Minimizing the generalization error

MMAI 5500 – lecture 3
Fall 2024

Content

Capacity, overfitting & underfitting

Regularization

- Weight decay
- Data augmentation
- Dropout
- Parameter tying & sharing
- Early stopping

A final repetition

Tutorial

- Visualize regularization

Additional reading

Chapter *7 Regularization for Deep Learning* in *Deep Learning*

by Aaron Courville, Ian Goodfellow, Yoshua Bengio

The book is comprehensive, written by leading deep learning researchers & freely available online.

Capacity, overfitting & underfitting

Network size & local minima

Networks with more hidden units have more & better local minima.

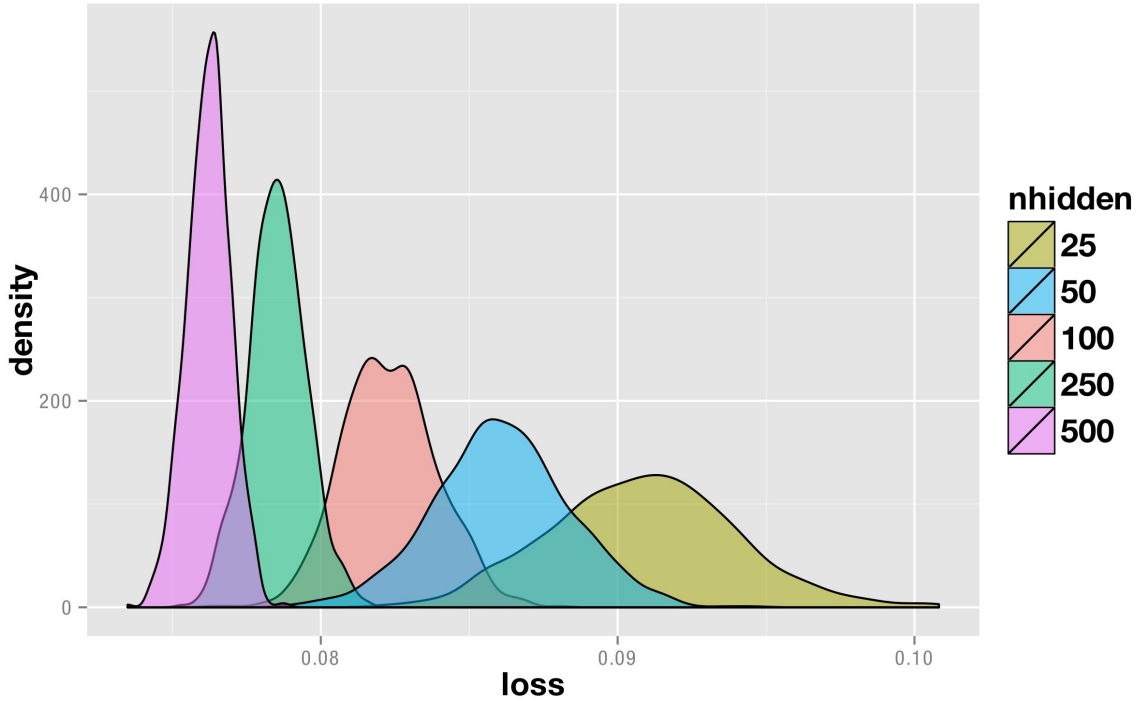
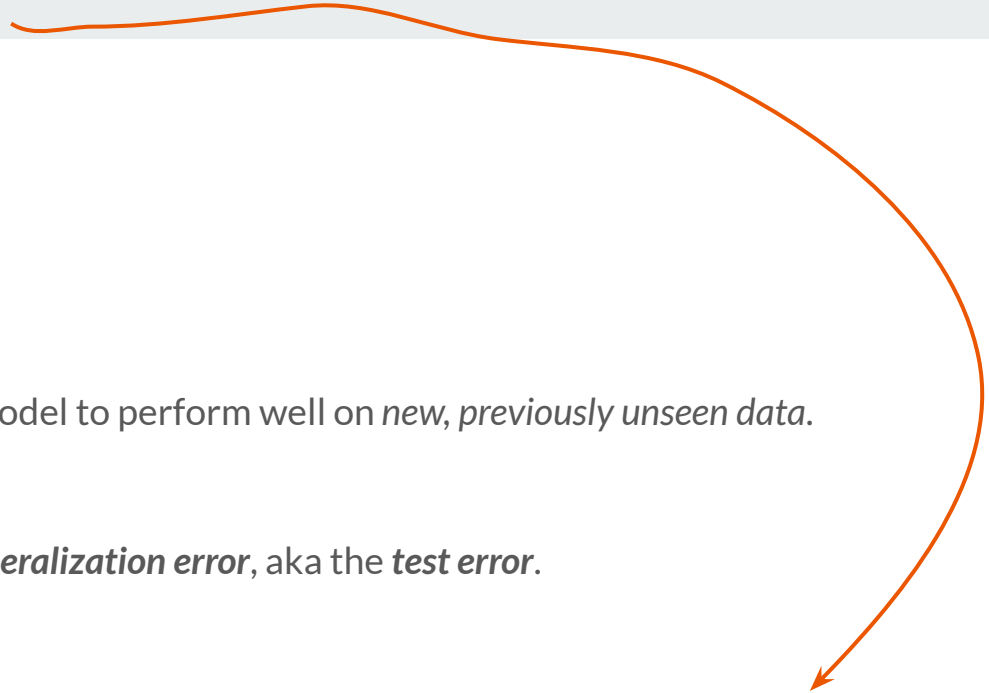


Image credit: Choromanska A, Henaff M, Mathieu M, Arous GB, LeCun Y (2014) The Loss Surfaces of Multilayer Networks; Arxiv

Question: What is this data set normally called?



The goal of ML

The goal of machine learning is for the model to perform well on *new, previously unseen data*.

That is, it has to ***generalize***.

Specifically, we want to minimize the ***generalization error***, aka the ***test error***.

Generalization error

- Expected error on a new, unseen, input. The expectation is taken over different inputs, drawn from a distribution representative of what the model is expected to encounter during practical use.

The goal in two parts

Small training error.

- Minimizing the training error without also keeping the difference between training & test error small leads to ***overfitting***.

Small difference between training & test error.

- Minimizing the difference between training & test error without also minimizing the training error leads to ***underfitting***.

Over- & underfitting are the two central challenges in ML.

Whether a model over- or under-fits depends on its ***capacity***.

Model capacity

Capacity – a model's ability to fit a wide variety of functions.

Low capacity

- The model cannot fit the training data well.

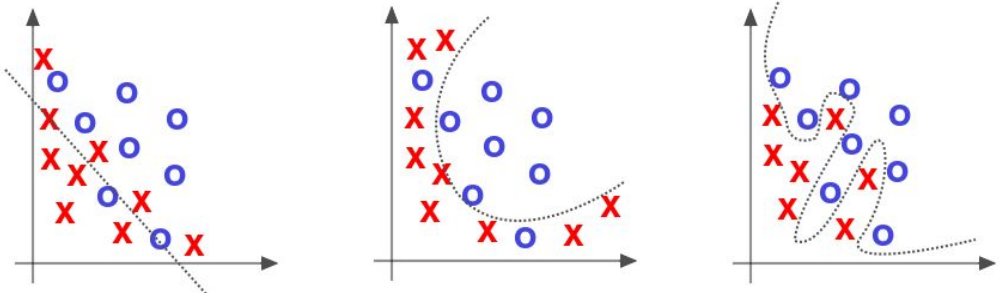
High capacity

- The model can learn properties (e.g. specific noise) of the training data that are absent in the test data.

The model capacity should match the complexity of the task.

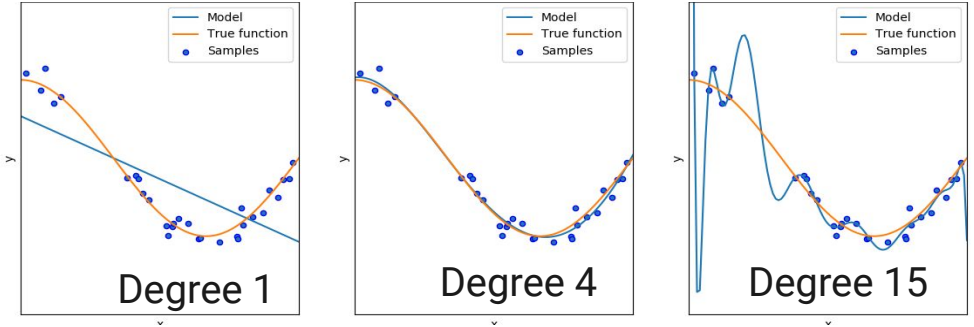
Examples of capacity

Classification

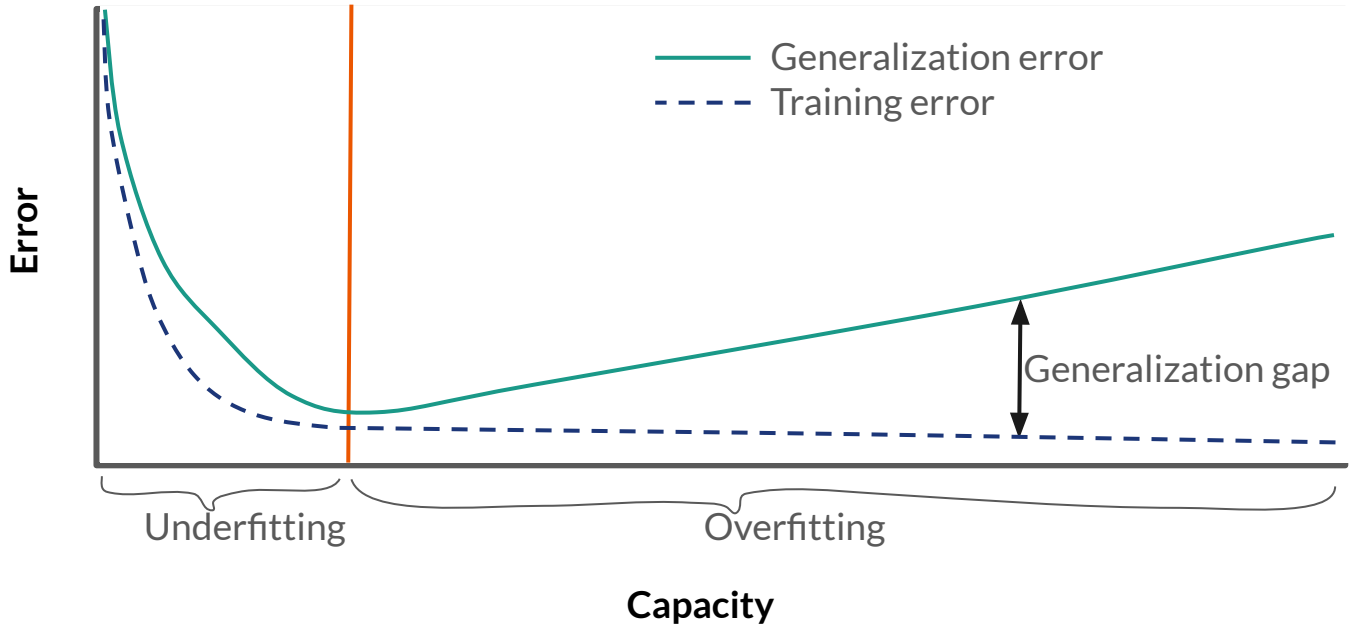


Too low capacity | **Good capacity** | Too high capacity

Regression



Model capacity



How to minimize the generalization gap

- Reduce the number of features.
  - Often used of linear regression.
- Reduce the model's capacity.
- Reduce the magnitude of parameters.
  - Good for many features where all contribute a bit to predicting the target.
- Early stopping.
  - Stop training when validation error increase (see later slides).
- More data.
- Data augmentation.
- Dropout.
- Weight tying & sharing.

Regularization

Regularization

Definition (Deep Learning by Aaron Courville, Ian Goodfellow & Yoshua Bengio):

> "*Any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error.*"

Examples:

- Constraints & penalties designed to encode specific kinds of prior knowledge.
- Constraints & penalties designed to give preference to simpler models.
- Constraints & penalties necessary to make an underdetermined problem determined.
- Combining multiple hypotheses as done with ensemble methods.

I.e. not only weight decay (see later slides).

Weight decay

Question: What is θ?

Weight decay

- L_1 & L_2

But, why?

Weights vs biases – *Typically only the weights are regularized since they need more data to fit accurately.* Fitting a weight needs observations of the previous & following neurons, whereas a bias is connected to only a single neuron (less data).

L_2

- Aka ridge regression & Tikhonov regularization.
- Sum of the square of the parameters.
- Most common.

$$L_2 = \lambda \sum \theta^2{}_i$$

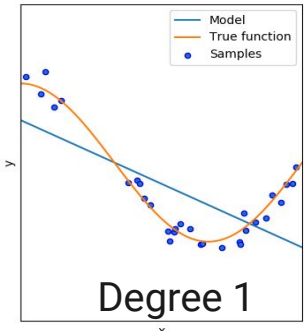
L_1

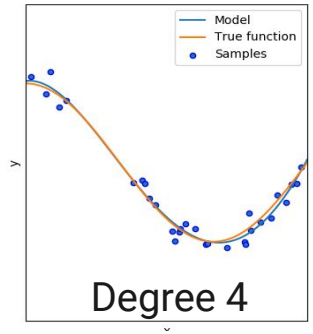
- Sum of the absolute value of the parameters.
- Used e.g. in LASSO regression

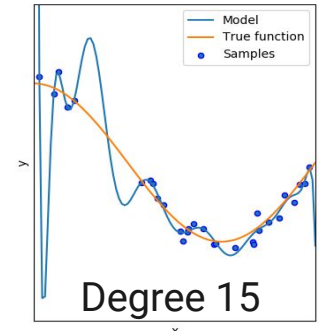
$$L_1 = \lambda \sum \left| \theta_i \right|$$

Weight decay

- Illustration







High bias, low variance
→ **underfit**

Just right

Low bias, high variance
→ **over fit**

Regression example

- Fit a polynomial
  - 1st degree: $h_w(x) = w_1 x + b$
  - 2nd: $h_w(x) = w_1 x + w_2 x^2 + b$
  - nth: $h_w(x) = w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_{n-1} x^{n-1} + w_n x^n + b$
- Minimize the error

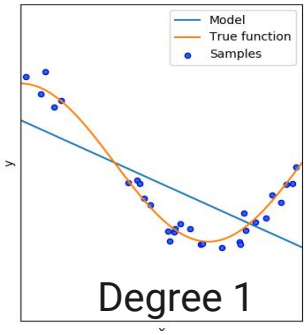
$$\min_w = \frac{1}{2m} \sum_{i=1}^{m} (h_w(x_i) - y_i)^2$$

Question: What type of regularization?

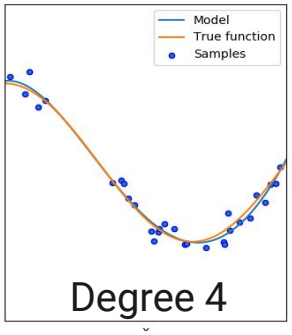
Weight decay

- Illustration

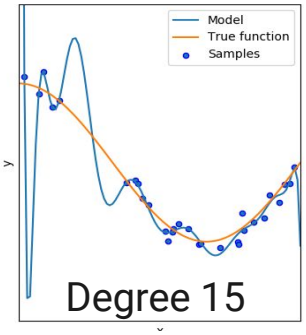
Regression example



High bias, low variance
→ **underfit**



Just right

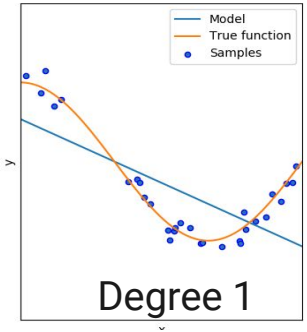


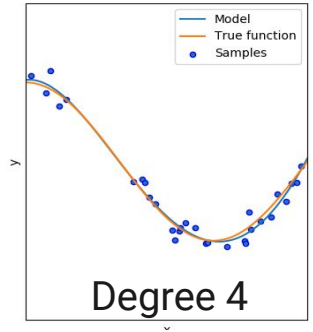
Low bias, high variance
→ **over fit**

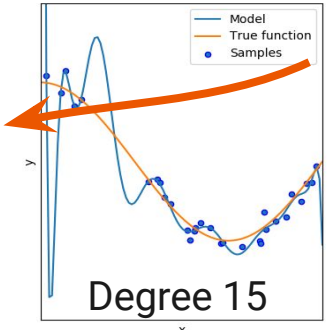
- Regularization term
  - Minimize $\lambda \sum_{i=1}^{m} (w_i)^2$
- Combined error & regularization terms
  - Minimize $\min_{w} = \frac{1}{2m} \sum_{i=1}^{m} (h_w(x_i) - y_i)^2 + \lambda \sum_{i=1}^{m} (w_i)^2$

Weight decay

- Illustration







High bias, low variance
→ **underfit**

Just right

Low bias, high variance
→ **over fit**

Regression example

$$\min_{w} = \frac{1}{2m}\sum_{i=1}^{m}\left(h_w(x_i) - y_i\right)^2 + \lambda\sum_{i=1}^{m}\left(w_i\right)^2$$

Parameters that do not contribute much to minimizing the prediction error get smaller.

- Lambda (λ)
  - Regularization parameter
  - Trades off between minimizing data error & "simple" model

$$h_w(x) = w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_{n-1} x^{n-1} + w_n x^n + b$$

($w_{n-1} x^{n-1} \approx 0$, $w_n x^n \approx 0$)

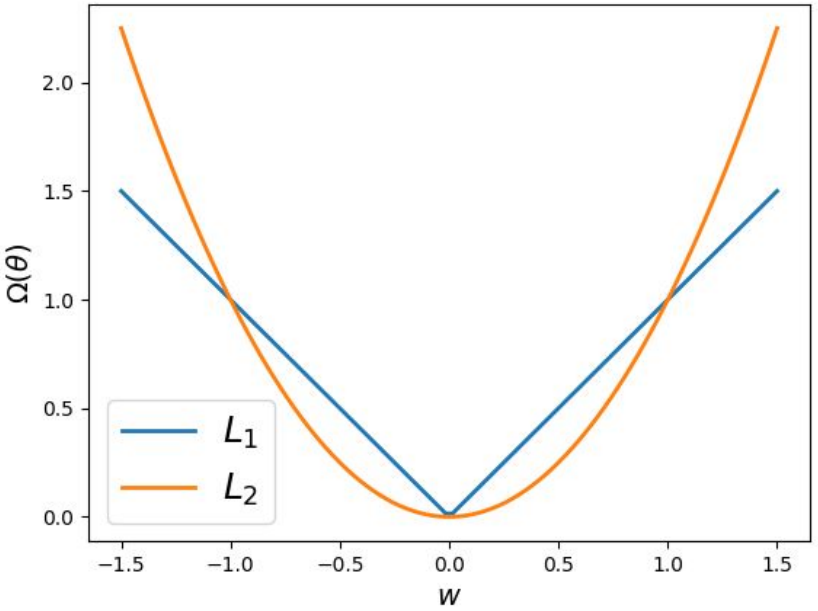
Question: What does sparse mean in this context?
Question: What does sparsity have to do with feature selection?

L_1 vs L_2

L_1 regularization results in more sparse solutions.

For this reason it is often used for *feature selection*.

However, outside of feature selection L_2 is most common.



Partial derivatives of L_1 & L_2

Both L_1 & L_2 have simple derivatives.

$$\frac{\partial L_1}{\partial \theta_i} = \lambda \begin{cases} 1 & \theta_i \geq 0 \\ -1 & \theta_i < 0 \end{cases}$$

$$\frac{\partial L_2}{\partial \theta_i} = \lambda 2\theta_i$$

Example NumPy implementation of L_1

```
import numpy as np

def get_dL1(weights, lmbd):
     dL1 = np.ones_like(weights)
     dL1[weights < 0] = -1
     dL1 *= lmbd

     return dL1
```

```
lmbd = 0.1
weights = np.random.randn(2, 3)

weights
     array([[0.38, -1.47,  0.39],
            [-0.57, -0.18, -0.71]])

dL1 = get_dL1(weights, lmbd)
dL1
     array([[0.1, -0.1, 0.1],
            [-0.1, -0.1, -0.1]])
```

Data augmentation

Question: Why is this easiest for classification?

Data augmentation

The best way to decrease the generalization error is to train on more data.

If more data is not available then the existing data can be augmented with fake data.

Easiest for classification, especially object recognition, but has been effective for speech recognition too.

Image credit: Shorten & Koshgoftaar (2019) A survey on Image Data Augmentation for Deep Learning

Image data augmentation

This is a field of active research.

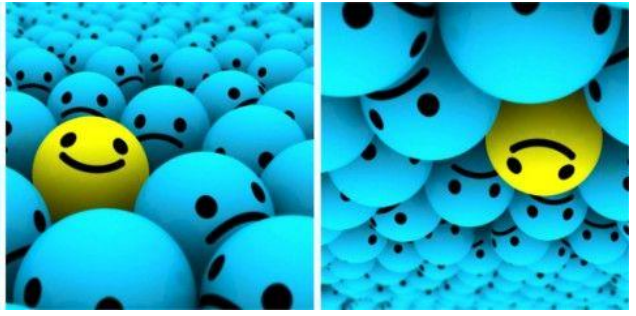
Question: What type of transformation was used on each of the rows (row 1, 2 & 3)?
Question: How good would this be for OCR?

Image data augmentation

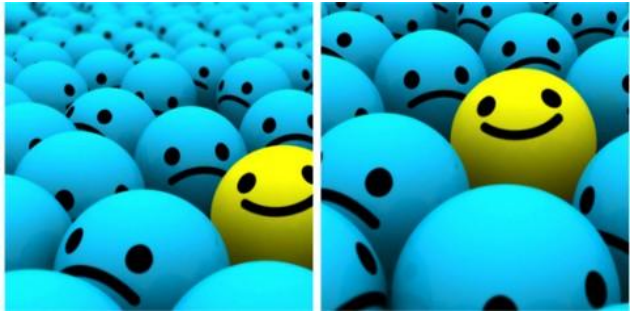
- Geometric transformations

Original

1



2



3

Image credit:
https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Image data augmentation

- Color augmentation

Color augmentation for melanoma (a type of skin cancer) classification.

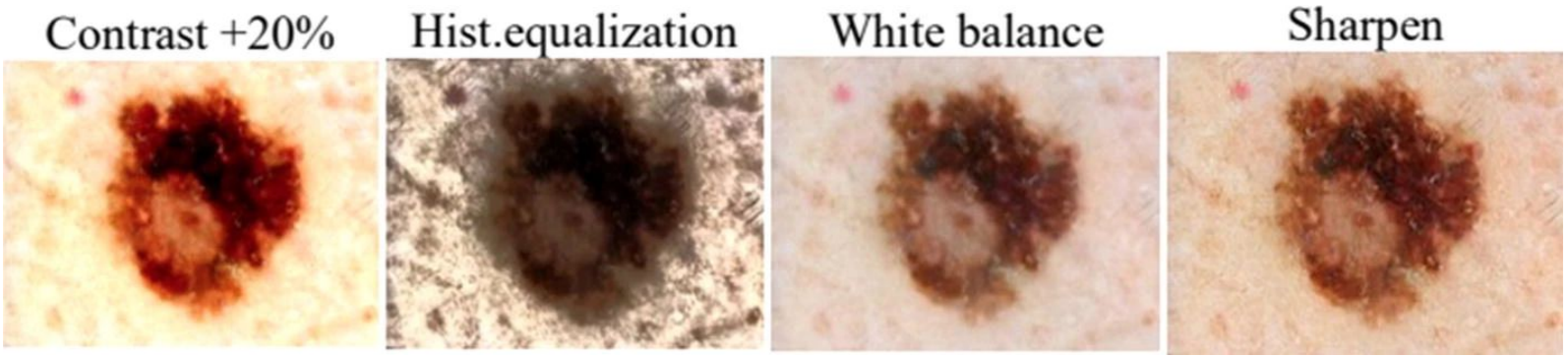


Image credit:
Agnieszka M, Michal G. (2018) Data augmentation for improving deep learning in image classification problem.

Color augmentation

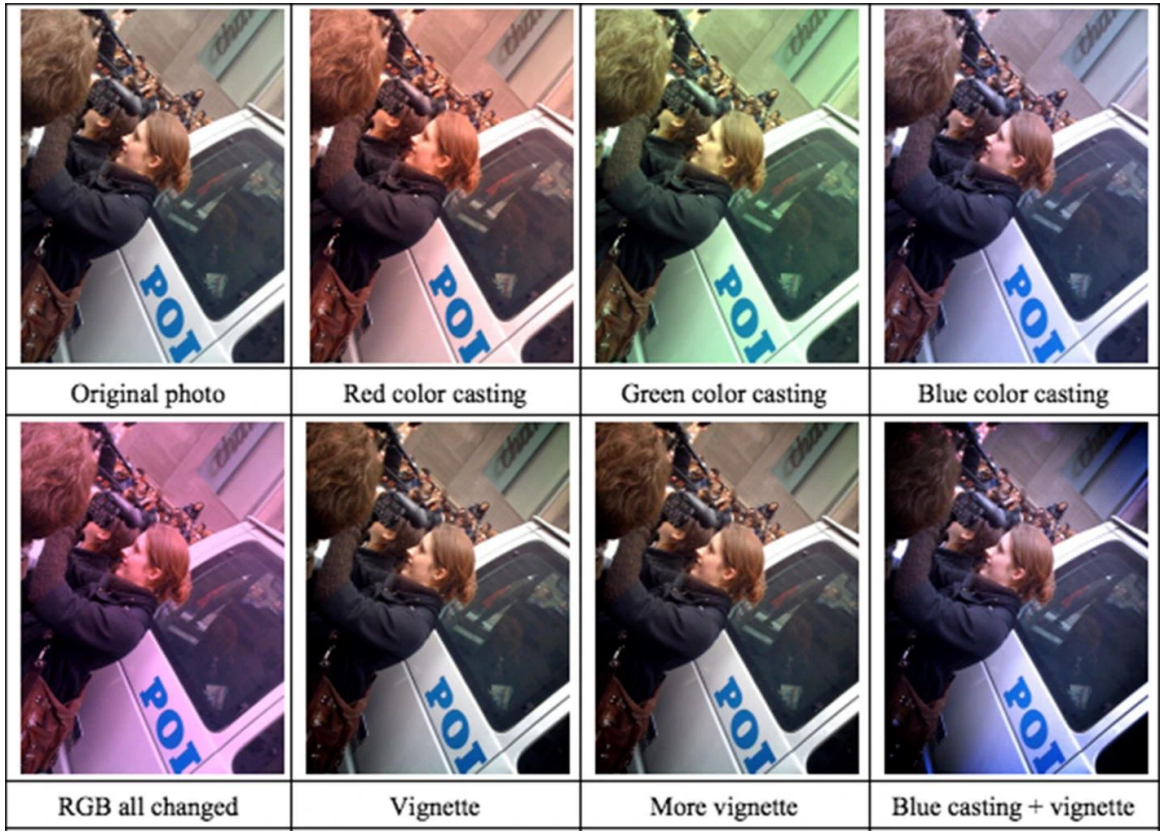


Image credit: Ren W, Shengen Y, Yi S, Qingqing D, Gang S. (2015) Deep image: scaling up image recognition.

Randomize the pixels within the kernel

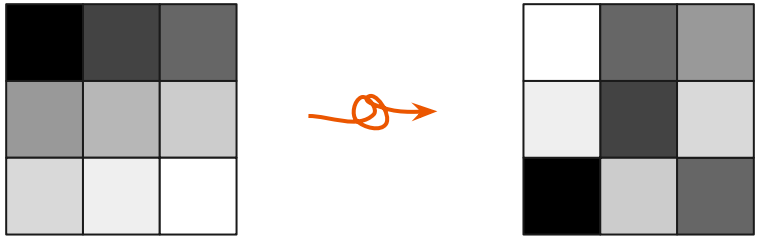
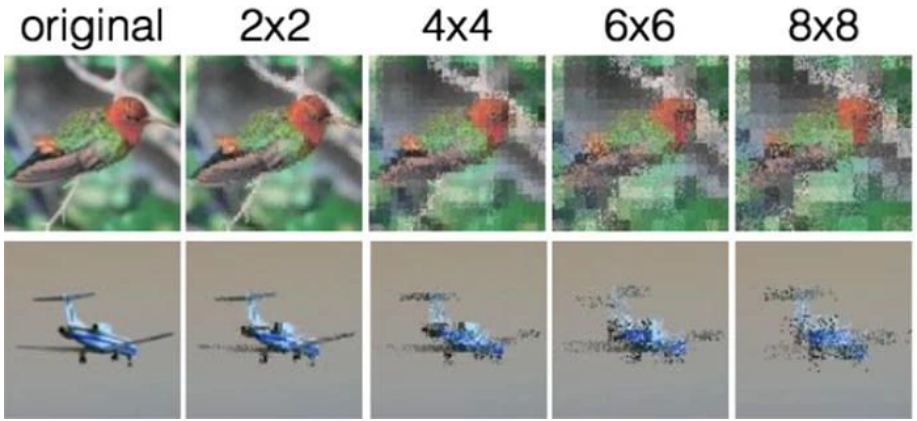


Image data augmentation

- Kernel filter randomly shifting pixels



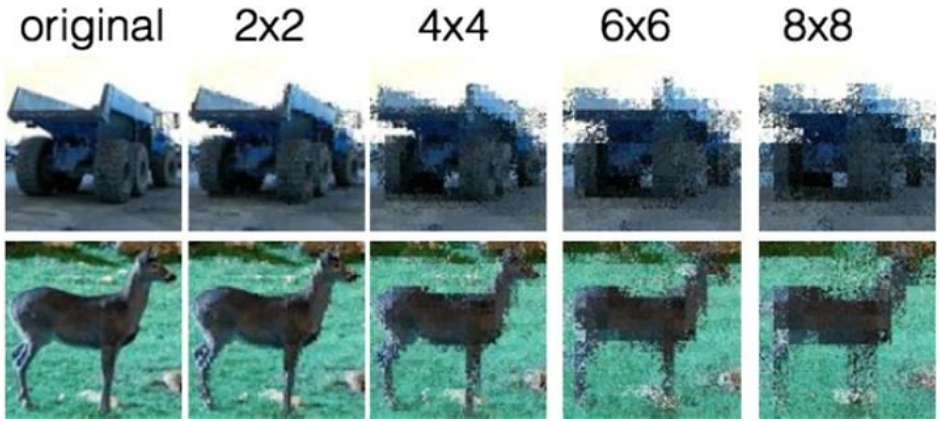


Image credit: Shorten & Koshgoftaar (2019) A survey on Image Data Augmentation for Deep Learning

Image data augmentation

- Noise

Image credit: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Data augmentation with noise

Many classification & some regression tasks should be robust to small amounts of random noise.

This is used for some unsupervised learners, e.g. denoising autoencoders (MMAI 5500 lecture 6).

Noise doesn't have to be added to the images but also be added to the hidden neurons (at a high level, this can also be seen as data augmentation).

Dropout (see below) can be seen as constructing new inputs by multiplying with a noise mask.

Dropout

Question: What is bagging?
Question: Why *approximate* bagging? Isn't it better to just do it?

Dropout

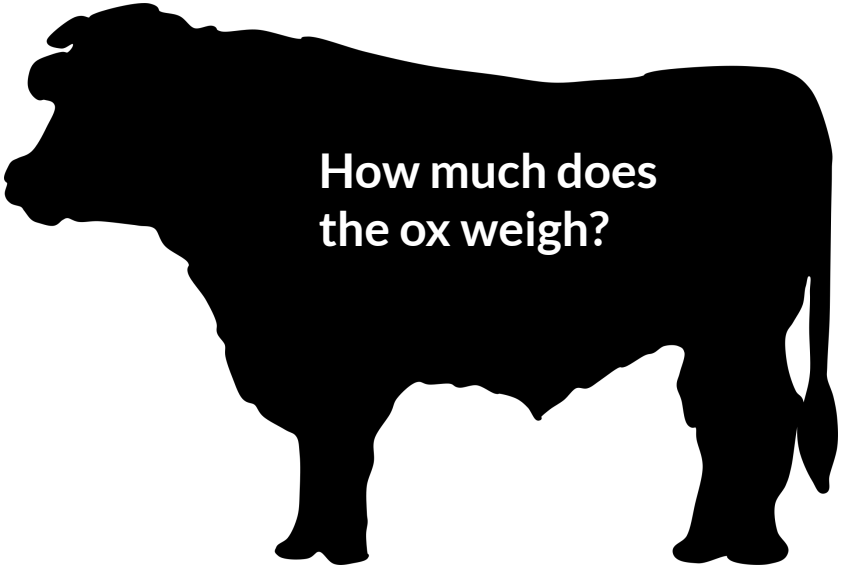
Dropout was introduced by Srivastava (Hinton's lab) in 2012.

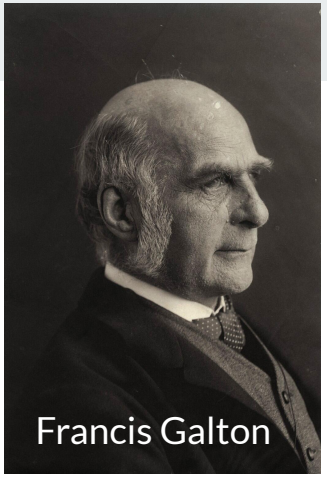
It can be seen as a computationally efficient approximation to *bagging* for neural networks (& a form of data augmentation).

A certain fraction of neurons are randomly disabled during the forward pass. That is, a new random mask is created for each data example.

The hyper-parameter `dropout_rate` (or dropout probability) controls the fraction of masked neurons.

Bagging & the wisdom of crowds





In 1906, 800 persons participated in a contest to guess the weight of an ox at a livestock fair.

Galton observed that the median weight was within 0.8% of the true weight, much better than the average individual guess.

I.e. the crowd was wiser than any individual.

Image credits:
Ox by HKT
Francis Galton by Eveleen Myers (née Tennant) http://www.npg.org.uk/collections/search/portrait/mw127193, Public Domain, https://commons.wikimedia.org/w/index.php?curid=61305409

Dropout

Dropout approximates bagging

- Dropout trains the ensemble consisting of all subnetworks that can be formed by removing hidden neurons from the network.

Dropout has been shown to work better than, e.g., weight decay. It can also be combined with other forms of regularization, & works well with many different architectures & training methods.

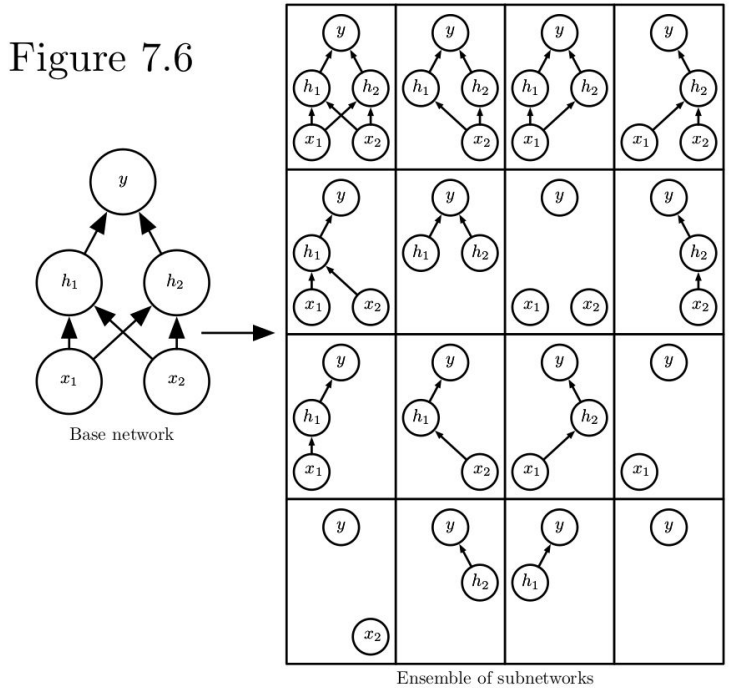
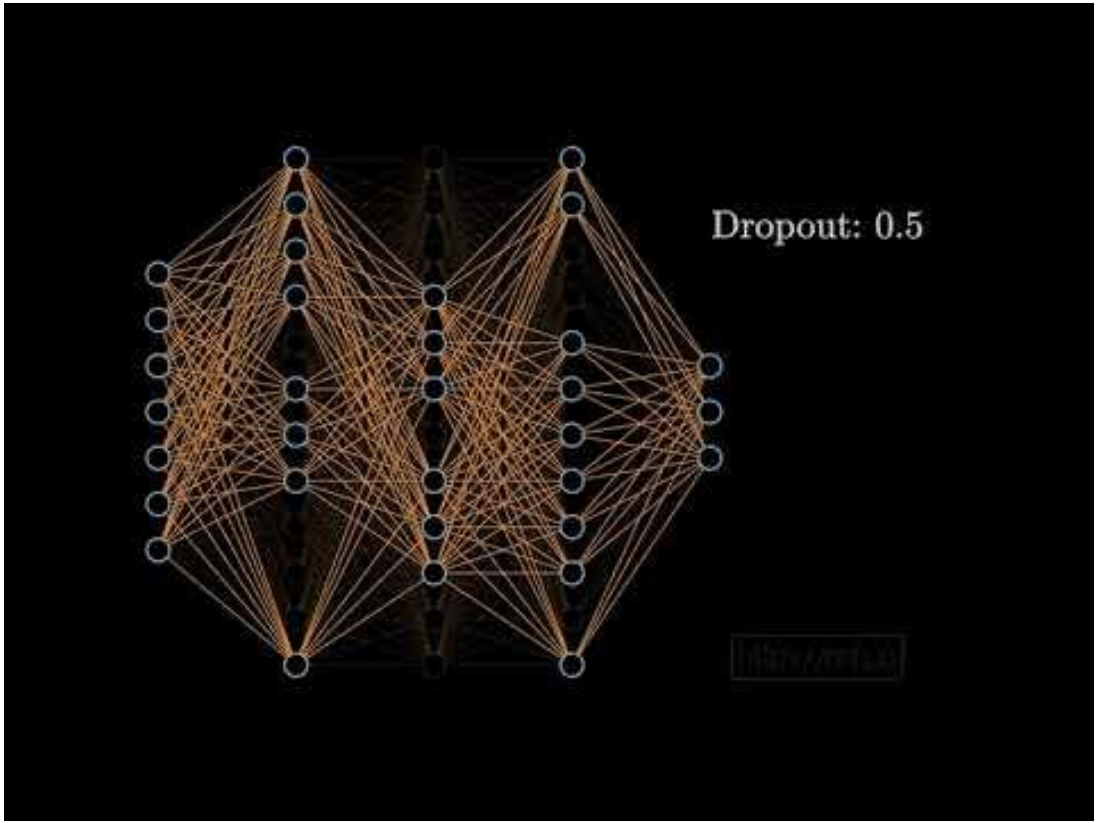


Image credit: Deep Learning by Aaron Courville, Ian Goodfellow, Yoshua Bengio (2016)

Dropout visualization



Dropout

- Prediction

When we make a prediction (aka inference) we don't want to use a single subnetwork (mask) but get the average over all subnetworks.

Not a great option

- If we were to mimic bagging then we could average the output of 10-20 masks.
- That would require 10-20 forward passes.

The **good** option: weight scaling

- Do the forward pass without any mask (i.e. no dropout).
- But, during training scale the weights by the inverse of their dropout rate.
- E.g. if the dropout rate was 0.5 then scale the weights by 2.

Example NumPy implementation of dropout

Use Numpy's `random.binomial(n, p, size=None)`. It draws samples from a binomial distribution. The arguments are, `n` trials (integer ≥ 0), `p` probability of success (interval $[0, 1]$) & `size` the size of the returned array.

```
import numpy as np

dropout_rate = 0.3
activations_output *= np.random.binomial(1, 1 - dropout_rate, activations.shape)
```

Random.binomial
https://numpy.org/doc/stable/reference/random/generated/numpy.random.binomial.html

Example NumPy implementation of dropout

```
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        self.rate = 1 - rate   # Store inverted rate.
                               # E.g. for dropout 0.1 -> success rate of 0.9

    def forward(self, inputs):  # Forward pass

        self.inputs = inputs   # Store input values
        # Generate and store scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                                              size=inputs.shape)
        self.output = inputs * self.binary_mask / self.rate # Apply scaled mask

    def backward(self, dvalues):  # Backward pass

        self.dinputs = dvalues * self.binary_mask # Gradient on values
```

Parameter tying & sharing

Question: Why does parameter tying/sharing regularize?

Parameter tying & sharing

In some situation we have multiple networks that have to learn mappings from very similar input & output distributions. This can be used to regularize the network(s) in 2 ways: *parameter tying* & *parameter sharing*.

Parameter tying

- Penalize the parameter sets (θ^A & θ^B) for being different. E.g. with the L_2 penalty.
- Not very common.

$$\Omega(\theta^A, \theta^B) = (\theta^A - \theta^B)^2$$

Parameter sharing

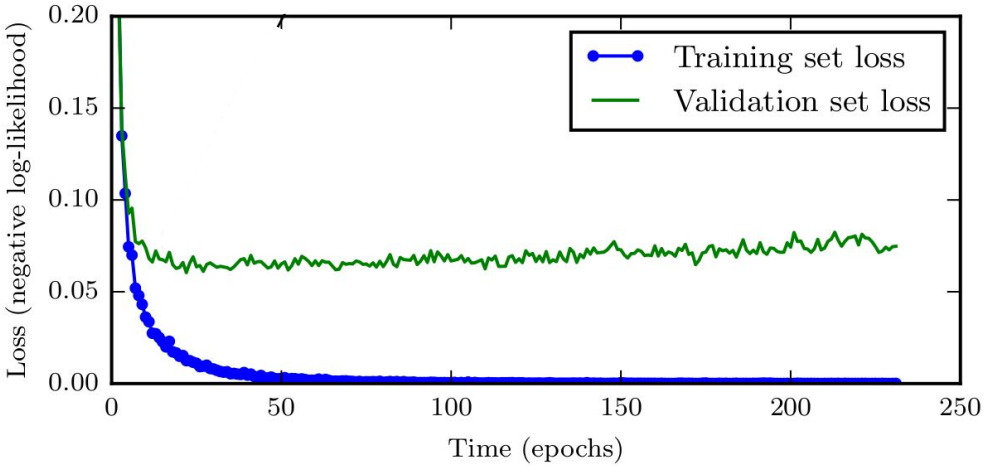
- Require the sets of parameters to be equal.
- Uses less memory.
- Common. E.g. in CNNs & Autoencoders.

$$\theta^A = \theta^B$$

Early stopping

Question: How many epochs should this network be trained?

Early stopping



Store a copy of the parameters every time the validation loss improves.

Restore the best parameters (not necessarily from the last epoch) for prediction/inference.

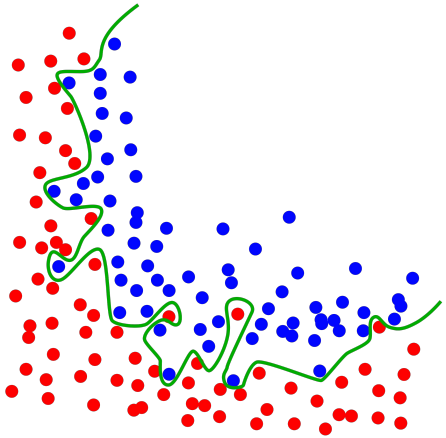
Early stopping is probably the most popular regularization method in deep learning.

To utilize all data (also the validation data), the network can be re-trained on the entire data set (no validation set) for the number of epochs determined by early stopping.

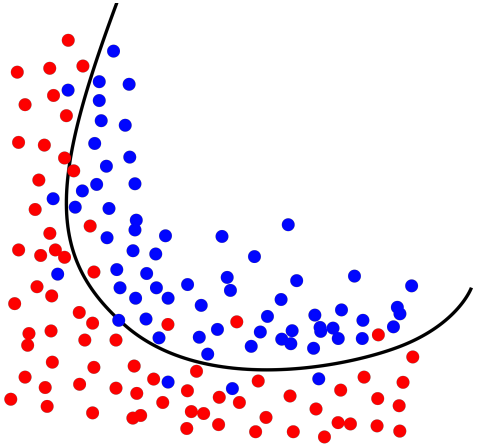
A final repetition

Question: Which of the two models will perform best on the *training data*?

Evaluating performance



Green line: unregularized model

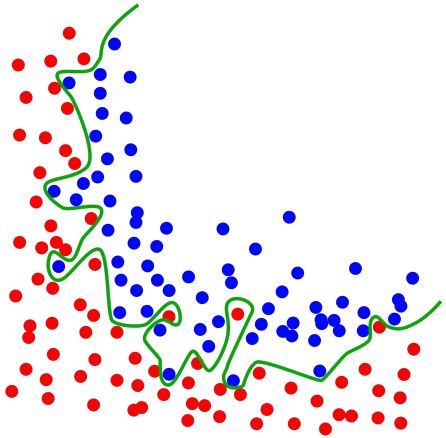


Black line: regularized model

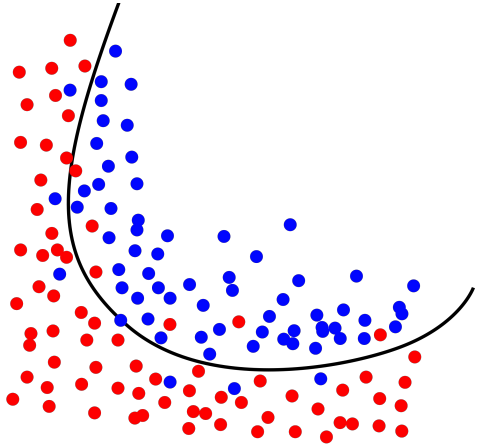
Image credit: Chabacano - CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=3610704

Question: Which of the two models will perform best on *previously unseen* data (i.e. not training data)?

Evaluating performance



Green line: unregularized model



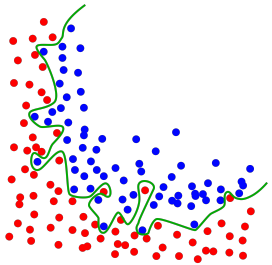
Black line: regularized model

Image credit: Chabacano - CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=3610704

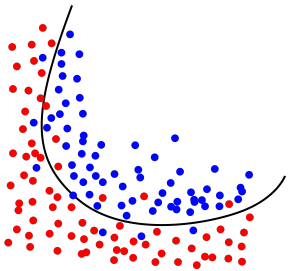
Evaluating performance

We want to estimate how well the model will perform in use, i.e. when it receives new input.

- The performance on the training data is irrelevant (only useful for training).
- The original data set is split into a ***training set*** (~70%) & a ***test set*** (~30%).
- The test set is only used to evaluate performance.
- The regularized model should perform better on the test set.



Green line: unregularized model



Black line: regularized model

Image credit: Chabacano - CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=3610704

Question: How can we do *model selection* if we're can neither use the training set nor the test set?

Overfitting is sneaky

Overfitting occurs when we select things (e.g. weights, tree-splits, learners, …) based on performance (e.g. lower error or better accuracy).

What happens when we select hyperparameters &/or algorithms* based on performance on the test set?

- We risk overfitting on the test set so that the test set performance will no longer be representative for unseen data.
- I.e. *we should **not** use the test set for model selection.*

* Selecting the best model among different algorithms & hyperparameter settings is called ***model selection***

Question: How can we do *model selection* if we're can neither use the training set nor the test set?

Overfitting is sneaky

Overfitting occurs when we select things (e.g. weights, tree-splits, learners, …) based on performance (e.g. lower error or better accuracy).

What happens when we select hyperparameters &/or algorithms* based on performance on the test set?

- We risk overfitting on the test set so that the test set performance will no longer be representative for unseen data.
- I.e. *we should **not** use the test set for model selection.*

We need a 3rd partition: a *validation set* (aka a *dev set*) for model selection.

* Selecting the best model among different algorithms & hyperparameter settings is called ***model selection***

Data partitioning

Training set (e.g. 60% of total data set)

- Used to learn model, e.g. fit weights &/or learn trees

Validation set (e.g. 20% of total data set)

- Used for model selection, e.g. choosing algorithm, tuning hyperparameters or for early stopping [e.g. deciding when to stop building a decision tree (slide 12) or stop training a neural network]

Testing set (e.g. 20% of the total data set)

- Used only to evaluate the performance of the final model
- No decisions should be made based on the test set

Tutorial

MMAI5000_class03_regularization.ipynb