AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering
Department of Biocybernetics and Biomedical Engineering

Computational Intelligence

Optimization and Regularization

Adrian Horzyk
horzyk@agh.edu.pl

Google: Adrian Horzyk

Introduction

Why do we need to optimize and regularize?

Optimization and Regularization

When dealing with artificial neural networks, we come across various difficulties with adapting the designed models and achieving good generalization properties that determine possible practical uses and model performance.

This is why we:

- choose a loss function and define a cost function that will measure the model performance to minimize mistakes made by the model,
- optimize neural network parameters during the training process,
- set up or search for appropriate hyperparameters,
- implement various regularization techniques during training,
- augment data or use transferred models trained of bigger datasets

to satisfy the desired requirements and goals of the constructed model for a given dataset.

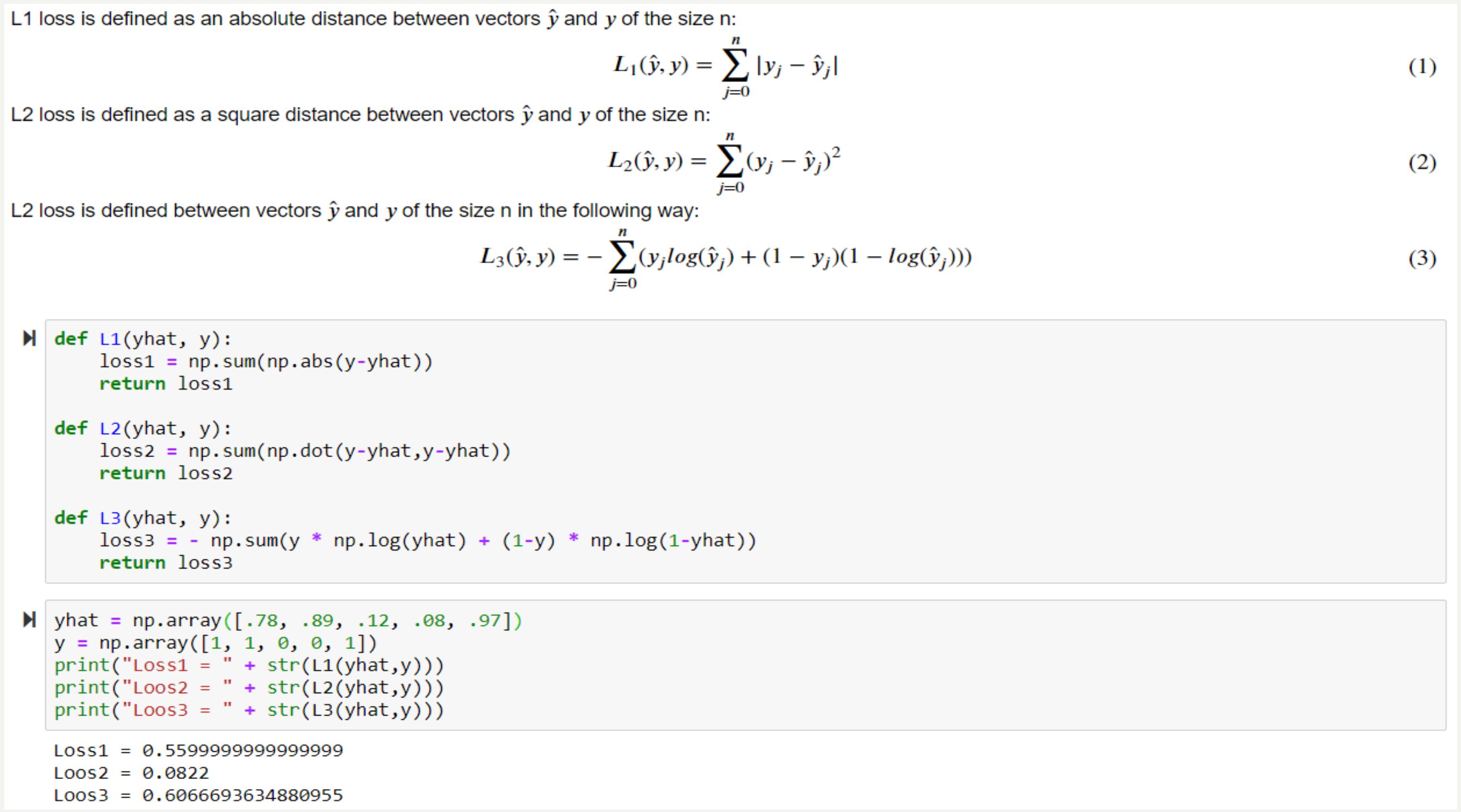
Loss Function and Cost Function

How to measure the model error for training data?

Loss Functions

First of all, we need to choose a loss function, which determines the method of calculating the model loss on training examples for the current state of network parameters (e.g. weights). We use predictions and desired values to calculate it.

The loss function can be specified individually for each output value.
It is usually the same for all outputs; however, it does not need to be.



L1 loss is defined as an absolute distance between vectors $\hat{y}$ and y of the size n:

$$L_1(\hat{y}, y) = \sum_{j=0}^{n} |y_j - \hat{y}_j| \tag{1}$$

L2 loss is defined as a square distance between vectors $\hat{y}$ and y of the size n:

$$L_2(\hat{y}, y) = \sum_{j=0}^{n} (y_j - \hat{y}_j)^2 \tag{2}$$

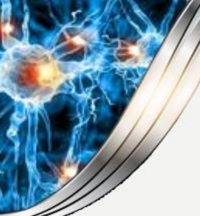
L2 loss is defined between vectors $\hat{y}$ and y of the size n in the following way:

$$L_3(\hat{y}, y) = -\sum_{j=0}^{n} (y_j log(\hat{y}_j) + (1 - y_j)(1 - log(\hat{y}_j))) \tag{3}$$

```
def L1(yhat, y):
    loss1 = np.sum(np.abs(y-yhat))
    return loss1

def L2(yhat, y):
    loss2 = np.sum(np.dot(y-yhat,y-yhat))
    return loss2

def L3(yhat, y):
    loss3 = - np.sum(y * np.log(yhat) + (1-y) * np.log(1-yhat))
    return loss3
```

```
yhat = np.array([.78, .89, .12, .08, .97])
y = np.array([1, 1, 0, 0, 1])
print("Loss1 = " + str(L1(yhat,y)))
print("Loos2 = " + str(L2(yhat,y)))
print("Loos3 = " + str(L3(yhat,y)))
```

```
Loss1 = 0.5599999999999999
Loos2 = 0.0822
Loos3 = 0.6066693634880955
```



Cost Function

We also define a cost function because the entire gradient descent process must be based on a single scalar value that is minimized. Therefore, for multiple output loss functions, we define a cost function that is typically an average of the losses calculated for all outputs and all training examples.

Finally, we define a cost function that measures the error on the entire training data set (for all examples):

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)\right)$$

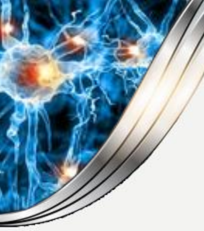
Here, we used the following loss function: $L_3(\hat{y}, y) = -\left(y \log \hat{y} + (1 - y)\log(1 - \hat{y})\right)$

For the given training data set $\left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \dots, \left(x^{(m)}, y^{(m)}\right)\right\}$ we want to get $\forall_i\ \hat{y}^{(i)} \approx y^{(i)}$

where $\hat{y}^{(i)} = \sigma\left(w^T x^{(i)} + b\right)$ and $\sigma\left(z^{(i)}\right) = \frac{1}{1+e^{-z^{(i)}}} \in (0, 1)$ $\quad$ (i) is the notation for i-th example

The success of the network training depends on the defined cost function (also called goal function or error function), that is minimized to maximize success measured by different scores like accuracy, recall, F1-score, etc.

Example: If the intention of learning were to maximize the welfare of all people, and the average welfare was taken, then a poorly chosen goal function could achieve the goal by removing all poor people to maximize this average welfare, instead of raising the standard of the poorer people!



The Most Effective Cost Functions

For specific groups of problems, we usually use proven and effective categories of cost functions, e.g., to avoid getting stuck in the learning process in local minima or saddle points.

We typically use:

- **cross-entropy in the case of binary classification,**
- **categorical cross-entropy for multi-class classification,**
- **mean squared error for regression problems.**

Moreover, CTC (Connectionist Temporal Classification) is often used for sequential learning problems.

Goal functions should be created independently when working on a new research problem.

Parameters vs. Hyperparameters

What's the difference? How to optimize them?

Parameters vs. Hyperparameters

Parameters in DNN / CI / ML:

- are weights ($W^{[l]}$), biases ($b^{[l]}$), and other variables of the model that are updated (adjusted, tuned) during the training process according to the chosen training algorithm and its optimizer method.

Hyperparameters in DNN / CI / ML:

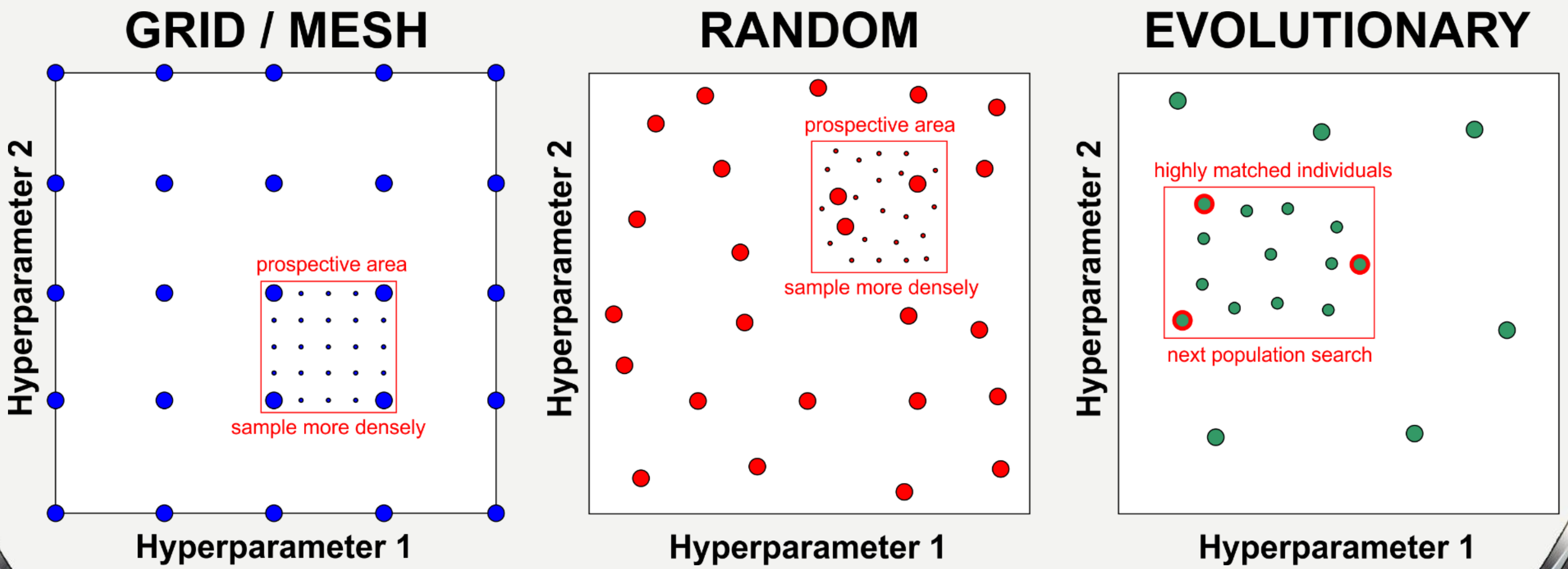
- are all variables and parameters of the model that are not adjusted by the training algorithm but by a designer of the DNN or an external method;
- are all parameters that can be changed independently of the way how the training algorithm works;
- can be adjusted by extra supporting algorithms like genetic or evolutional ones;
- a number of layers (L), a number of neurons in hidden layers ($n^{[l]}$);
- activation functions ($g^{[l]}$) and types of used layers, and weights initialization;
- learning rate (α), mini-batch size, number of epochs (training cycles);
- augmenting and normalizing training and testing (dev) data;
- dropout, momentum and other implemented regularization and optimization techniques and their parameters.

Optimization of Hyperparameters

In deep learning, we have a quite big number of hyperparameters that must be tuned to get a good enough computational model.

We have various techniques that help us to deal with this problem:

1. **Systematically choose hyperparameters over the grid (mesh), tightening the prospective areas (sampling more densely prospective areas) (computationally very expensive due to the huge number of combinations to check).**
2. **Choose hyperparameters randomly many times, sampling more densely prospective areas (uncertain but may be faster if you are lucky).**
3. **Use evolutional and genetic approaches (smart choice based on previous populations).**

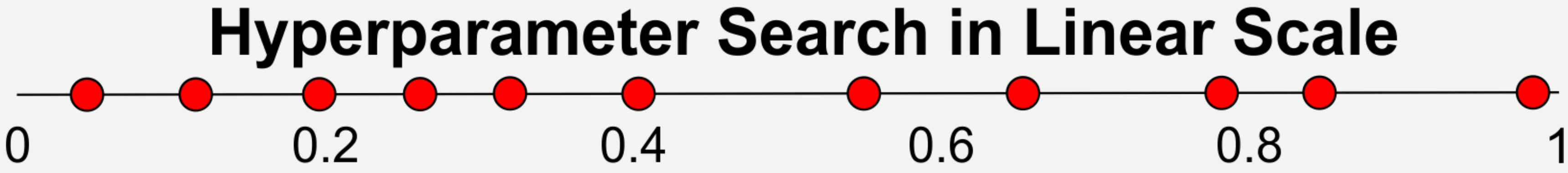


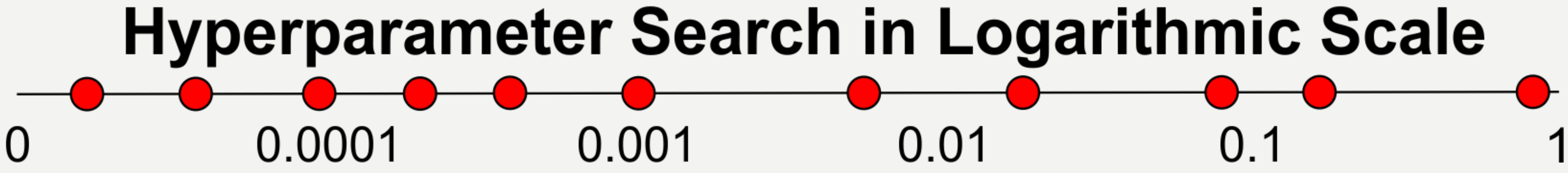
Scales of Hyperparameter Optimization

When sampling hyperparameters, we cannot simply scale them in a linear scale. Sometimes we need to use a different scale, e.g. logarithmic or exponential. Otherwise, we will sample not useful hyperparameters, not improving the developed computational model.

For example, when we want to sample learning rate, we should use a logarithmic scale, e.g.:

$$\alpha = 10^r \quad where \qquad r = -4 * np.random.rand()$$

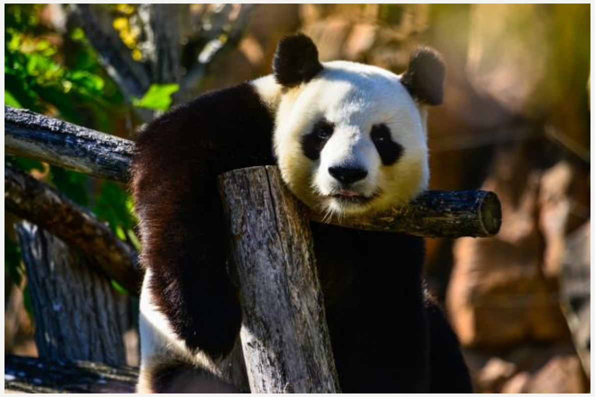




Approaches to Search for Hyperparameters

There at two main approaches to search for suitable hyperparameters:

- **A babysitting model (Panda strategy) – in which we try to look at the performance of a model and improve its hyperparameters patiently.**



- **Many models train in parallel (Caviar strategy) – check many models using various combinations of the hyperparameters and choose the best one automatically. If you have enough computational resources, you can afford this model.**

Training, Validation, and Test Data and Their Distributions

How to divide data?

Division of Data

To train, validate, and test a model, we usually use:

- Training examples (train set) for adjusting a model not to underfit (decreasing bias) during the training process.
- Validation examples (dev set) for validating the training progress and controlling overfitting of a model (decreasing variance) during the training time.
- Testing examples (test set) for checking the generalization properties of the trained model (decreasing variance).

Sometimes, we don't use a test set (only checking the model during its training process) or a dev set (only checking the model after the training process is finished).

We usually follow the chain of goals when developing and training the model:

1. Fit a train set well on a cost function, trying to achieve the human-level performance.
2. Fit a dev set (validation set) well on a cost function to get good generalization properties verified during the training process.
3. Fit a test set well on a cost function to be sure that the generalization is good enough, and the model tested on the data that have not been used during the training process.
4. Next, we hope that the model will perform well on real-world data.

If the model does not fit well in any of the first three steps, we need to know what we can do with the hyperparameters of the model to achieve the desired goal and generalization! Therefore, we need to define knobs that help us control the training process to fit the model well.

Splitting data into train, dev, and test sets?

Splitting data for small datasets (< 100 thousands examples), e.g.:

- Train set : dev set : test set = 60% : 20% : 20%
- Train set : dev set : test set = 70% : 15% : 15%
- Train set : dev set : test set = 80% : 10% : 10%

Splitting data for large datasets (minions of examples), e.g.:

- Train set : dev set : test set = 98% : 1% : 1%
- Because training data today a have huge amount of training examples, 1% of them is usually enough to validate or test the model (e.g., 1% from 1.000.000 is 10.000 examples for validation or testing), and thanks to it, we can use more examples for training the model!
- The test set should be big enough to give high confidence in the overall performance of a trained system or a solved task.

Distributions of train, dev, and test sets

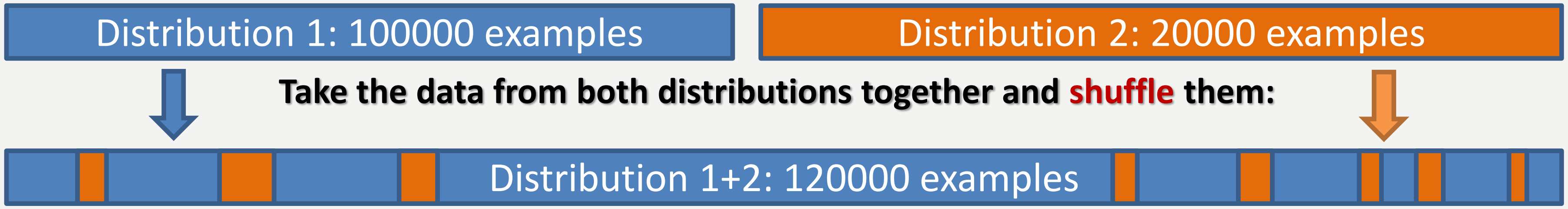
Train, dev (validation), and test sets should be set up in such a way that they **share data of all distributions** in the same way (**be representative** for the solved problem) to minimize variance and achieve good generalization properties, e.g.:

- When we would like to create a classifier or a predictor for data coming **from various data distributions**, e.g., different countries, ethnic groups, or companies of the world, we should take care about the way how are the data distributed to the train, dev, and test sets. On the other hand, we can train the model almost perfectly on the train set and validate it on the dev set, but it will not work on a test set and on real data!
- If we train and validate the model for data coming from Europe, it will rather not work for data coming from China or US and vice versa.
- If we use, e.g., images from different sources to train a model (e.g. a CNN), we must take care of the suitable division of the data **from each distribution** to the training, validation, and testing data. On the other hand, we don't be able to adjust the model and achieve high performance and generalization properties.
- If we train and validate the model on data coming from rich people, it will rather not work for people with low incomes and vice versa.
- If we train and validate the model for men, it cannot work for women etc.

Combining Data from Different Distributions

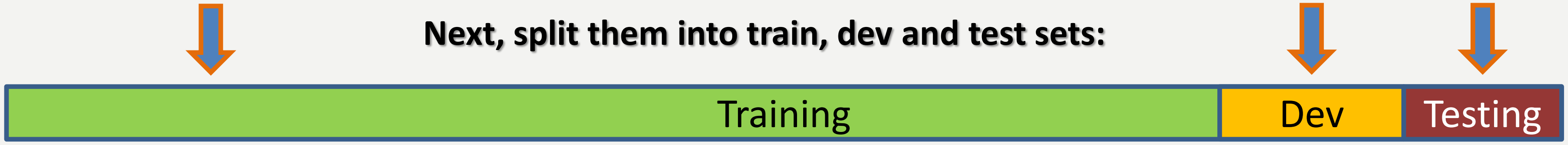
Construct train, dev and test sets from all distributions of the data from, e.g.:
(a possible data mismatch problem)

High-quality data distribution **Low-quality data distribution**

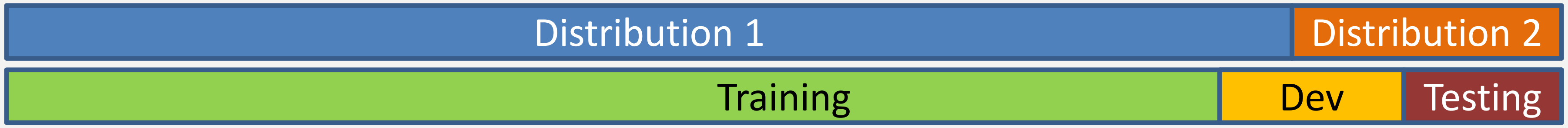

Distribution 1: 100000 examples
Distribution 2: 20000 examples
Distribution 1+2: 120000 examples

Take the data from both distributions together and shuffle them:

Next, split them into train, dev and test sets:



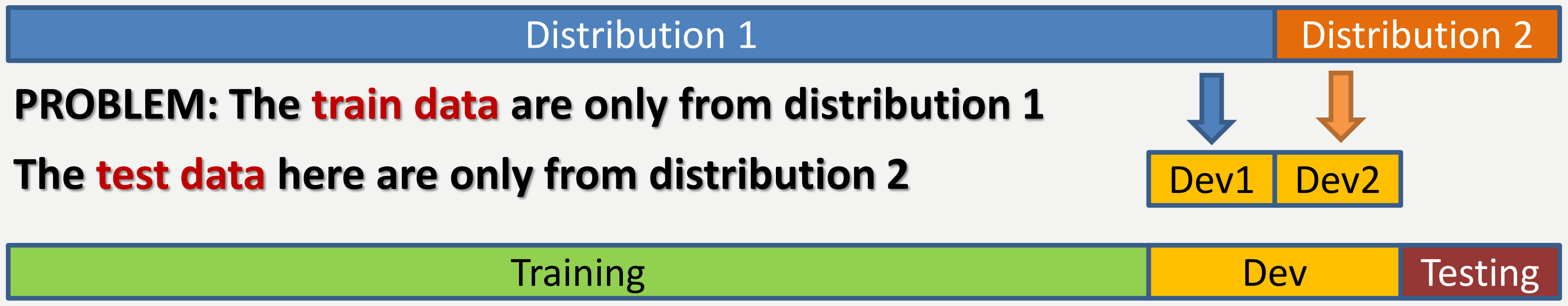
If you don't put the data together and shuffle them, they might be trained, validated, and tested on different distributions and dev and test results might be very poor:



To achieve good generalization, train, dev and test data must be representative of the problem.

Identifying Data Mismatch Problem

If the dev set is composed of various distributions, but the training set is taken only from one distribution (or a subset of distributions), then the dev error on the subsets of dev sets will differ!



PROBLEM: The train data are only from distribution 1

The test data here are only from distribution 2

In this case, we usually achieve big differences in the following errors:

human-level error < training error < dev1 error < dev error < dev2 error < testing error

This indicates the data mismatch problem, i.e. the model has been trained on a limited subset of distributions (not all distributions), so it will not work correctly on testing data.

The big difference between testing and training errors indicates the overfitting problem.

Bayes Optimal and Human Level Performances

How to validate the quality of our results?

When we cannot train the model better?

Bayes Optimal Performance and Error

Bayes optimal performance and error are defined by the blurred and noisy training examples that nobody and nothing can never recognize or differentiate them due to their low quality:

- They can never be surpassed (the best possible):

  Bayes optimal error ≤ Human-level, train, dev, or test error

  and

  Bayes optimal performance ≥ Human-level, train, dev, or test performance
- The Bayes optimal performance is always higher than or equal to human-level performance.
- The Bayes optimal error is always lower than or equal to a human-lever error.
- It is many times very close to the human-level performance, where humans are very good at.
- Sometimes, it is equal to the human-level performance when data are labelled by humans, so we cannot surpass this level in these cases.
- In some situations, it is difficult to determine Bayes optimal performance and error.

Human-Level Performance and Error

Human-level performance and error are defined by the human team of the world's best experts:

- They cannot be contemporary surpassed by any human or a human team.
- If they would be surpassed in the future for a task, then they automatically set a new human-level performance and a new human-level error for this task.

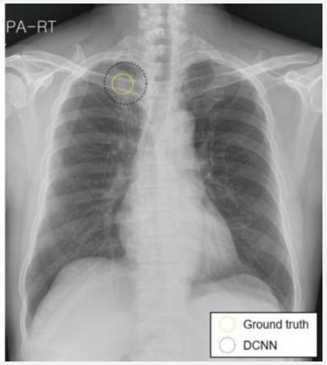
Human-level performance:

- is the classification/prediction performance achieved by the committee of highly expertise humans (e.g. surgeons, psychologists, teachers, engineers);
- is treated as a high bound and goal of training the model, which we should strive for;
- can be sometimes surpassed by machines and retrospectively checked by human experts, which can shift it to a higher level expanding the knowledge of the experts.

The **final performance** achieved by the model can surpass the human-level one, but we do not know how much better the final performance might be because nobody can contemporary do it better.

Human-Level Error

Suppose that we try to classify some medical images:



The classification can be made by different humans or their teams:

Classification made by:		Produces the error
(a)	Typical human	25%
(b)	Typical doctor (expert)	4%
(c)	Experienced doctor (expert)	1%
(d)	Team of experienced doctors (experts)	0.4%

Human-level error (d) is defined as the lowest possible error that might be achieved by any human team consisting of the best-experienced experts (here: doctors). We assume that nobody contemporarily can do it better!

Define the quality of the solution

Human-level performance shows the required quality of training data and reflects wisdom and experience of humanity to solve problems of a given kind.

- If the training data quality is poor or training data are contradictory, we cannot achieve better performance because nobody (event the team of human experts) can do the given task better.
- We can try to raise the quality of the training data to rise the human-level performance for a solved task to allow better training performance.
- Sometimes training data are described/defined by a too small subset of attributes that disallow to differentiate them enough (ambiguity producing contradictions) in the classification process. In this case, we should redefine the train set adding new attributes (new features) that would diversify the training examples good enough to discriminate them during the training.

Surpassing Human-level Performance

Generally, it is not easy to surpass human-level performance, especially for perception, speech, and image recognition problems.

Surpassing human-level performance is possible for many tasks:

- Product recommendations,
- Online advertising,
- Predicting transit time in logistics,
- Loan approvals,
- Many big data problems where humans cannot analyse them,
- Non-natural perception tasks that were not evolved in humans over millions of years,
- Tasks defined by various structural data requiring complex comparisons and analyses.

Surpassing human-level performance occurs when the achieved training error (e.g. 0.3%) is less than the human-level error (e.g. 0.5%).

It may be difficult to establish how much the human-level performance might be surpassed and what would be:

the final performance that could be still achieved and
the final bias that could be still avoided.

Bias and Variance of Models and Generalization Impact

How to validate quality of a model and strive for a good generalization?

Model Validation and Generalization

When training a model, we are interested not only in its performance on the training data, but also on new (test) data, i.e. on data on which this model has not been trained because it determines the quality of generalization of the model.

Generalization is good when the gained knowledge about the data associations and transformations is correctly represented by the model. It defines the model acquired skills in dealing with new data, which is crucial in practical applications.

Therefore, we divide the data into training, validation, and testing examples to have separate datasets for training the model and validating it during the training process, and then for testing it on not previously seen data.

To do it correctly, we must be sure that testing data are not repeated in training and validation data, which may happen when we choose them randomly.

Otherwise, it would be enough for the model to remember the training data (like in databases), but then the model would be useless for new cases and practical applications where we require the best generalization possible.

So, let's look at how can we control the training and validation processes to strive for the best generalization and model usability in practice!

Bias, Avoidable Bias, and Variance

Avoidable bias is an error defined by the difference between the training error and the human-level error:

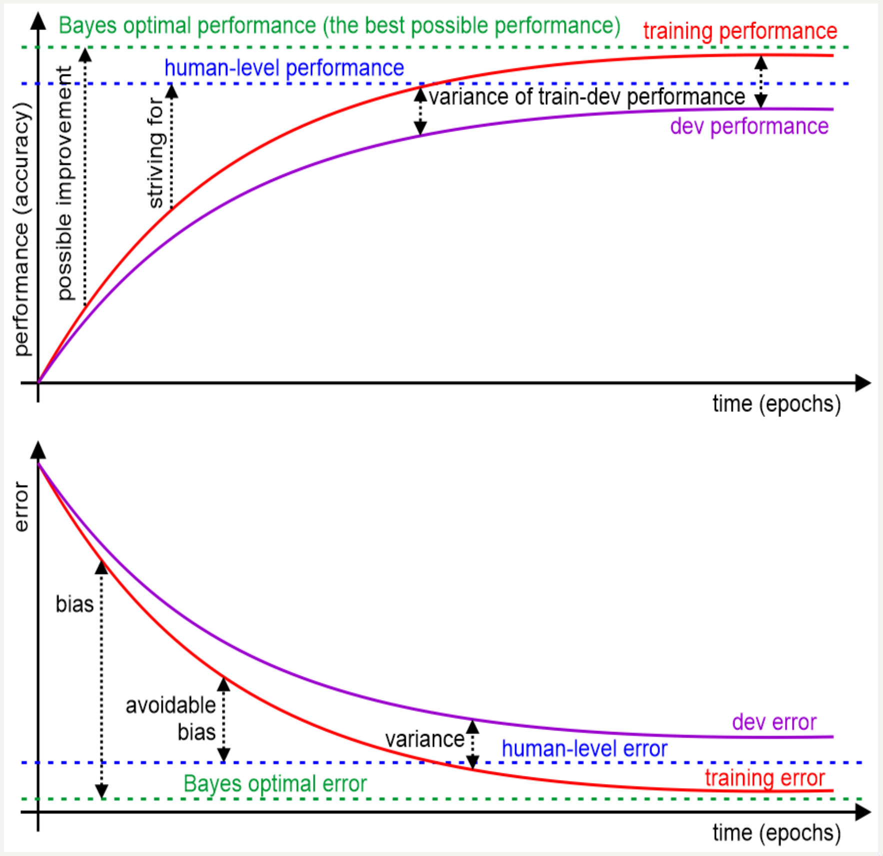
avoidable bias = training error – human-level error

Bias is an error defined by the difference between the training error and the Bayes error:

bias = training error – Bayes-level error

Variance is an error defined by the difference between the dev error and the training error:

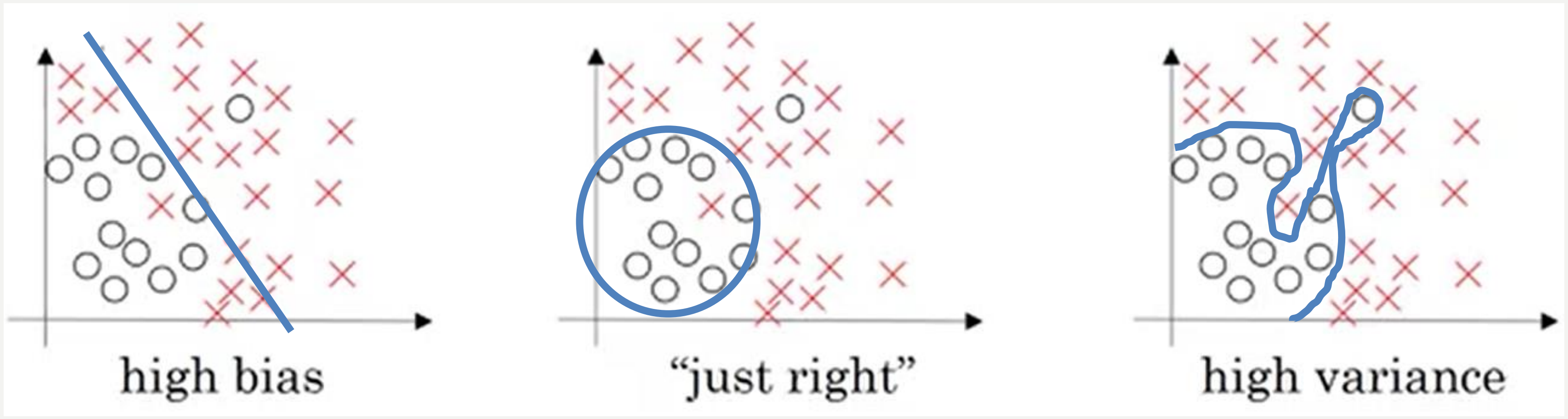
variance = dev error – training error



Bias and Variance of the Model

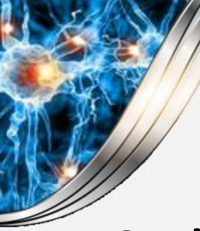
When adapting the parameters of the model, we can:

- not enough model the training dataset (underfitting)
- adjust the model too much, achieving poor generalization (overfitting)
- fit the dataset adequately, achieving good generalization (right fitting)



Example Results	Example 1	Example 2	Example 3	Example 4
Train set error	1%	12%	12%	1%
Dev set error	12%	13%	20%	2%
Bias	low	high	high	low
Variance	high	low	high	low
Overfitting	yes	no	yes	no
Underfitting	no	yes	yes	no

Dependently on high bias and/or high variance, we can try to adjust hyperparameters in the model to lower them appropriately and achieve better performance of the final model.



Error Analysis of Models

Analysis of the collected results and error levels allows us to look for a better solution by the implementation of some tips and tricks to improve the model performance.

Let's analyse a few examples with different levels of bias and variance:

Error Type	Model A	Model B	Model C	Model D
Bayes error*	0.5%	0.5%	0.5%	0.5%
Human-level error	0.7%	0.7%	0.7%	0.7%
Bias / Avoidable Bias	3.0% / 2.8%	3.0% / 2.8%	0.5% / 0.3%	0.3% / 0.1%
Training error	3.5%	3.5%	1.0%	0.8%
Variance	4.3%	0.8%	4.0%	0.2%
Dev error	7.8%	4.0%	5.0%	1.0%
Conclusion:	**High Bias** **High Variance** **First focus on Bias**	**High Bias** **Low Variance** **Focus on Bias**	**Low Bias** **High Variance** **Focus on Variance**	**Low Bias & Variance** **Quite well-trained model**

* Bayes error is not always known because the human-level abilities many times disallow to determine it. It might be experimentally determined or known due to the constructed training data set.

Tips and methodology:

We should not try to decrease variance until the avoidable bias is still high. Therefore, first, decrease bias and when it is low enough, start decreasing variance.

Minimizing Bias and Variance

During the model development, its training and tuning, we should try to minimize its bias and variance.

Bias tells us about the ability of the model to adapt to the training data, so if it is high, the model underfits (underfitting problem).

- Bias is the basic evaluation of the quality of the constructed model.
- If the bias is high, we should reconstruct the model, change its type, architecture or hyperparameters, and use bias-decreasing methods to minimize it as far as possible.

Variance defines the generalization possibilities of the model that tells us how well it can generalize about training data, dealing with a dev set and probably also with a test set and real-world data.

- If the variance is high, the constructed model is useless because the main goal of machine learning is to achieve good generalization, which allows us to use the trained models to real-world data with high confidence of correct predictions.
- Hence, we should reconstruct the model, change its type, architecture, or hyperparameters, and use variance-decreasing methods to achieve smaller variance.

Tackling with high bias and variance

What are the guidelines for minimizing errors and improving the performance of the model when the quality indicators (like bias and variance) of the model are low?

High Bias = training error is high	High Variance = validation/dev error is high
Bigger or different network architecture (insufficient number of parameters to adapt the model)	Smaller or different network architecture (less parameters avoid overfitting = learning by heart)
Training data do not allow for discrimination of the different classes or predictions (enrich training data using additional data attributes, describe the solved problem with new features that can help discriminate, clarify or highlight vital differences)	Training or validation data are not taken from all data distributions, i.e. they are unrepresentative for a solved problem (collect more data and shuffle them and recreate train and dev sets from all data distributions)
Smarter initialization, e.g. Xavier's one	Different training algorithm or network type
Use different Optimization algorithms (e.g. Adam, RMSprop, momentum)	Use different Regularization methods (L1, L2, dropout)
Train longer (more epochs, adapt training rate)	Early stopping when dev error starts rising
Redefine cost function (to better define the goal)	Redefine cost function (to better define the goal)
Data Augmentation and Strengthening (balance the number of representatives, augment less numerous ones or strengthen the impact of them)	Data Augmentation and Strengthening (balance the number of representatives, augment less numerous ones or strengthen the impact of them)
Transfer Learning and Freezing of Layers	Transfer Learning and Freezing of Layers
Adapt hyperparameters (manually or automatically using evolutional or grid approaches, e.g. mini-batch size)	Adapt hyperparameters (manually or automatically using evolutional or grid approaches, e.g. mini-batch size)

Regularization

How not to let the network overfitting

and control its validation performance?

Regularization Factor “Weight decay”

Regularization can be implemented by using an additional regularization factor and parameter λ to the loss function that penalizes the model for the weight growth that usually causes overfitting and is responsible for poor generalization:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(a^{(i)}, y^{(i)}\right) + \frac{\lambda}{2\cdot m}\cdot\sum_{i=1}^{m}\left\|w^{[l]}\right\|_F^2$$

where $\|\ldots\|_F$ means the Frobenius norm:

$$\left\|w^{[l]}\right\|_F = \sqrt{\sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}\left(w_{i,j}^{[l]}\right)^2}$$

$$w^{[l]} := w^{[l]} - \alpha\cdot dJw^{[l]} - \frac{\alpha\cdot\lambda}{m}\cdot w^{[l]} = w^{[l]} - \alpha\frac{\partial J\left(w^{[l]}, b\right)}{\partial w^{[l]}} - \frac{\alpha\cdot\lambda}{m}\cdot w^{[l]}$$

$$dJw^{[l]} = \frac{\partial J\left(w^{[l]}, b\right)}{\partial w^{[l]}} = \frac{1}{m}X\cdot dJZ^T + \frac{\lambda}{m}\cdot w^{[l]}$$

This kind of regularization is often called the “weight decay” because it prevents weight growth and the creation of too complex functions.

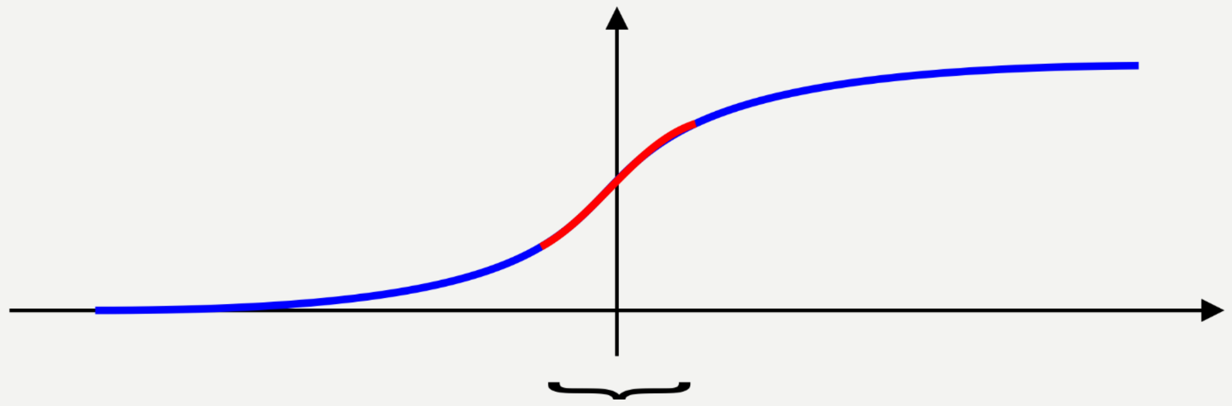
Regularization prevents overfitting

Regularization penalizes the weight matrices to be too large thanks to this extra regularization factor:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(a^{(i)}, y^{(i)}\right) + \frac{\lambda}{2\cdot m}\cdot\sum_{i=1}^{m}\left\|w^{[l]}\right\|_F^2$$

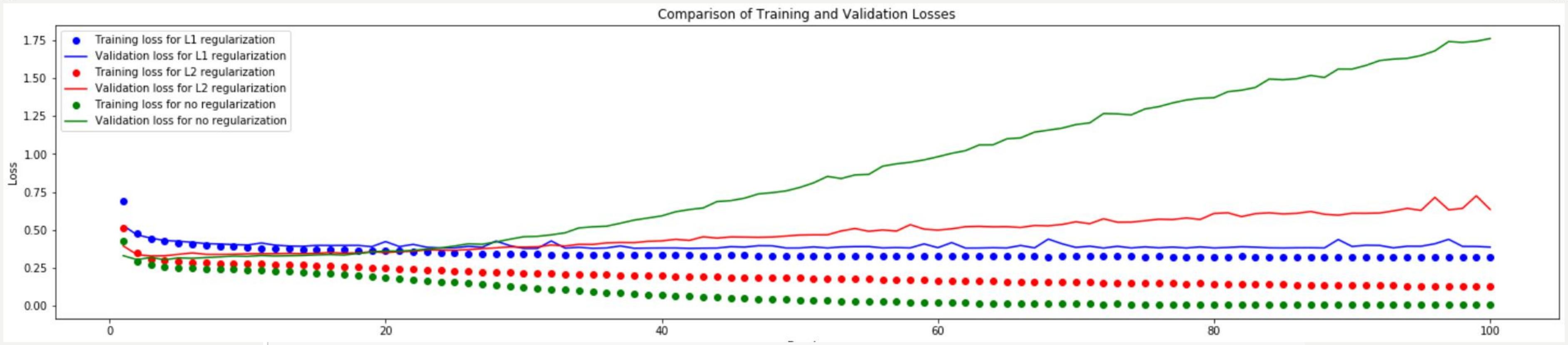
because we want to minimize the above cost function during the training!

If the weights are small, the output values of the activation functions of the neurons will also be not exceeding the middle, almost linear part of the activation function, so in case the activation function will be nearly linear, preventing the overfitting of the model:



Regularization L1 and L2

Examples of models with implemented L1 and/or L2 regularizations:



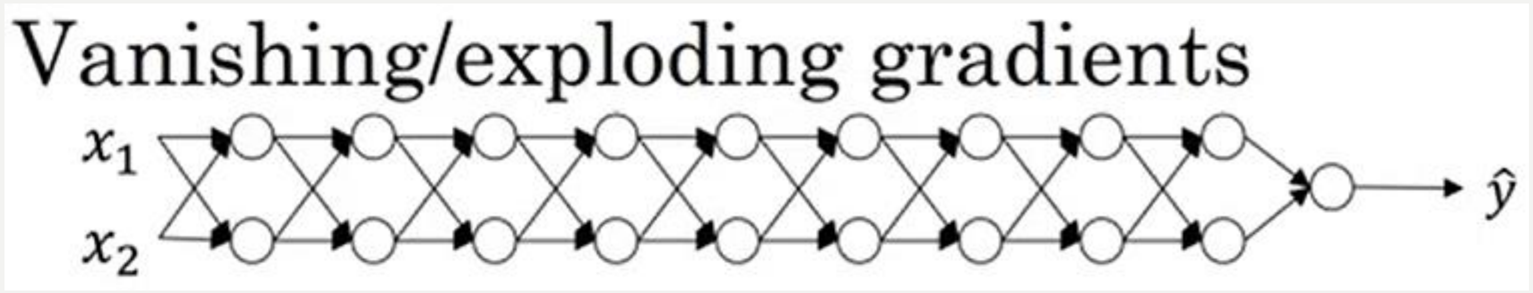
```
def Create3LModelL1(num_top_words, hl1 = 8, hl2 = 4):
    model = models.Sequential()
    model.add(layers.Dense(hl1, kernel_regularizer=regularizers.l1(0.001),
                           activation='relu', input_shape=(num_top_words,)))
    model.add(layers.Dense(hl2, kernel_regularizer=regularizers.l1(0.001),
                           activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))
    return model

def Create3LModelL2(num_top_words, hl1 = 8, hl2 = 4):
    model = models.Sequential()
    model.add(layers.Dense(hl1, kernel_regularizer=regularizers.l2(0.001),
                           activation='relu', input_shape=(num_top_words,)))
    model.add(layers.Dense(hl2, kernel_regularizer=regularizers.l2(0.001),
                           activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))
    return model

def Create3LModelL1L2(num_top_words, hl1 = 8, hl2 = 4):
    model = models.Sequential()
    model.add(layers.Dense(hl1, kernel_regularizer=regularizers.l1_l2(l1=0.001, l2=0.001),
                           activation='relu', input_shape=(num_top_words,)))
    model.add(layers.Dense(hl2, kernel_regularizer=regularizers.l1_l2(l1=0.001, l2=0.001),
                           activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))
    return model
```

Vanishing and Exploding Gradients

Regularization, inception blocks, and skip connections (like those used in ResNets) also prevent vanishing and exploding gradients.

In deep structures, computed gradients in previous layers are:

- **smaller and smaller (vanish) when small values (lower than 1) are multiplied/squared;**
- **greater and greater (explode) when big values (greater than 1) are multiplied/squared.**



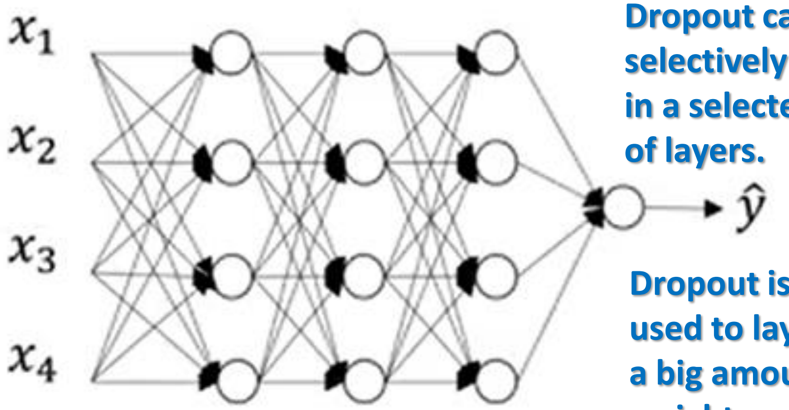
because today we use deep neural networks that consist of tens of layers!

We can fix exploding gradients by applying gradient clipping; which places a predefined threshold on the gradients to prevent them from getting too large, and by doing this it doesn't change the direction of the gradients; it only changes its length.

One of the newest and most effective ways to resolve the vanishing gradients is with residual neural networks (ResNets), which use skip connections (also called shortcuts or residual connections).

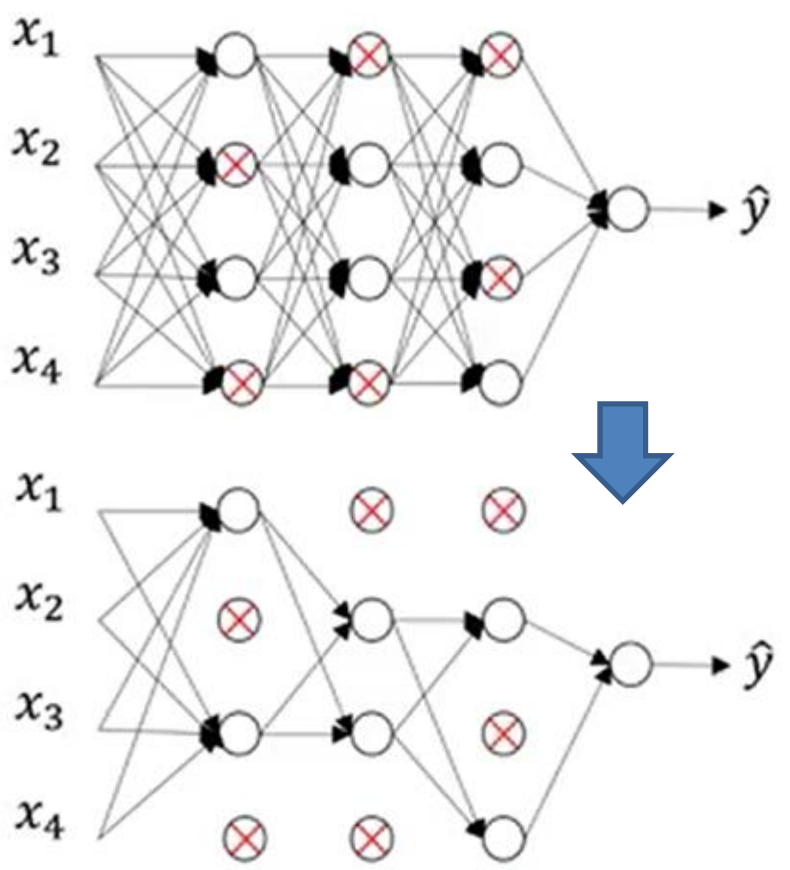
Dropout Regularization

Dropout regularization switches off some neurons with a given probability, not using them temporarily during propagation and backpropagation steps forcing the network to learn the same by various combinations of neurons in the network:



Dropout can be selectively used only in a selected subset of layers.

Dropout is usually used to layers with a big amount of weights and neurons.

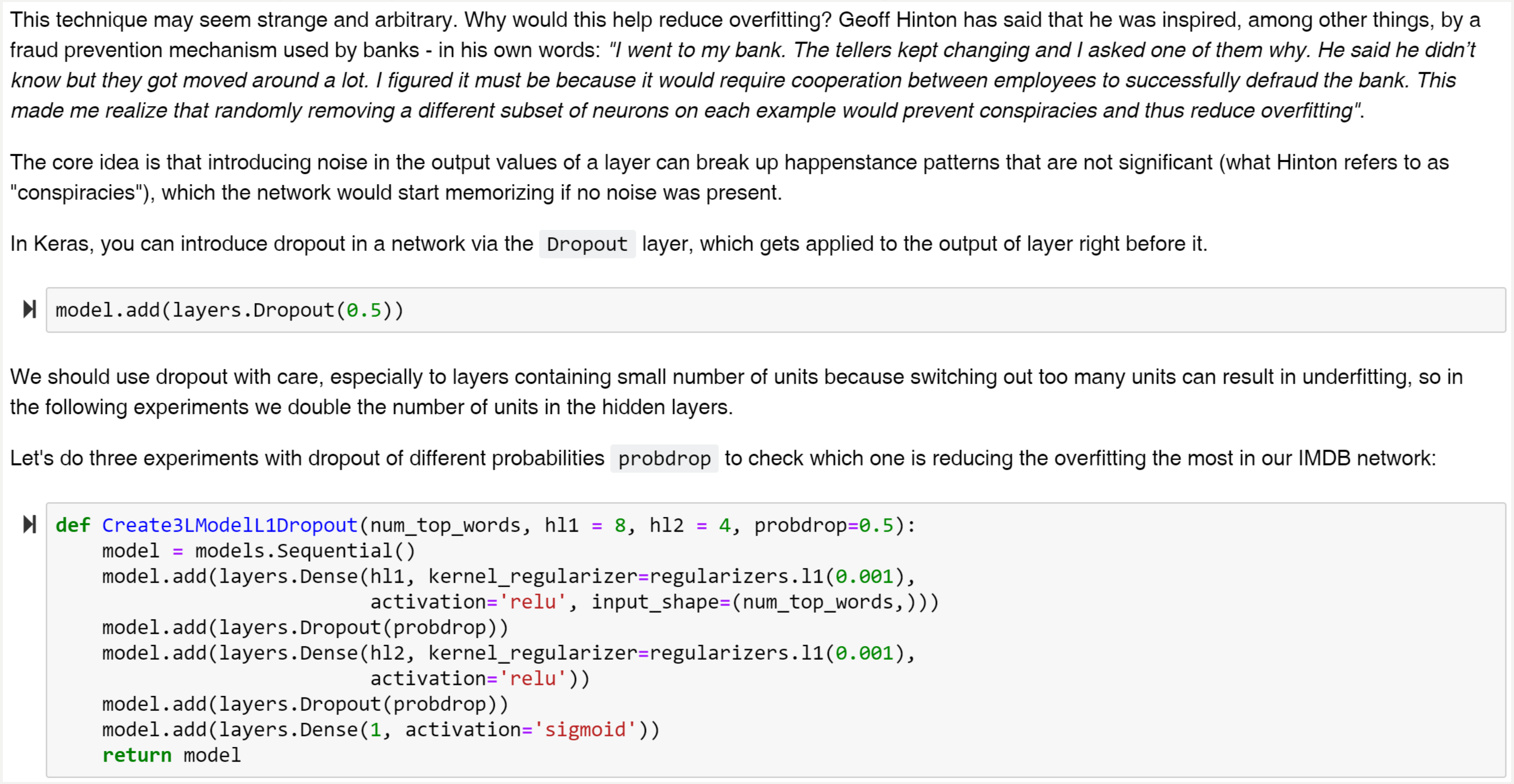


Implementing dropout regularization, the input stimuli of neurons are weakened according to the number of the shut-off neurons (i.e., the chosen probability of dropout on average, e.g. p = 0.25), so the stimulation must be higher to achieve the right stimulation of the neurons, e.g. the classification neurons in the last layer.

Dropout is one of the most popular regularization techniques for deep neural nets. Moreover, it usually speeds up the training.

Dropout Regularization

An example implementation of dropout regularization in a simple model:



This technique may seem strange and arbitrary. Why would this help reduce overfitting? Geoff Hinton has said that he was inspired, among other things, by a fraud prevention mechanism used by banks - in his own words: *"I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting"*.

The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that are not significant (what Hinton refers to as "conspiracies"), which the network would start memorizing if no noise was present.

In Keras, you can introduce dropout in a network via the `Dropout` layer, which gets applied to the output of layer right before it.

```python
model.add(layers.Dropout(0.5))
```

We should use dropout with care, especially to layers containing small number of units because switching out too many units can result in underfitting, so in the following experiments we double the number of units in the hidden layers.

Let's do three experiments with dropout of different probabilities `probdrop` to check which one is reducing the overfitting the most in our IMDB network:

```python
def Create3LModelL1Dropout(num_top_words, hl1 = 8, hl2 = 4, probdrop=0.5):
    model = models.Sequential()
    model.add(layers.Dense(hl1, kernel_regularizer=regularizers.l1(0.001),
                           activation='relu', input_shape=(num_top_words,)))
    model.add(layers.Dropout(probdrop))
    model.add(layers.Dense(hl2, kernel_regularizer=regularizers.l1(0.001),
                           activation='relu'))
    model.add(layers.Dropout(probdrop))
    model.add(layers.Dense(1, activation='sigmoid'))
    return model
```

As we can see, the dropout layers are put before the regularized layer.

Strengthening Influence of Examples

Sometimes training data do not represent training classes or features of samples equally representative, so we must perform some actions to balance the influence of different training examples to achieve the adequate impact (to their numerosity and/or representativeness of rare features) on the training process, then they can achieve the visible influence on the training process with different strengths:

- Modifying the error function in such a way that we add the strengthening factor $s^{(i)}$ for each training example (i = 1, …, m), we let them impact the training process with different strengths $s^{(i)}$:
- $J(w,b) = \frac{1}{\sum_{i=1}^{m} s^{(i)}} \sum_{i=1}^{m} s^{(i)} \cdot L\left(a^{(i)}, y^{(i)}\right)$
- In this way, we can avoid some unwanted classifications of the examples that cannot self-affect the training process because of their less numerously represented features in the training dataset.

Data attributes can also have different values for the training process (e.g. we want the gender not to influence the training process so much as the other data attributes), so we can bind different strengths with different attributes, weakening these which should have reduced influence on the classification process and strengthening those which are especially important from the classification point of view.

Sometimes we even avoid using some attributes like race, gender, age, disabilities, health condition, political or religious affiliation etc. not to discriminate some groups of people or other objects because of the law or equal opportunities.

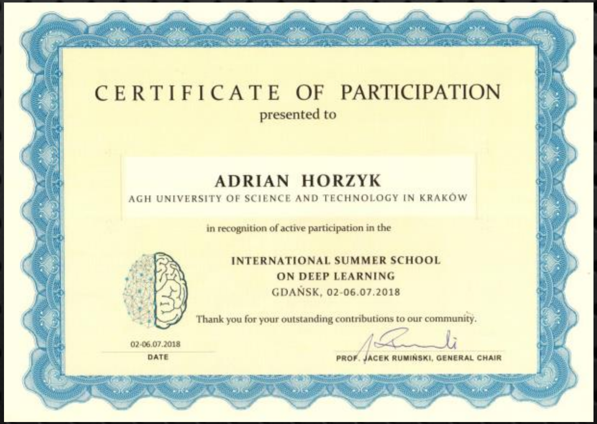
BIBLIOGRAPY

1. Francois Chollet, "Deep learning with Python", Manning Publications Co., 2018.
2. Ian Goodfellow, Yoshua Bengio, Aaron Courville, "Deep Learning", MIT Press, 2016, ISBN 978-1-59327-741-3.
3. Home page for this course: http://home.agh.edu.pl/~horzyk/lectures/ahdydci.php
4. Nikola K. Kasabov, Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence, In Springer Series on Bio- and Neurosystems, Vol 7., Springer, 2019.
5. Holk Cruse, Neural Networks as Cybernetic Systems, 2nd and revised edition
6. R. Rojas, Neural Networks, Springer-Verlag, Berlin, 1996.

8. Convolutional Neural Network (Stanford)
9. Visualizing and Understanding Convolutional Networks, Zeiler, Fergus, ECCV 2014.

BIBLIOGRAPY

10. IBM: https://www.ibm.com/developerworks/library/ba-data-becomes-knowledge-1/index.html
11. NVIDIA: https://developer.nvidia.com/discover/convolutional-neural-network
12. JUPYTER: https://jupyter.org/


CERTIFICATE OF PARTICIPATION
presented to

ADRIAN HORZYK
AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY IN KRAKÓW

in recognition of active participation in the

INTERNATIONAL SUMMER SCHOOL
ON DEEP LEARNING
GDAŃSK, 02-06.07.2018

Thank you for your outstanding contributions to our community.

02-06.07.2018
DATE

PROF. JACEK RUMIŃSKI, GENERAL CHAIR