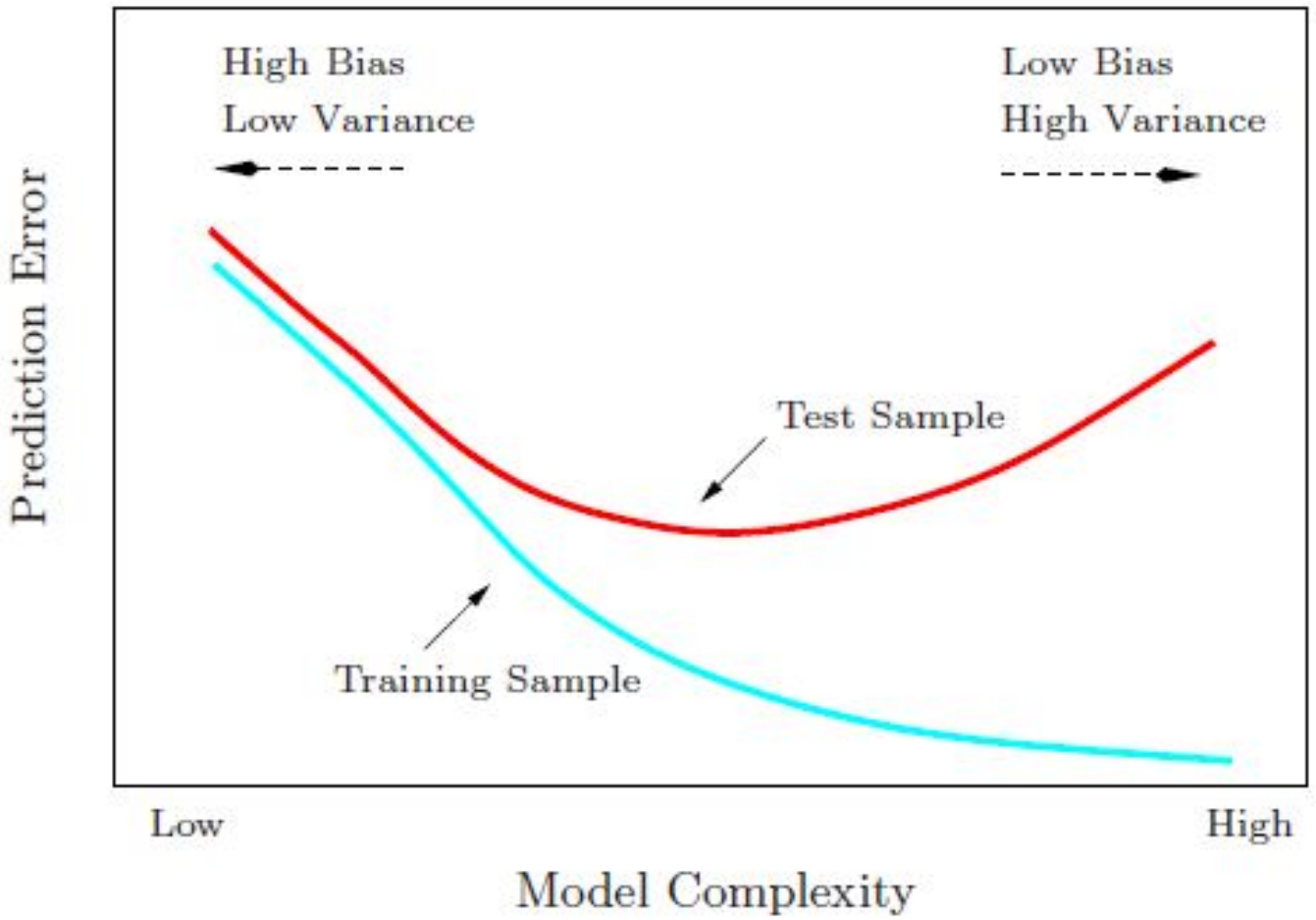


Bias-Variance Trade-off. Illustration 3



Generalization and Model Complexity

Generalization of model depends on its complexity



Estimation of Model Error

$\mathscr{D} = (\mathscr{D}_T \cup \mathscr{D}_V) \cup \mathscr{D}_{Tst}$ — available data
F — model trained on training sample $\mathscr{D}_T$

How to estimate the error of model F on unseen data?

$E(F)$ — true error of model F on unseen data

$E^*_T(F), E^*_V(F), E^*_{Tst}(F)$ — empirical errors (e.g. MSE) over train, validation and test samples

$E^*_T(F)$ — this estimate is optimistic (i.e. biased)
$E^*_V(F)$ — validation sample $\mathscr{D}_V$ was used in training process
$E^*_{Tst}(F)$ — estimation of model error over unseen examples

$E^*_{Tst}(F)$ looks good estimation but...

Estimation of Model Error

$\mathscr{D} = (\mathscr{D}_T \cup \mathscr{D}_V) \cup \mathscr{D}_{Tst}$ — available data
F — model trained on training sample $\mathscr{D}_T$

How to estimate the error of model F on unseen data?

$E(F)$ — true error of model F on unseen data

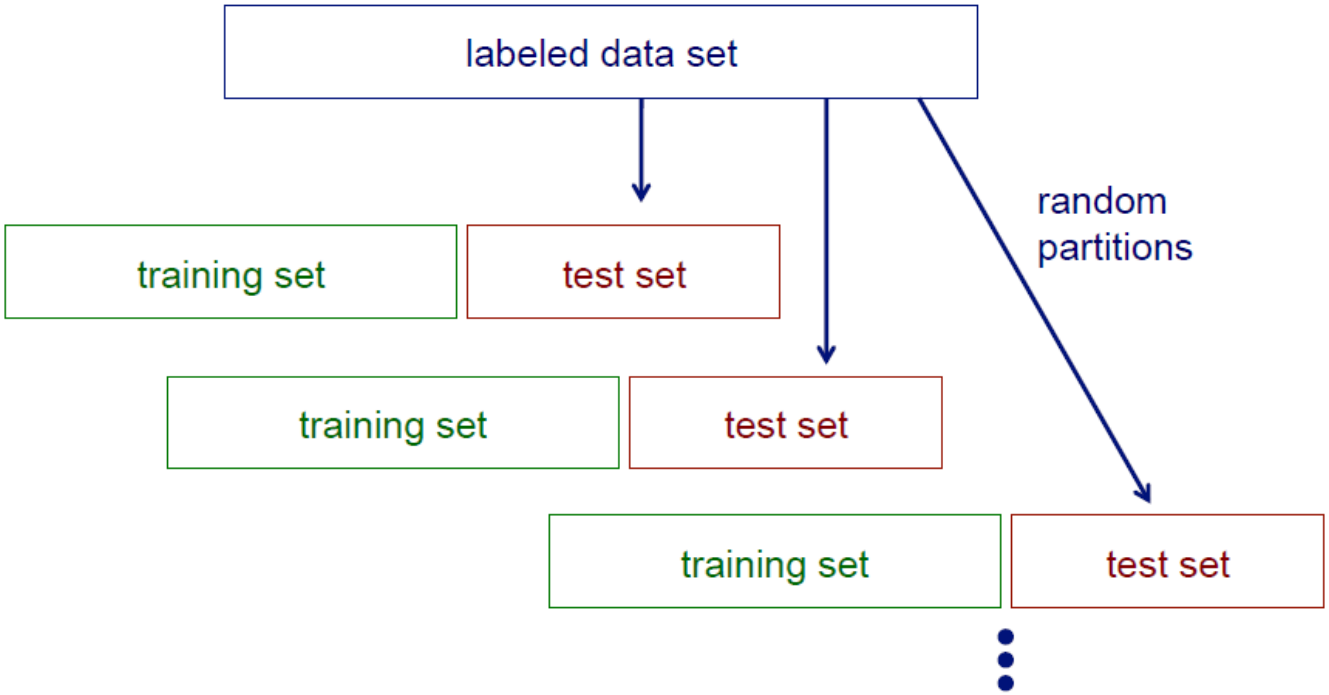
$E^*_T(F), E^*_V(F), E^*_{Tst}(F)$ — empirical errors (e.g. MSE) over train, validation and test samples

$E^*_T(F)$ — this estimate is optimistic (i.e. biased)
$E^*_V(F)$ — validation sample $\mathscr{D}_V$ was used in training process
$E^*_{Tst}(F)$ — estimation of model error over unseen examples

$E^*_{Tst}(F)$ looks good estimation but... a single training and test sets used don't tell us how sensitive error is to particular training and test samples

Partitioning the Data

Solution: repeatedly partitioning the available data $\mathscr{D}$ into training and test sets



F_i — neural network trained on training data from i-th partition

$E^*_{Tst}(F_1), ..., E^*_{Tst}(F_k)$ — estimations of error for models $F_1, ..., F_k$

Cross-Validation Techniques

Definition

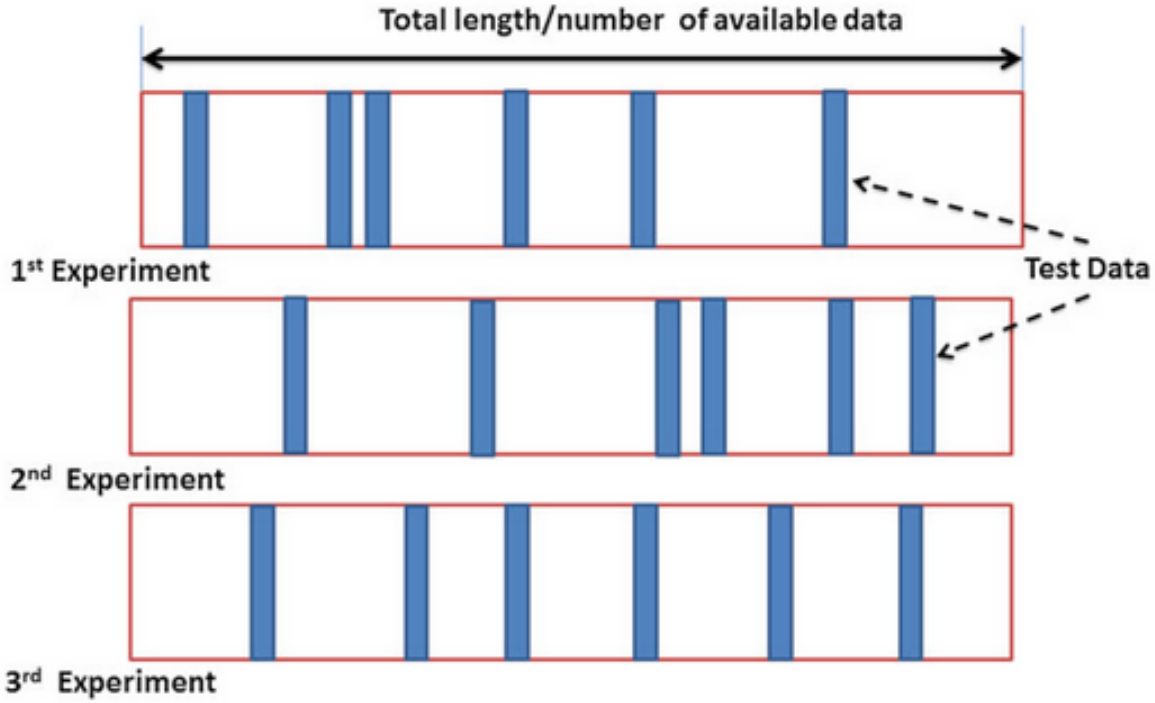
Cross-validation (CV) is a model evaluation technique used to assess a machine learning algorithm's performance in making predictions on new datasets that it has not been trained on

Cross validation techniques:

- Repeated random sub-sampling (Monte-Carlo CV)
- k-fold
- Holdout
- Leave-one-out (LOOCV)
- Resubstitution

Resubstitution does not partition the data, uses the training data for validation

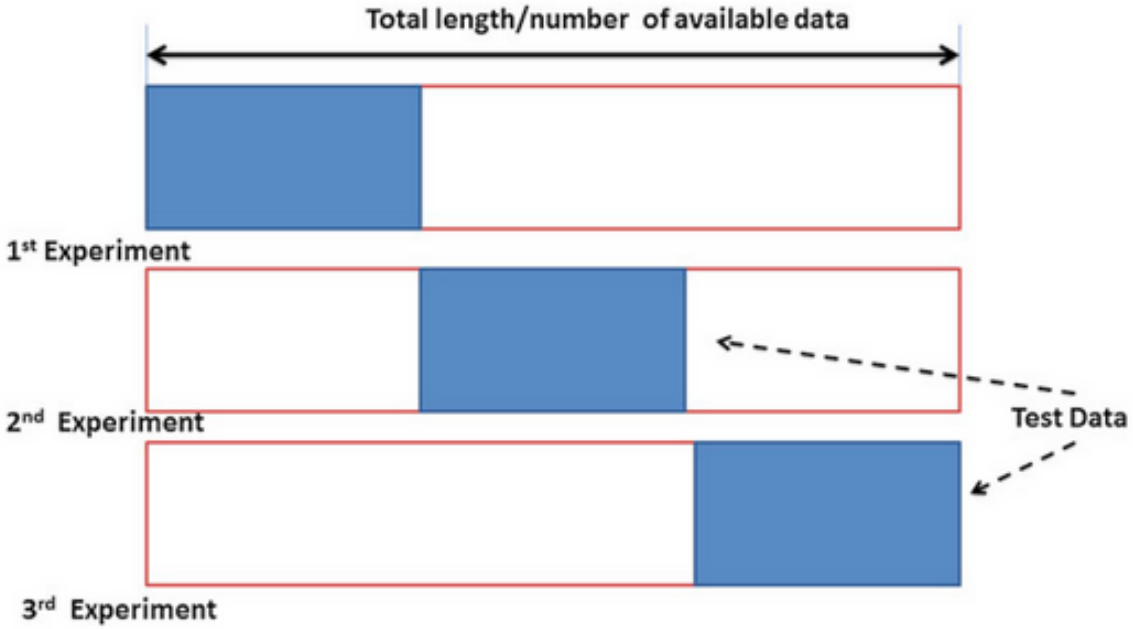
Repeated Random Sub-sampling CV



Whole data is randomly partitioned into training and test subsamples k times in specified proportion

Sample of errors: $E^*_{Tst}(F_1), ..., E^*_{Tst}(F_k)$

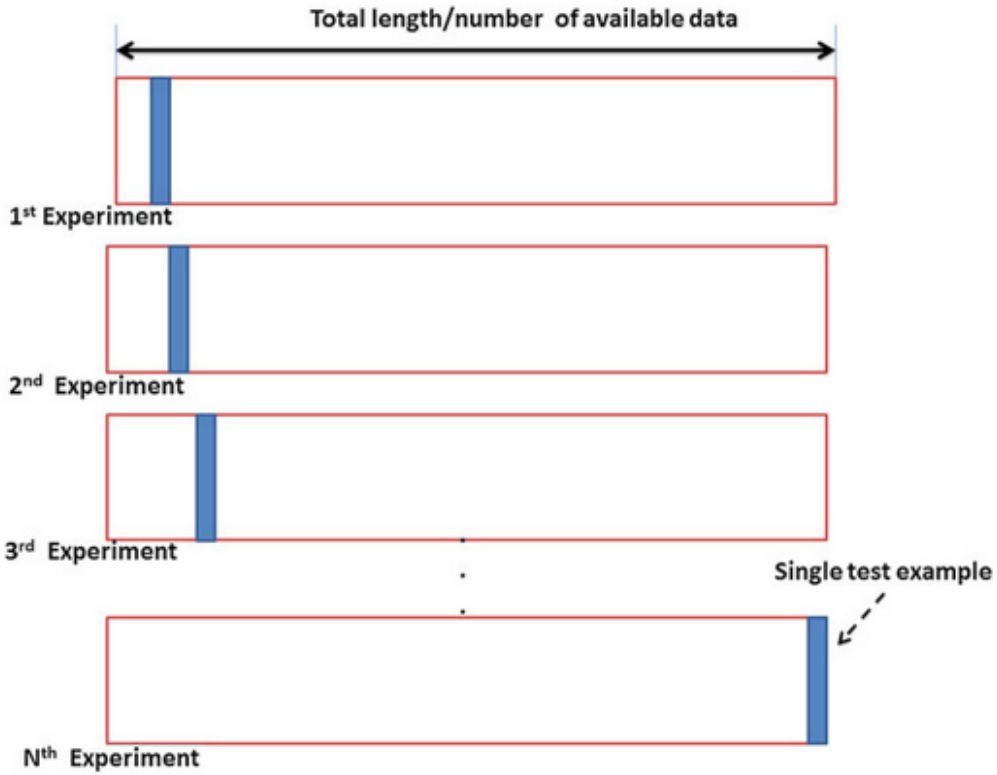
k-fold CV



Whole data is randomly partitioned into k equal sized subsamples (folds). One of k folds is retained as the test data, and the remaining $k-1$ folds are used as training data

Sample of errors: $E^*_{Tst}(F_1), ..., E^*_{Tst}(F_k)$

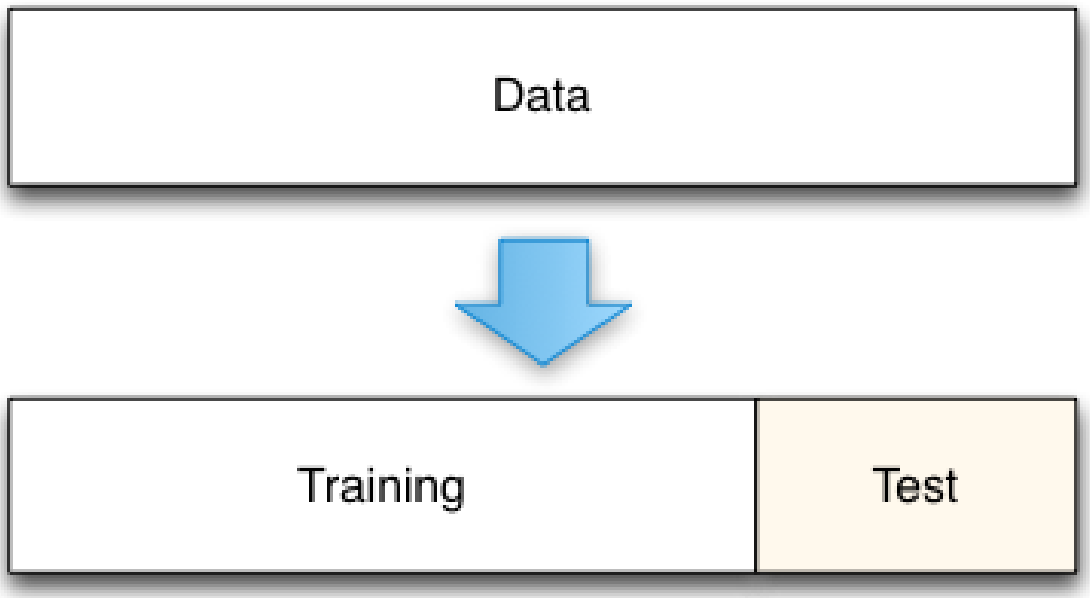
Leave-one-out CV



LOOCV is particular case of k-fold CV when $k = n$

Sample of errors: $E^*_{Tst}(F_1), ..., E^*_{Tst}(F_n)$

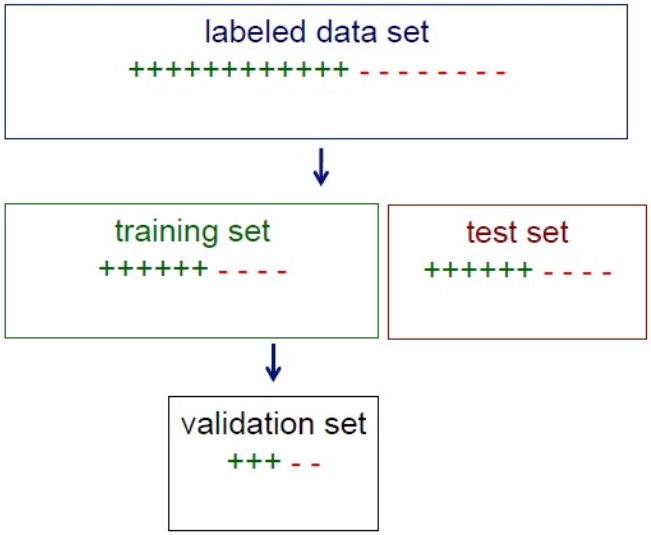
Holdout CV



Whole data is randomly partitioned into two sets: training and test subsamples in specified proportion

Sample of errors: $E^*_{Tst}(F)$

Stratified sampling



The test subsets (folds) are selected so that the mean response value is approximately equal in all the folds

In the case of a classification, stratified cross-validation keep the distribution of class labels in each fold

In practice: first stratify instances by class, then randomly select instances from each class proportionally

True error estimation

Whenever we use multiple training sets, as in k-fold CV and random sub-sampling CV, we are evaluating neural network architecture and learning algorithm, no individual learned neural network F

The true error E_{Tst} is the error when tested on the entire population of data instances

Sample of errors: $E^*_{Tst}(F_1), ..., E^*_{Tst}(F_k)$

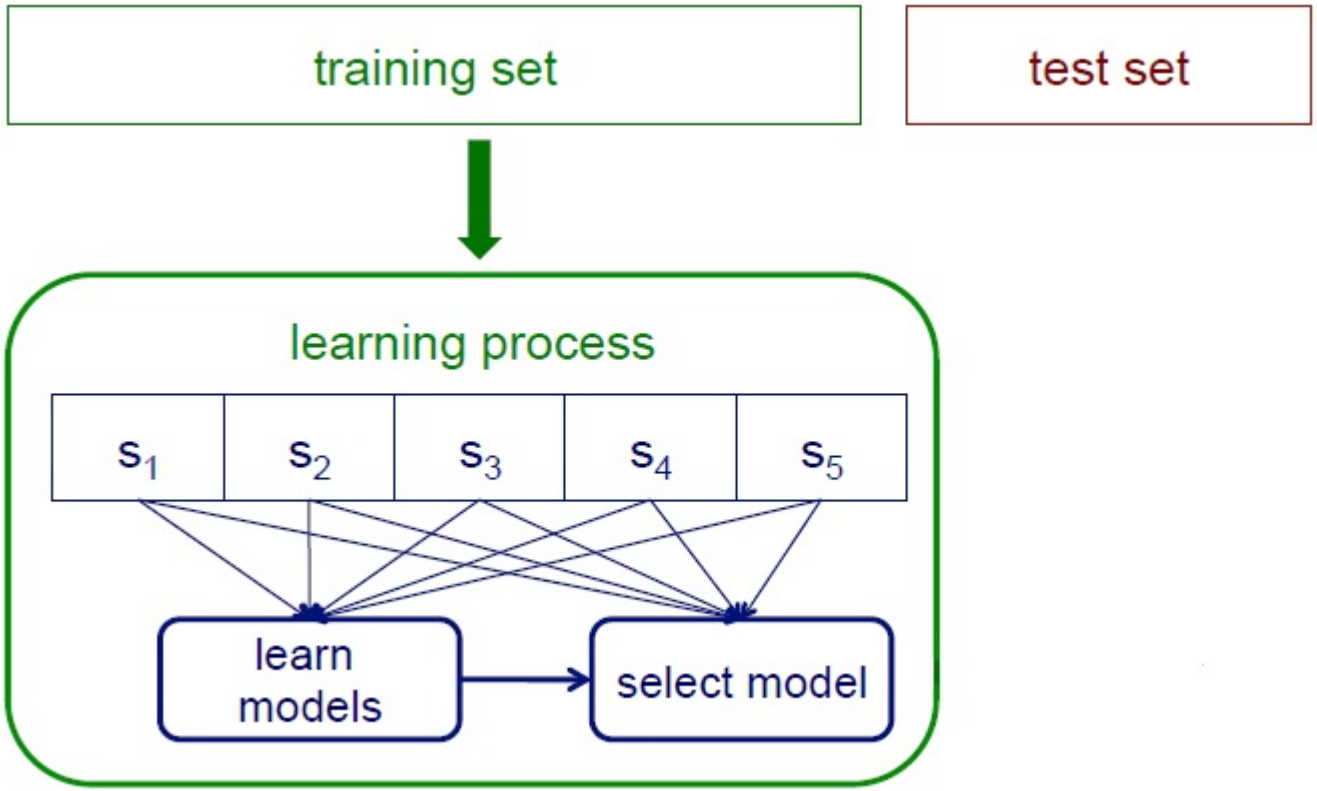
Point estimator: $\overline{E}_{Tst} = \frac{1}{k}\sum_{i=1}^{k} E^*_{Tst}(F_i)$

Variance: $s^2[E_{Tst}] = \frac{1}{k}\sum_{i=1}^{k}(E^*_{Tst}(F_i) - \overline{E}_{Tst})^2$

The cross-validation estimator $\overline{E}_{Tst}$ is very nearly unbiased for E_{Tst}
The variance $s^2[E_{Tst}]$ can be reduced by increasing the size of test set

Internal Cross-Validation

Instead of a single validation set, we can use cross-validation within a training set (e.g. to find meta-parameters and select a model)



Improving Generalization

Too low complexity $\Rightarrow$ the network is unable to fit training data

The larger network you use, the more complex the functions the network can create

Too high complexity $\Rightarrow$ the network tends to memorize the training examples, but it has not learned to generalize to new data

It is difficult to know beforehand how large a network should be for a specific application

Methods for improving generalization:

- Reduce complexity of the neural network
- Increase amount of the training data
- Regularization

Regularization Techniques

Definition

Regularization is a process of introducing additional information into the learning process, reducing the possible space of appropriate solutions and preventing overfitting

Regularization techniques:

- L_2 weight regularization
- L_1 weight regularization
- Early stopping
- Data augmentation
- Noise injection
- Dropout
- Batch normalization

L_2 Weight Regularization

The idea of L_2 weight regularization is to add L_2 regularization term to the objective:

$$E(w) = E_0(w) + \frac{\lambda}{2m}||w||_2^2 = E_0(w) + \frac{\lambda}{2m}\sum_{i=1}^{m} w_i^2 \to \min_w$$

where $E_0(0)$ is mean loss function over training sample,
$\lambda > 0$ is a regularization parameter

Other implementation:

$$E(w) = (1-\lambda)E_0(w) + \frac{\lambda}{2m}||w||_2^2 \to \min_w$$

where $0 < \lambda < 1$ is a regularization parameter

Regularization parameter λ is additional hyper-parameter to the training process

Gradient of L_2-Regularized Objective

Gradient of regularized objective:

$$\nabla E(w) = \nabla E_0(w) + \frac{\lambda}{m} w$$

Gradient descent:

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau)) = w(\tau) - \alpha \left(\nabla E_0(w(\tau)) + \frac{\lambda}{m} w(\tau) \right)$$

$$= w(\tau) - \alpha \nabla E_0(w(\tau)) - \frac{\alpha\lambda}{m} w(\tau)$$

$$= \left(1 - \frac{\alpha\lambda}{m}\right) w(\tau) - \alpha \nabla E_0(w(\tau))$$

This is the same as the usual gradient descent learning rule, except we first rescale the weights $w(\tau)$ by a factor $\left(1 - \frac{\alpha\lambda}{m}\right)$

L_2 weight regularization is also called weight decay

L_1 Weight Regularization

The idea of L_1 weight regularization is to add L_1 regularization term to the objective:

$$E(w) = E_0(w) + \frac{\lambda}{m}||w||_1 = E_0(w) + \frac{\lambda}{m}\sum_{i=1}^{m}|w_i| \to \min_w$$

where $E_0(0)$ is mean loss function over training sample,
$\lambda > 0$ is a regularization parameter

Other implementation:

$$E(w) = (1-\lambda)E_0(w) + \frac{\lambda}{m}||w||_1 \to \min_w$$

where $0 < \lambda < 1$ is a regularization parameter

Regularization parameter λ is additional hyper-parameter to the training process

Gradient of L_1-Regularized Objective

Gradient of regularized objective:

$$\nabla E(w) = \nabla E_0(w) + \frac{\lambda}{m}\text{sgn}(w)$$

Gradient descent:

$$\begin{aligned} w(\tau+1) &= w(\tau) - \alpha\nabla E(w(\tau)) \\ &= w(\tau) - \alpha\left(\nabla E_0(w(\tau)) + \frac{\lambda}{m}\text{sgn}(w(\tau))\right) \\ &= w(\tau) - \alpha\nabla E_0(w(\tau)) - \frac{\alpha\lambda}{m}\text{sgn}(w(\tau)) \\ &= \left(w(\tau) - \frac{\alpha\lambda}{m}\text{sgn}(w(\tau))\right) - \alpha\nabla E_0(w(\tau)) \end{aligned}$$

This is the same as the usual gradient descent learning rule, except we first shrink the weights $w(\tau)$ by a value $\frac{\alpha\lambda}{m}$

L_2 vs L_1 Regularization

L_2 regularization:

$$w(\tau+1) = \left(1 - \frac{\alpha\lambda}{m}\right) w(\tau) - \alpha\nabla E_0(w(\tau))$$

L_1 regularization:

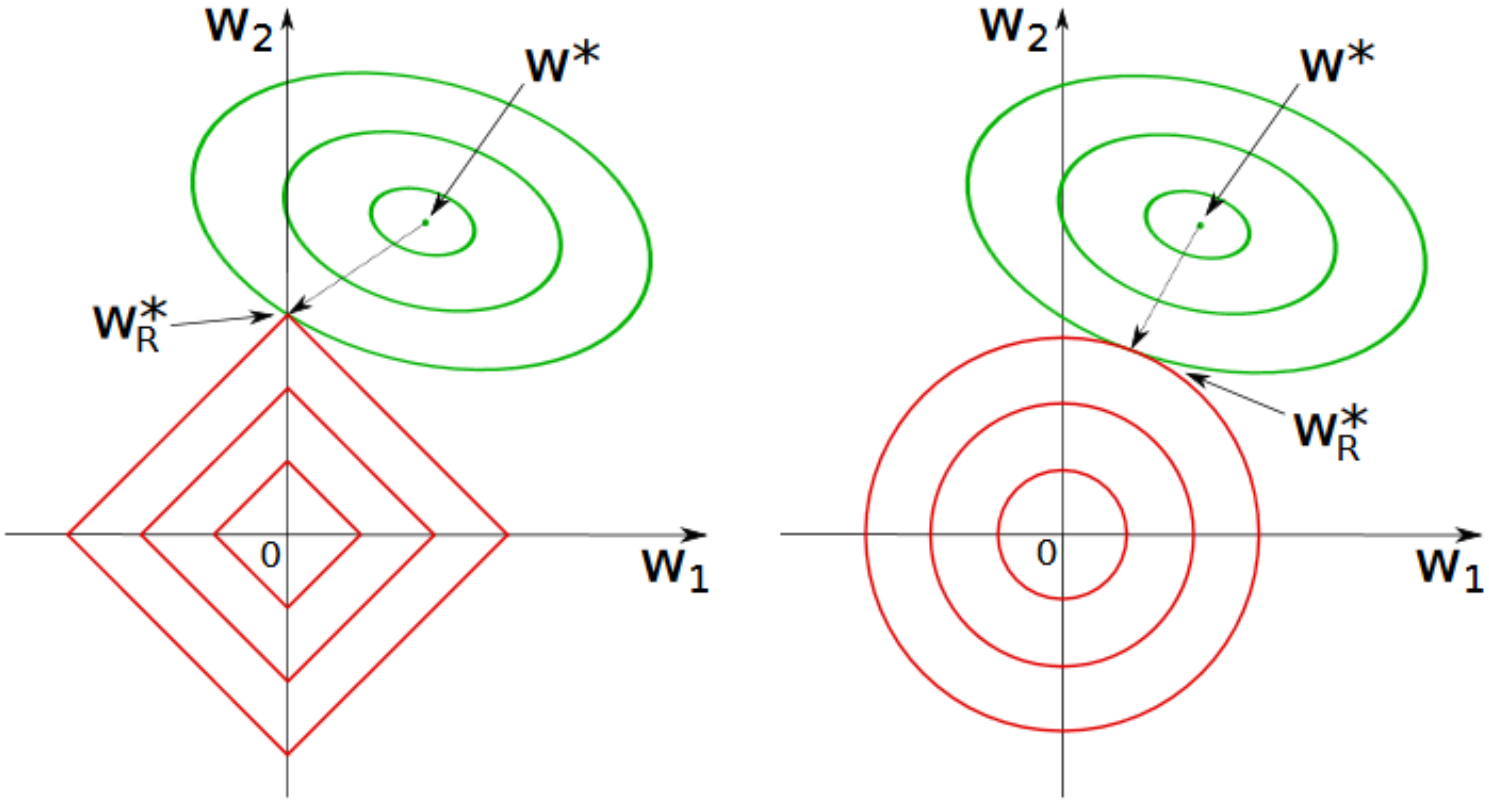
$$w(\tau+1) = \left(w(\tau) - \frac{\alpha\lambda}{m}\text{sgn}(w(\tau))\right) - \alpha\nabla E_0(w(\tau))$$

Both methods penalize large weights and shrink them toward zero but using different ways:

L_2 regularization shrinks the weights by an amount which is proportional to w
L_1 regularization shrinks the weights by a constant amount

w is large $\Rightarrow$ L_2 shrinks more than L_1
w is small $\Rightarrow$ L_1 shrinks more than L_2

L_2 vs L_1 Regularization. Illustration



(a) L_1 weight penalty function. (b) L_2 weight penalty function.

L_1 regularization tends to concentrate the neural network's parameters in a relatively small number of high-important weights, while the other weights are driven toward zero

Early Stopping

The idea of early stopping is in stopping training process before the objective reaches its minimum

Available data: $\mathscr{D} = (\mathscr{D}_T \cup \mathscr{D}_V) \cup \mathscr{D}_{Tst}$

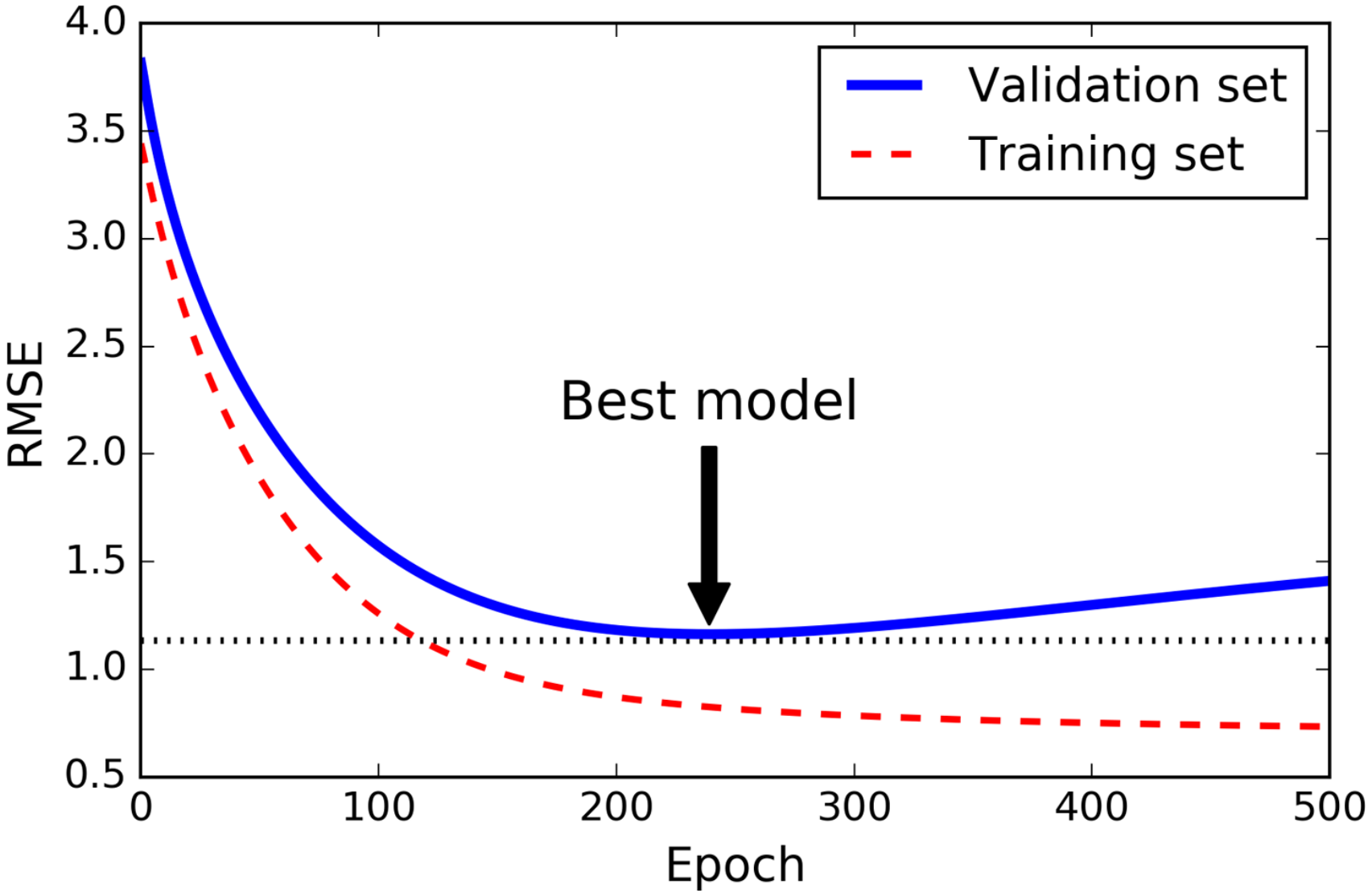
The error on the validation set $\mathscr{D}_V$ should be monitored during the training process

The validation error $E_V^*(F)$ normally decreases during the initial phase of training, as does the training set error $E_T^*(F)$

When the network begins to overfit the data, the error on the validation set begins to rise

Early stopping rule:
When the validation error $E_V^*(F)$ increases for a specified number of iterations the training process should be stopped, and the weights and biases are returned at the minimum of the validation error

Early Stopping. Illustration



Data Augmentation

The idea of data augmentation is in generation of new training examples using some transformation function $g(x, \vartheta)$ applied to the existing examples in the training set $\mathscr{D}_T$

The transformation function $g(x, \vartheta)$ depends on the problem and incorporates knowledge about it into the learning algorithm

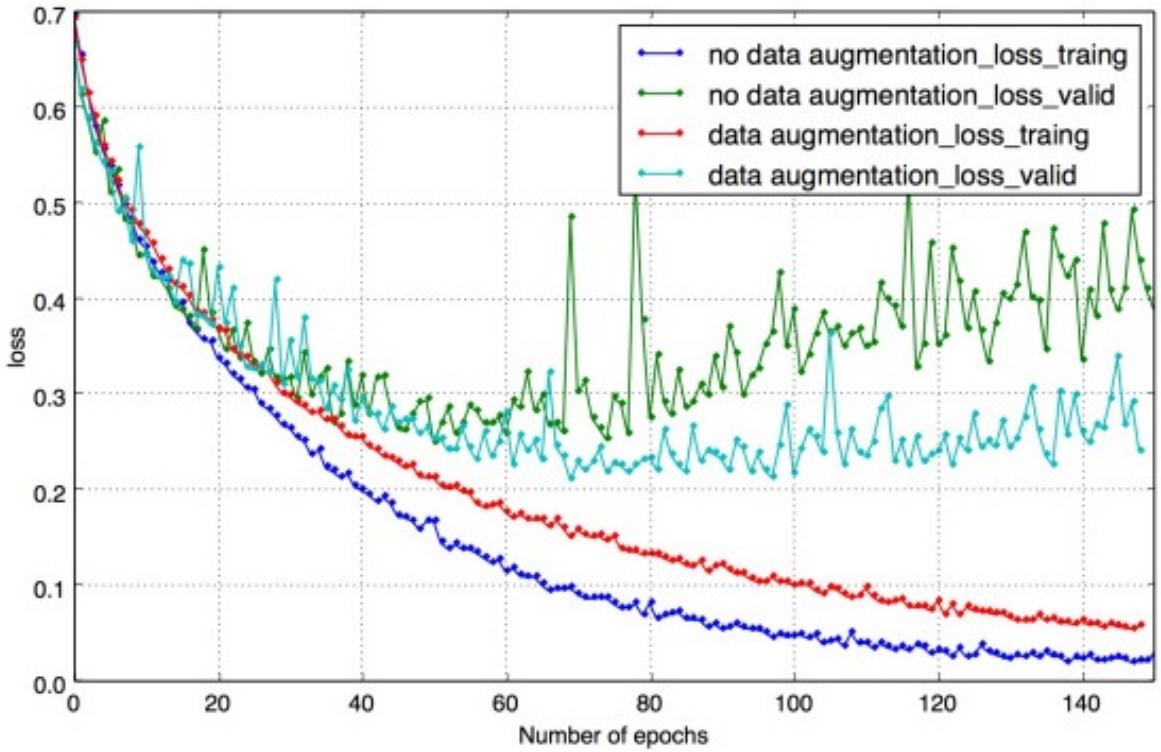
For image classification task: scaling, rotation, flipping, adding blur, etc.

Notes:

- Unlike other regularization methods, data augmentation does not reduce the model complexity
- The main drawback is increased learning time
- Augmented data can be generated on the fly to use it in stochastic gradient descent and to avoid storing all the additional data in memory

Data Augmentation. Illustration

Training curves of a network trained on Dogs-vs-Cats images dataset*



Augmented data: flipped and rotated images

*https://www.kaggle.com/c/dogs-vs-cats

Noise Injection

The idea of noise injection is in adding some random noise to inputs and/or gradients:

$$x_j^{(p)} := x_j^{(p)} + \varepsilon \quad \text{or} \quad \frac{\partial E(w(\tau))}{\partial w_i} := \frac{\partial E(w(\tau))}{\partial w_i} + \varepsilon$$

where $\varepsilon \sim N(0, \sigma)$, σ is the standard deviation of noise

Notes:

- Adding noise can make the neural network more robust to the variations in input vector
- Noise can be added on the fly before each iteration of training process
- Adding gradient noise makes networks more robust to poor initialization and gives the model more chances to escape the local minima

Dropout

The idea of dropout is in excluding randomly chosen neurons from the network at each training iteration

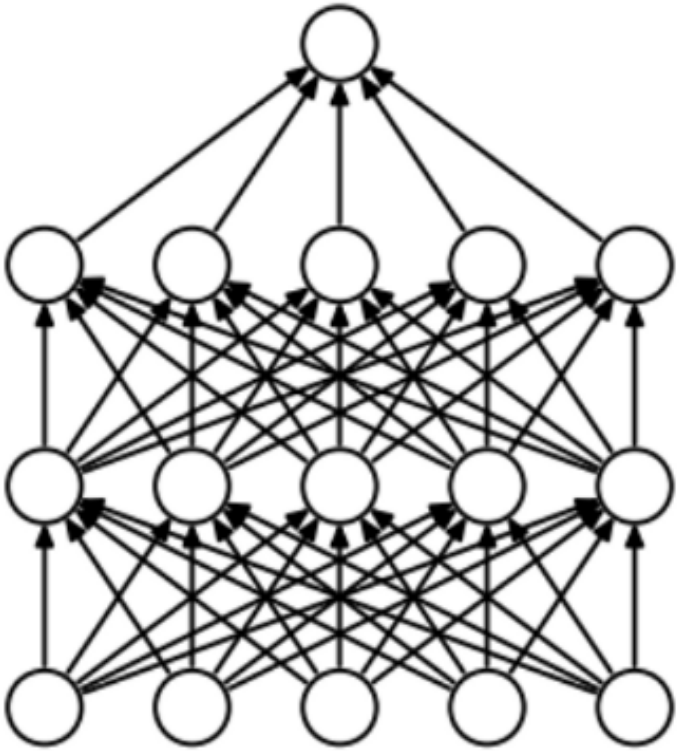
Dropout applied to l-th layer of neurons:

1. Generate a vector $v_l = (v_{l1}, ..., v_{lN_l})^T$, where each element is sampled from the Bernoulli distribution $B(1, p)$, $v_l \in \{0, 1\}^{N_l}$
2. On the forward pass multiply the values y_l by this vector (element-wise), $l = 1, ..., N_l$
3. On the backward pass multiply the gradients $\frac{\partial E}{\partial y_l}$ by this vector (element-wise), $l = 1, ..., N_l$

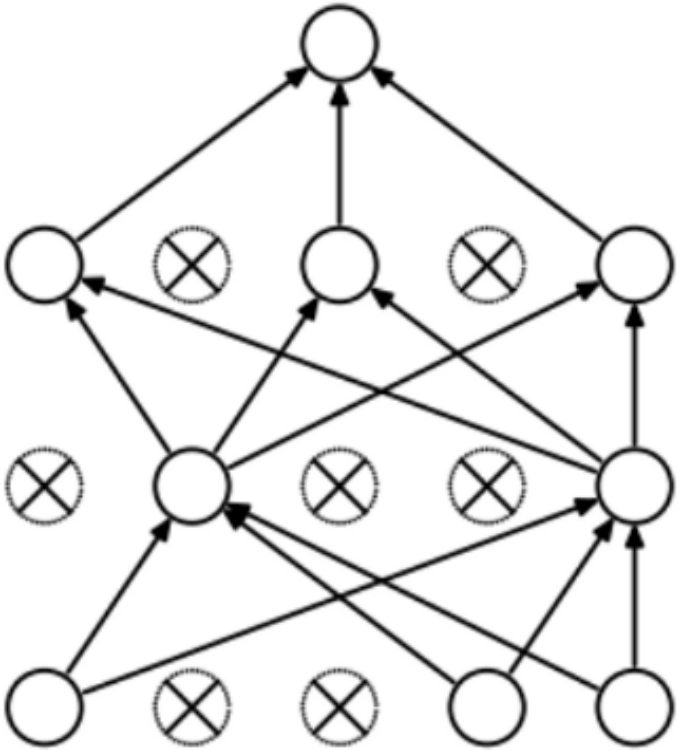
A new random subset of neurons to delete is chosen after every update of weights:

- for batch training – after every epoch
- for mini-batch training – after every mini-batch
- for stochastic training – after every training example

Dropout. Illustration



(a) Standard Neural Net



(b) After applying dropout.

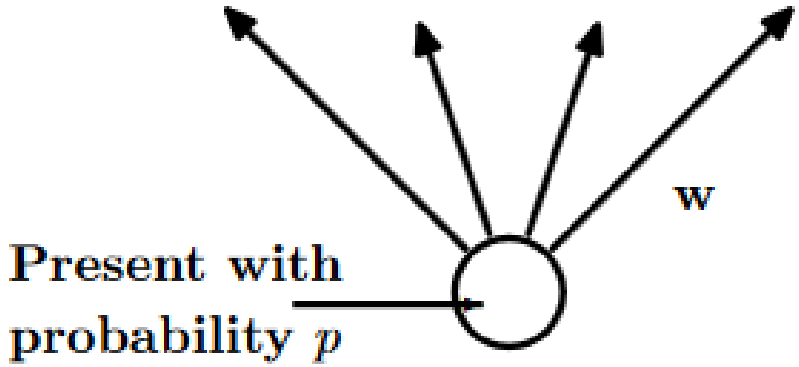
Droupout applied to l-th layer can be implemented as additional layer of neurons with non-trainable random binary weights v_l

Dropout at Test Phase

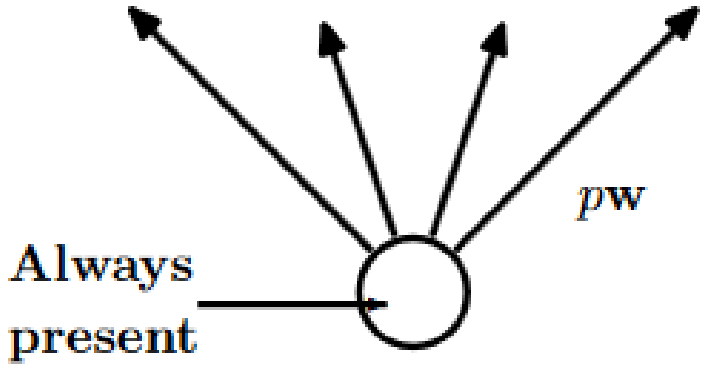
After training process the weights and biases of l-th layer will have been learnt under conditions in which only pN_l the hidden neurons were present (at average)

At test phase all neurons are present in the network

To compensate for that, the weights outgoing from the l-th layer should be multiplied by p



(a) At training time



(b) At test time

Why Does Dropout Work?

Training a neural network with dropout can be seen as training a collection of 2^N thinned networks with parameters sharing (N is the number of neurons under dropout), where each thinned network gets trained very rarely, or not at all

Most of the thinned models, in fact, will never be used. Those which are used will likely get only one training example, which make it an extreme form of bagging

This thinned models will overfit in different ways, and so, hopefully, the net effect of dropout (averaging) will be to reduce overfitting

The dropout reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons

Some Notes on Dropout

- Dropout parameter p, $0 < p < 1$, is additional hyper-parameter to the training process. The standard choice of the dropout probability is $p = 0.5$, so the method does not typically require hyper-parameter tuning
- Dropout is typically applied to the hidden layers but also can be applied to network's inputs (with higher p)
- For the network's inputs introducing noise instead of dropout might perform better
- Dropout increases the number of iterations required to training (for $p = 0.5$ roughly doubles). However, training time for each epoch is less
- Dropout is very simple to implement, and demonstrates a significant improvement on a large variety of tasks

Intuitive Explanation of Dropout

Imagine that you have a team of workers and the overall goal is to learn how to construct a building

When each of the workers is overly specialized, if one gets sick or makes a mistake, the whole building will be severely affected

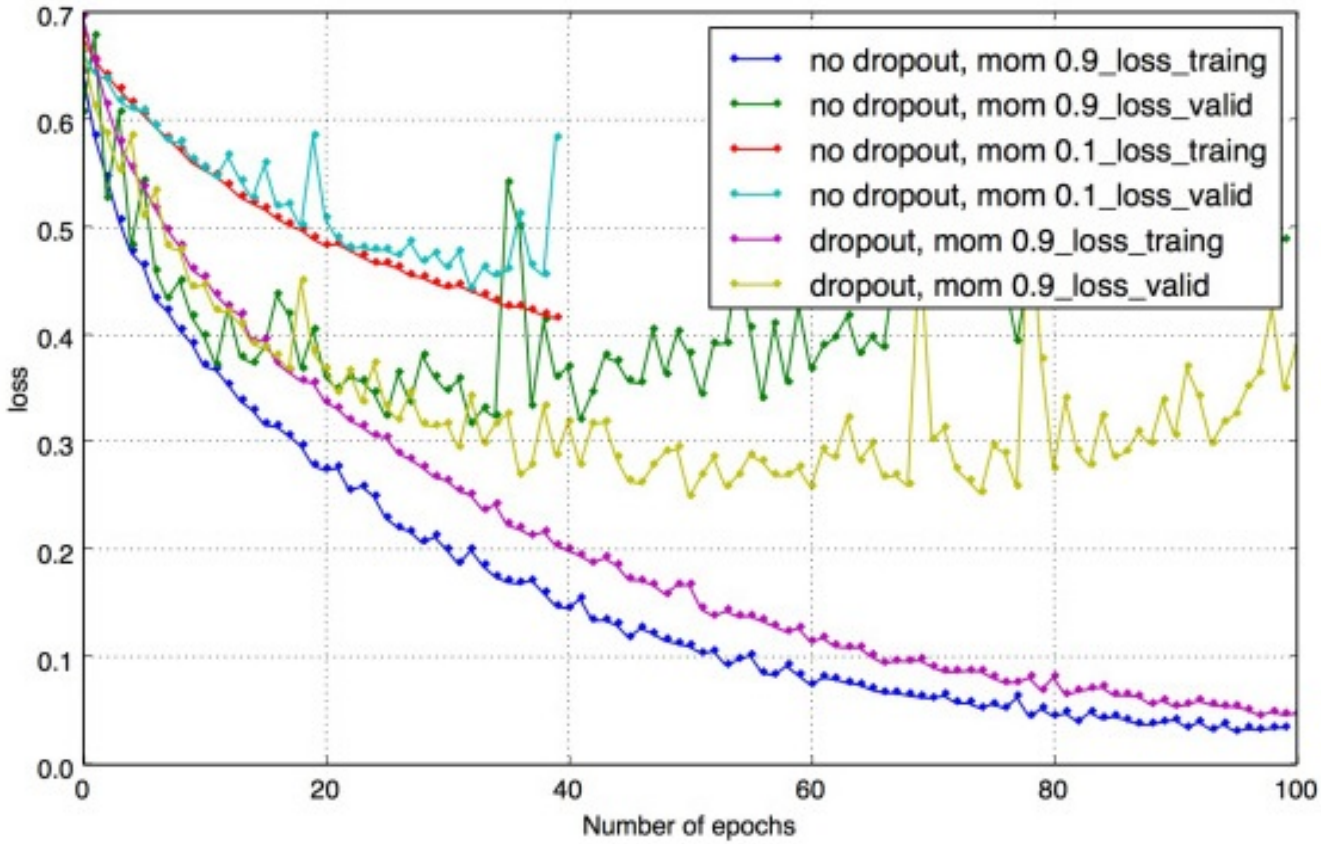
The solution proposed by dropout technique is to pick randomly every week some of the workers and send them to business trip

The hope is that the team overall still learns how to build the building and thus would be more resilient to noise or workers being on vacation

The dropout technique has been first proposed in a paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"by N.Srivastava, G.Hinton, A.Krizhevsky, I.Sutskever and R.Salakhutdinov in 2014 (Journal of Machine Learning Research, 15, Pp. 1929-1958)

Dropout. Illustration

Training curves of a network trained on Dogs-vs-Cats images dataset*



*https://www.kaggle.com/c/dogs-vs-cats

Internal Covariate Shift

The normalization of inputs (shifting inputs to zero-mean and unit variance) is often used as a pre-processing step to make the data comparable across features

It is known that network training converges faster if its inputs are whitened — i.e., linearly transformed to have zero means and unit variances, and decorrelated

As the data flows through layers of the neural network, their outputs can become too big or too small again

This effect is called internal covariate shift

Denormalized inputs of hidden layers make the training hard due to possible saturation of non-linearities in activation functions

A way to reduce internal covariate shift is to normalize inputs of hidden layers

Batch Normalization

The idea of batch normalization is to normalize the outputs of each layer to have zero mean and unit variance over each training batch:

$$\hat{y}_i^{(p)} = \gamma_i \frac{y_i^{(p)} - \bar{y}_i}{\sqrt{s_i^2 + \varepsilon}} + \beta_i, \quad i = 1, ..., N_l$$

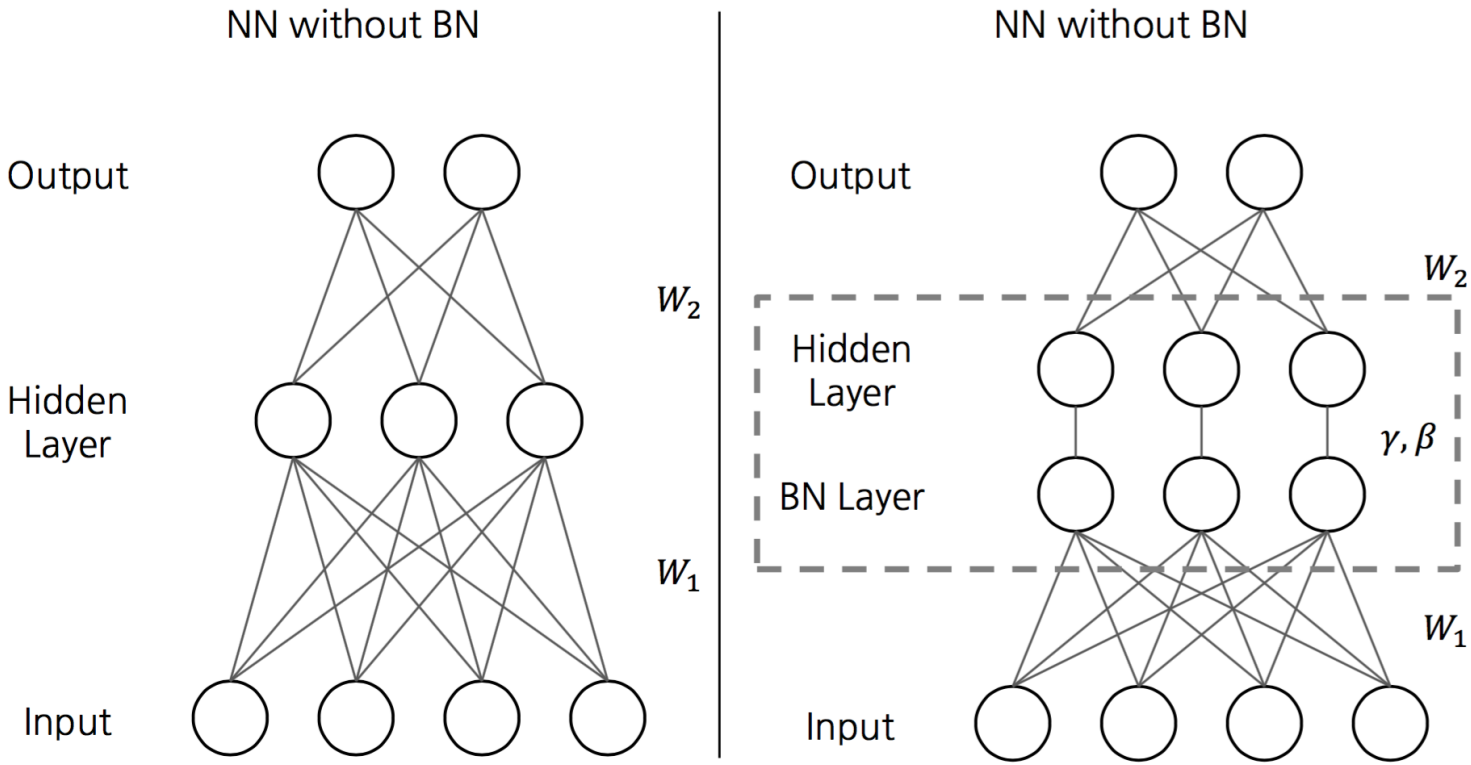
where $\bar{y}_i$ and s_i^2 are mean and standard deviation of i-th neuron's output y_i over current training mini-batch,
ε — smoothing term that avoids division by zero (usually $\varepsilon \simeq 10^{-8}$),
γ_i and β_i — adjustable batch normalization parameters

Why we need adjustable parameters in batch normalization
They are introduced in order to guarantee that the normalization procedure transforms the values only when it is necessary

Indeed, if $\gamma_i = \sqrt{s_i^2 + \varepsilon}$ and $\beta_i = \bar{y}_i$ the batch normalization performs identity transform

Batch Normalization Layer

Batch normalization can be viewed as an additional layer in neural network



The derivatives of training objective $E(w)$ w.r.t. parameters γ and β can be obtained using the backpropagation algorithm

Derivatives of Batch Normalization Layer

Model of batch normalization neuron:

$$\hat{y}_i^{(p)} = \gamma_i \frac{y_i^{(p)} - \bar{y}_i}{\sqrt{s_i^2 + \varepsilon}} + \beta_i$$

Partial derivatives of training objective $E(w)$ w.r.t. parameters γ_i and β_i:

$$\frac{\partial E^{(p)}(w)}{\partial \gamma_i} = \frac{\partial E^{(p)}(w)}{\partial \hat{y}_i^{(p)}} \frac{\partial \hat{y}_i^{(p)}}{\partial \gamma_i} = \frac{\partial E^{(p)}(w)}{\partial \hat{y}_i^{(p)}} \frac{y_i^{(p)} - \bar{y}_i}{\sqrt{s_i^2 + \varepsilon}}$$

$$\frac{\partial E^{(p)}(w)}{\partial \beta_i} = \frac{\partial E^{(p)}(w)}{\partial \hat{y}_i^{(p)}} \frac{\partial \hat{y}_i^{(p)}}{\partial \beta_i} = \frac{\partial E^{(p)}(w)}{\partial \hat{y}_i^{(p)}}$$

where $\frac{\partial E^{(p)}(w)}{\partial \hat{y}_i^{(p)}}$ can be obtained using the backpropagation algorithm through higher layers

Backpropagation through Batch Normalization Layer

Partial derivatives of objective $E(w)$ w.r.t. inputs $y_i^{(p)}$:

$$\frac{\partial E^{(p)}(w)}{\partial y_i^{(p)}} = \frac{\partial E^{(p)}(w)}{\partial \hat{y}_i^{(p)}} \frac{\partial \hat{y}_i^{(p)}}{\partial y_i^{(p)}} = \frac{\partial E^{(p)}(w)}{\partial \hat{y}_i^{(p)}} \frac{\gamma_i}{\sqrt{s_i^2 + \varepsilon}}$$

The partial derivatives $\frac{\partial E^{(p)}(w)}{\partial y_i^{(p)}}$ are to be used in backpropagation through lower layers

After each training step (updating the neural network's parameters) the means $\bar{y}_i$ and standard deviations s_i should be updated

The outputs $\hat{y}_i^{(p)}$ of batch normalization layer depends both on the neural network's input $x^{(p)}$ and all other examples in current mini-batch

Batch Normalization Layer after Training

After training process:

$$\hat{y}_i = \gamma_i \frac{y_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} + \beta_i$$

where μ_i and σ_i are mean and standard deviation estimated on the entire training sample, β_i and γ_i are trained parameters

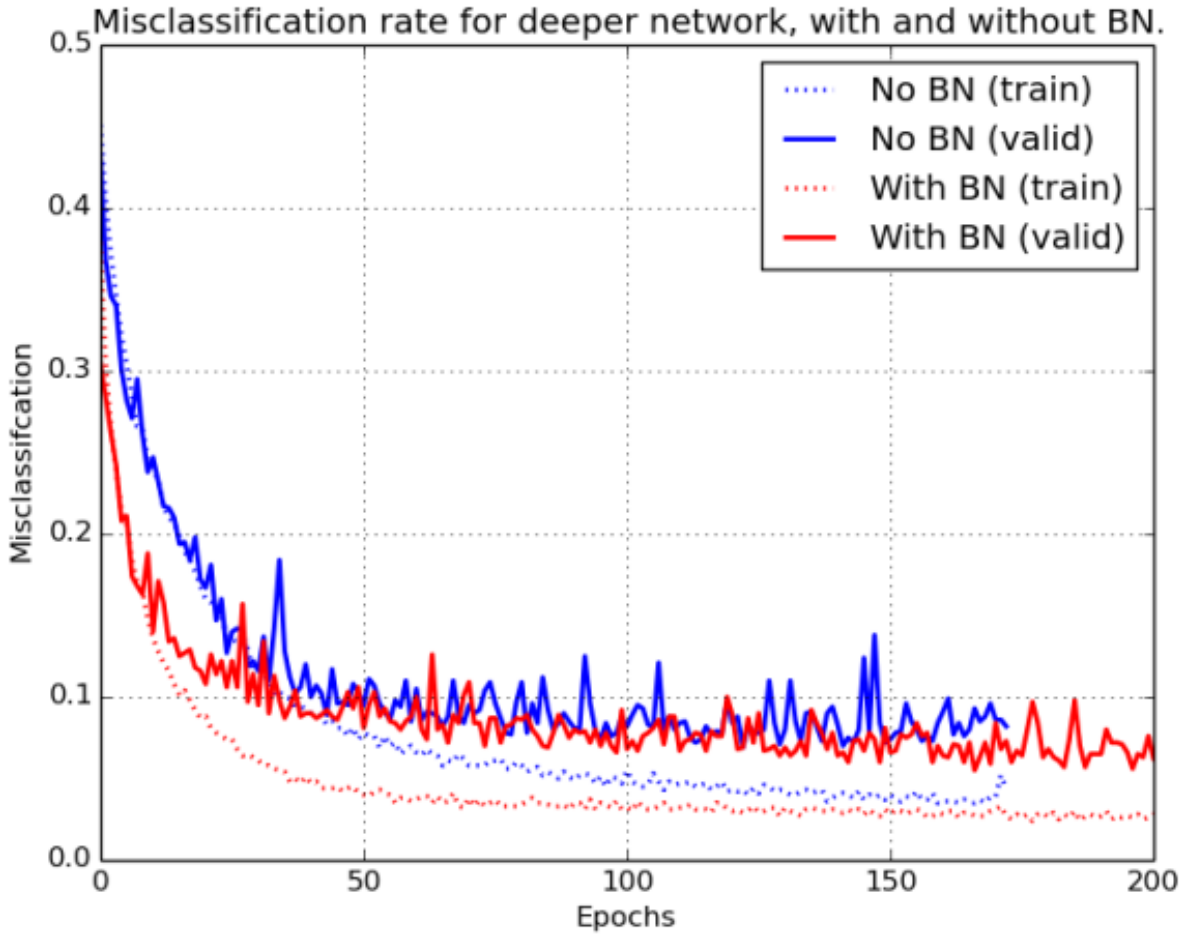
The batch normalization layer can be viewed as a layer of linear neurons:

$$\hat{y}_i = \frac{\gamma_i}{\sqrt{\sigma_i^2 + \varepsilon}} y_i + \left(\beta_i - \frac{\gamma_i \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} \right)$$

Batch normalization increases the generalization of the neural network and enables higher learning rates in training

Batch Normalization. Illustration

Training curves of a network trained on Dogs-vs-Cats images dataset*



*https://www.kaggle.com/c/dogs-vs-cats