UNIVERSIDAD POLITÉCNICA DE MADRID

Departamento Inteligencia Artificial

ETS DE INGENIEROS INFORMÁTICOS

Unit 1

Training Feedforward Neural Networks

Deep Learning

Professor

- Office: 2101
- Email: rvalle@fi.upm.es
- Personal web: https://bobetocalo.github.io/
- Department: https://dia.fi.upm.es/
- Research group: https://pcr-upm.github.io/

Departamento de
Inteligencia Artificial

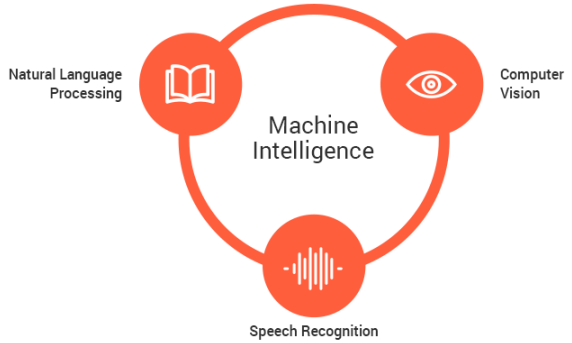
Grupo de Percepción
Computacional y Robótica

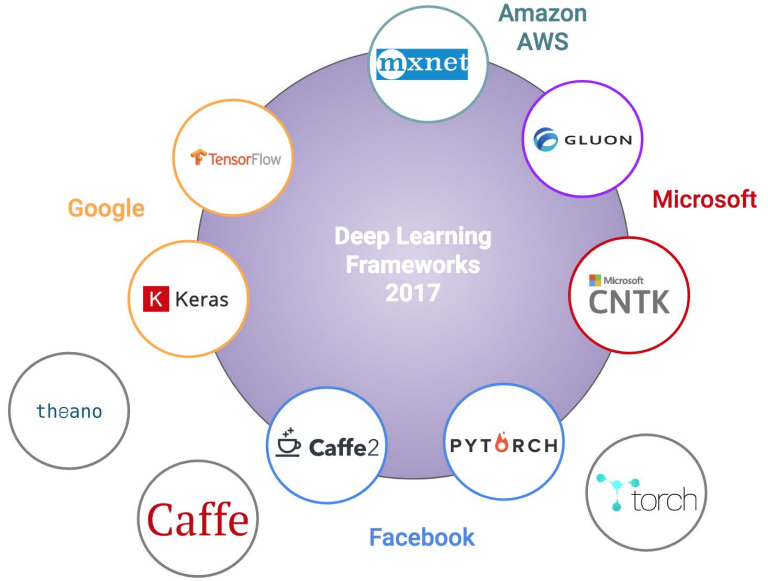
Contents

- *Introduction and motivation*
  - *Relevance*
  - *Historical overview*
  - *Feed-forward NNs notation*
- Model assessment
- Initialization and normalization
- Regularization layers

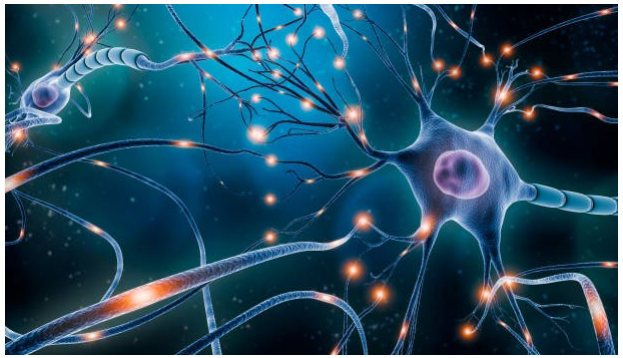
NNs introduction

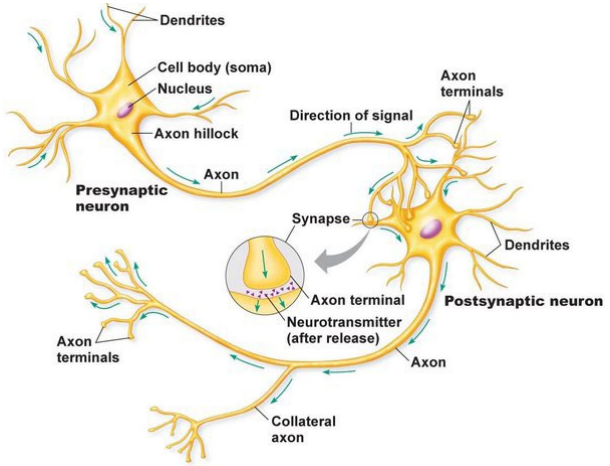
- Nowadays, **deep learning** is the state-of-the-art for:
  - *Computer vision*
  - *Speech recognition*
  - *Natural language processing*
- Widely used in industry (Google, Meta, IBM, ...). Why?
  - *Less feature engineering*
  - *Learn complex relationships between features in data*
  - *Applicable to wide spectrum of data and problems*
  - *Large dataset training*

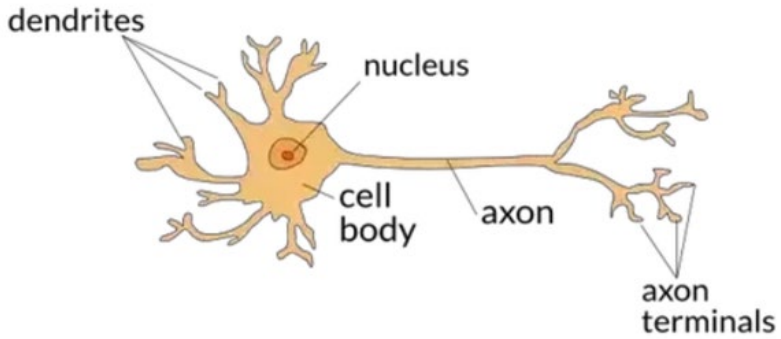




Biological motivation

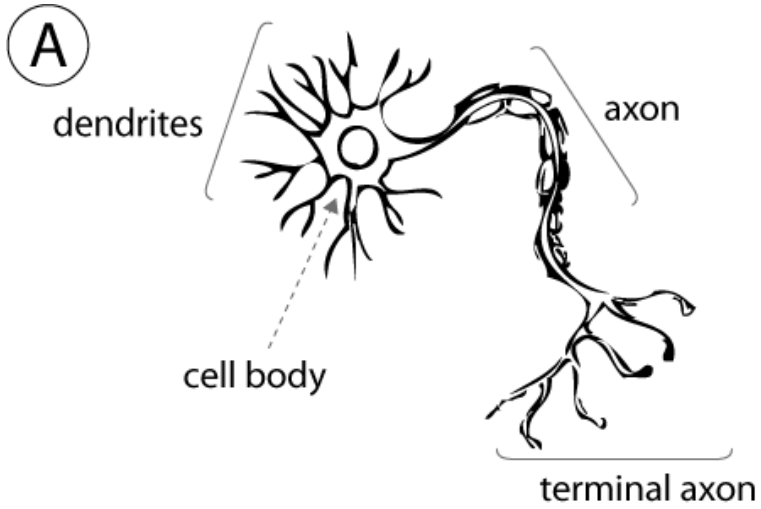


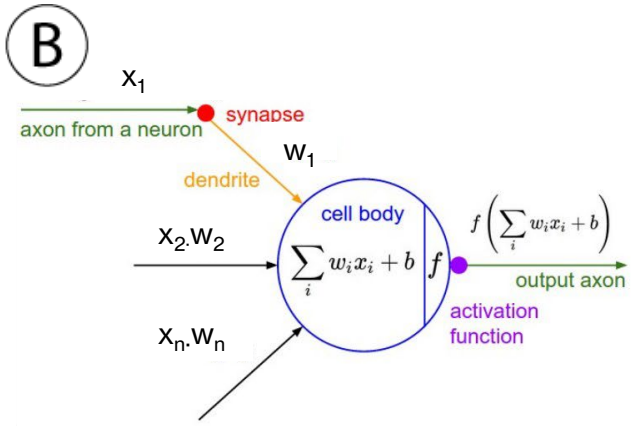




- Neurons are responsible for sending and receiving neurotransmitters/chemicals that carry information between brain cells
- Dendrites collect input from other neurons
- The neuron body adds these inputs to obtain an activation level
- The axon transmits, through synaptic terminals, the neuron output

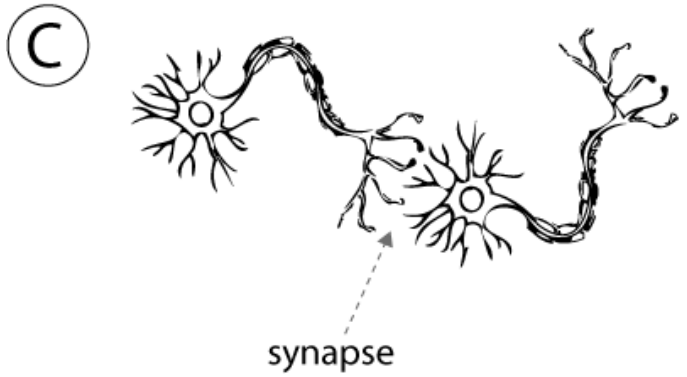
Biological motivation

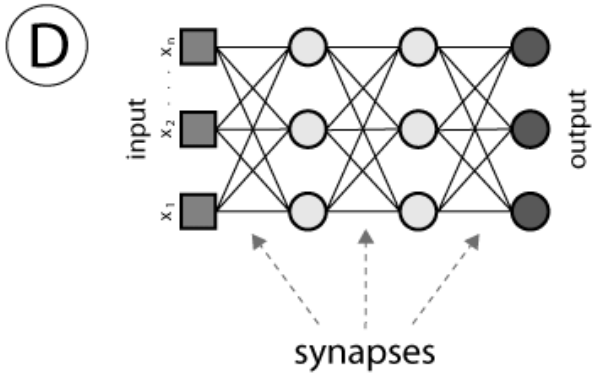




Artificial neuron = perceptron

- Neural network with one unit and activation function
- It learns a decision boundary that separates two classes using a line (hyperplane) in the feature space





- Each neuron receives input from other neurons
- The effect of each input is controlled by the synaptic weight
- Human brain contains 10^{11} neurons each with 10^4 synapses ...

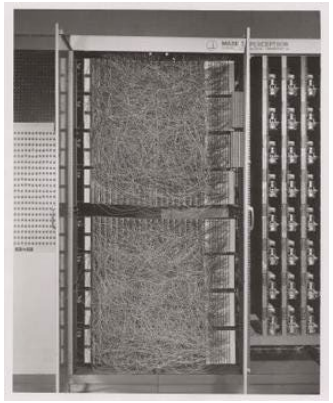
Historical overview

Deep learning is the “third emergence” of neural nets

Cybernetics (1950s–1960s)

- *1957: F. Rosenblatt, “Perceptron machine”*
- *1969: M. Minsky and S. Papert, “Perceptrons book”*

various limitations (linear models)



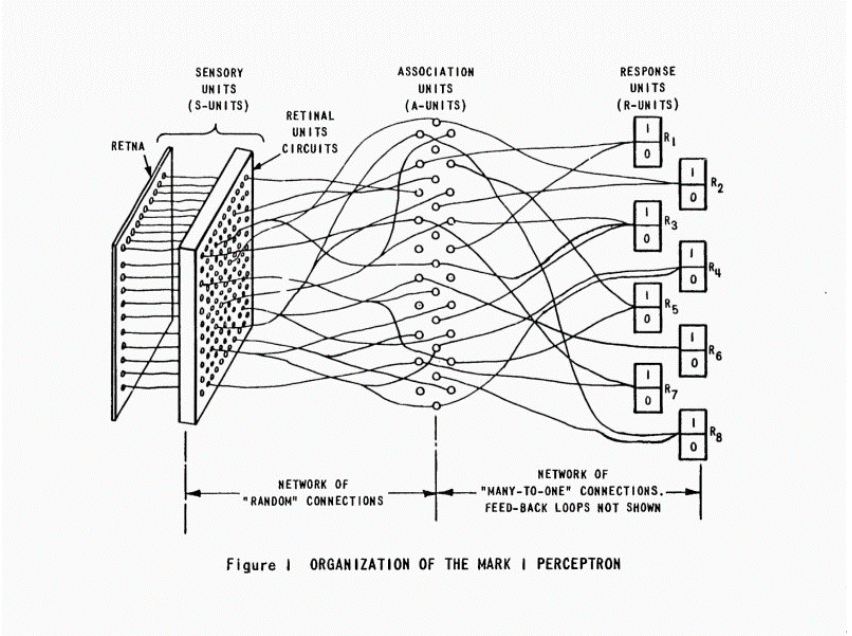


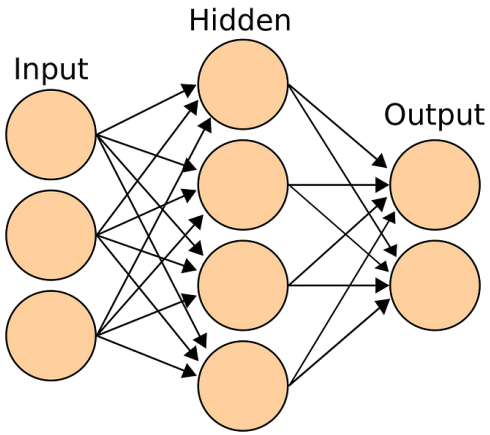
Figure 1 ORGANIZATION OF THE MARK I PERCEPTRON

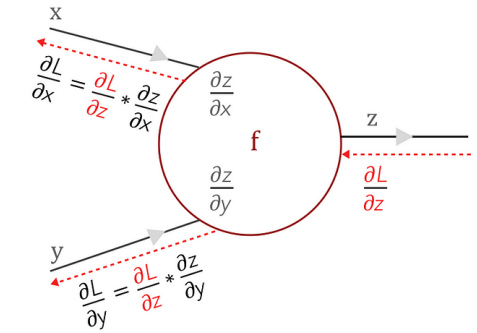
Historical overview

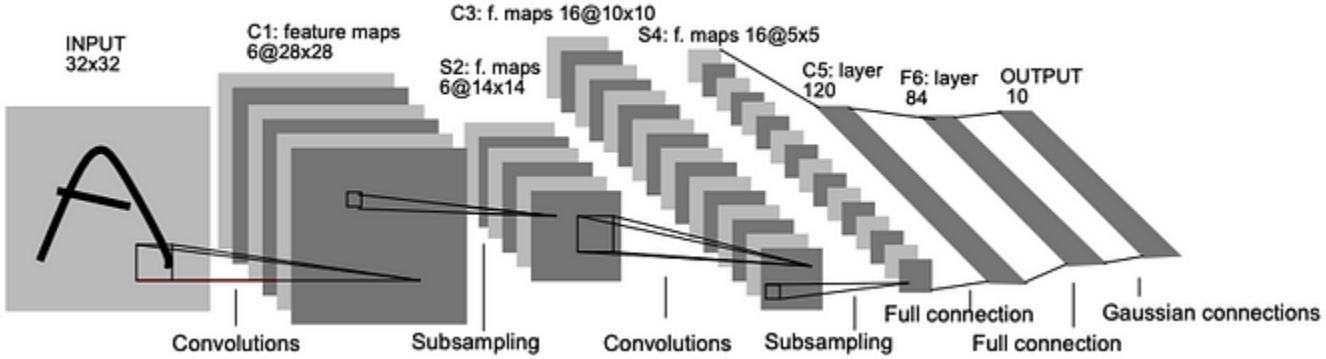
Deep learning is the “third emergence” of neural nets

Connectionism (1980s–1990s) *lack of computational resources for NNs*

- *1986: P. Werbos and G. Hinton and D. Rumerhart, “Backpropagation algorithm”*
- *1998: Y. LeCun, “Convolutional Neural Networks”*





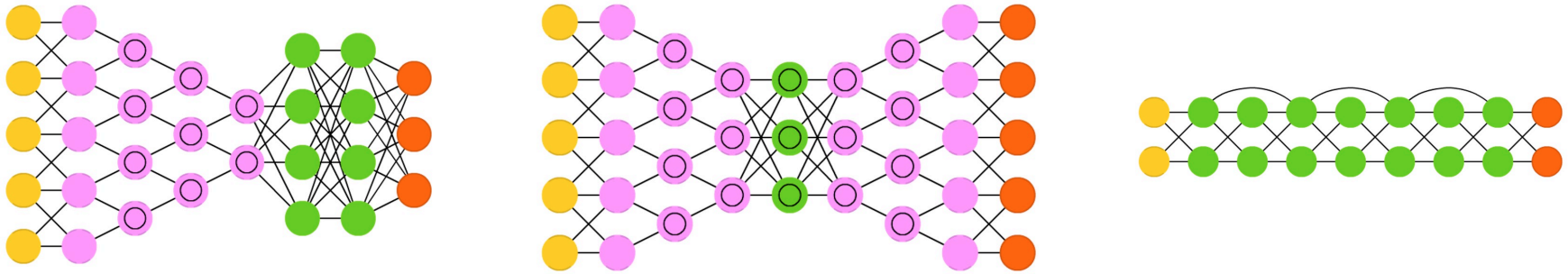


Historical overview

Deep learning is the “third emergence” of neural nets

Deep learning (2010s–today)

- *2012: A. Krizhevsky, “AlexNet”* *8 layers architecture + ReLU + dropout*
- *2015: K. He, “Deep Residual Networks”*
- *2017: A. Vaswani, “Transformers”*



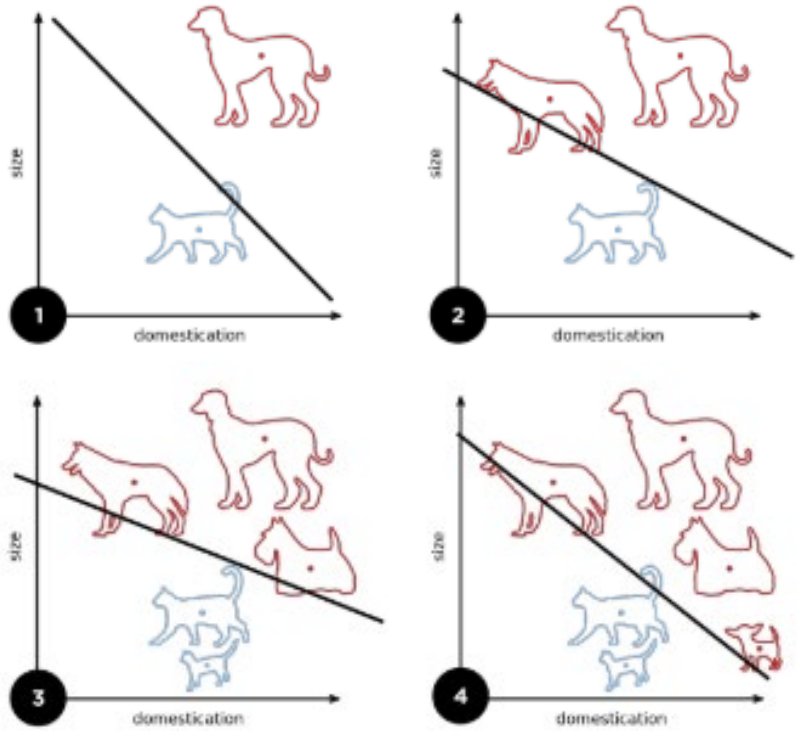
© https://www.asimovinstitute.org/category/deep-learning/

Perceptron

- Perceptron works by iteratively correcting its mistakes

Algorithm 1 Perceptron Algorithm

$w_0 \leftarrow 0$
for $t = 0 \ldots N-1$ **do**
  select $(x_t, y_t) \in \mathcal{D}$
  if $\text{sign}(\langle w_t, x_t \rangle) \neq y_t$ **then**
    $w_{t+1} \leftarrow w_t + y_t x_t$
  else
    $w_{t+1} \leftarrow w_t$
  end if
end for
return w_N



Single-layer neural network

- Single-layer neural network

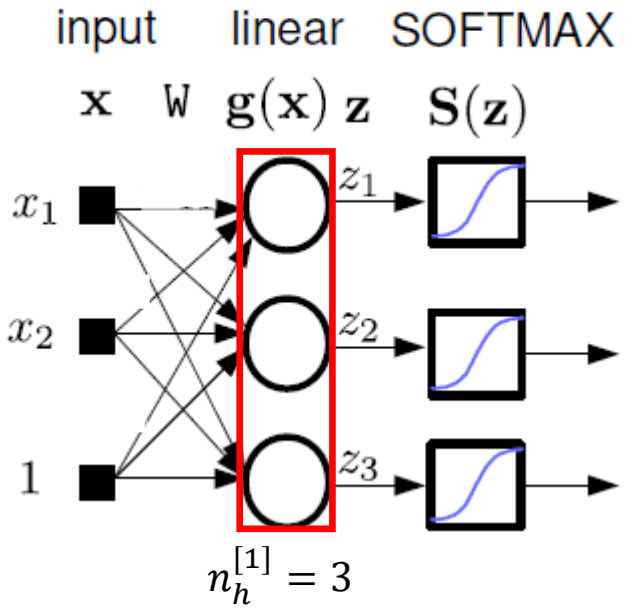
Response function

$$\mathbf{S}(\mathbf{g}(\mathbf{x})) = \mathbf{S}(\mathbf{W}\mathbf{x})$$

Training

$$\arg\min_{\mathbf{W}} \mathcal{L}(\mathbf{W}, \mathcal{D})$$

$n_x = 2$



$n_h^{[1]} = 3$

$n_y = 3$

$$\mathbf{S}(\mathbf{W}_1\mathbf{W}_2\mathbf{W}_3\mathbf{x}) = \mathbf{S}(\mathbf{W}'\mathbf{x})$$

n_x : input size

n_y : output size (or number of classes)

$n_h^{[l]}$: number of hidden units of the l^{th} layer

In the simplest case, all input units are connected to all output units

We call this a Fully Connected (FC) layer

Linear models:

- *Are very stable*
- *Only solve linearly separable problems*
- *Add more units?* ***useless !***

© https://playground.tensorflow.org/

Multi-layer neural network

- The construction of a ffNN is relatively straightforward

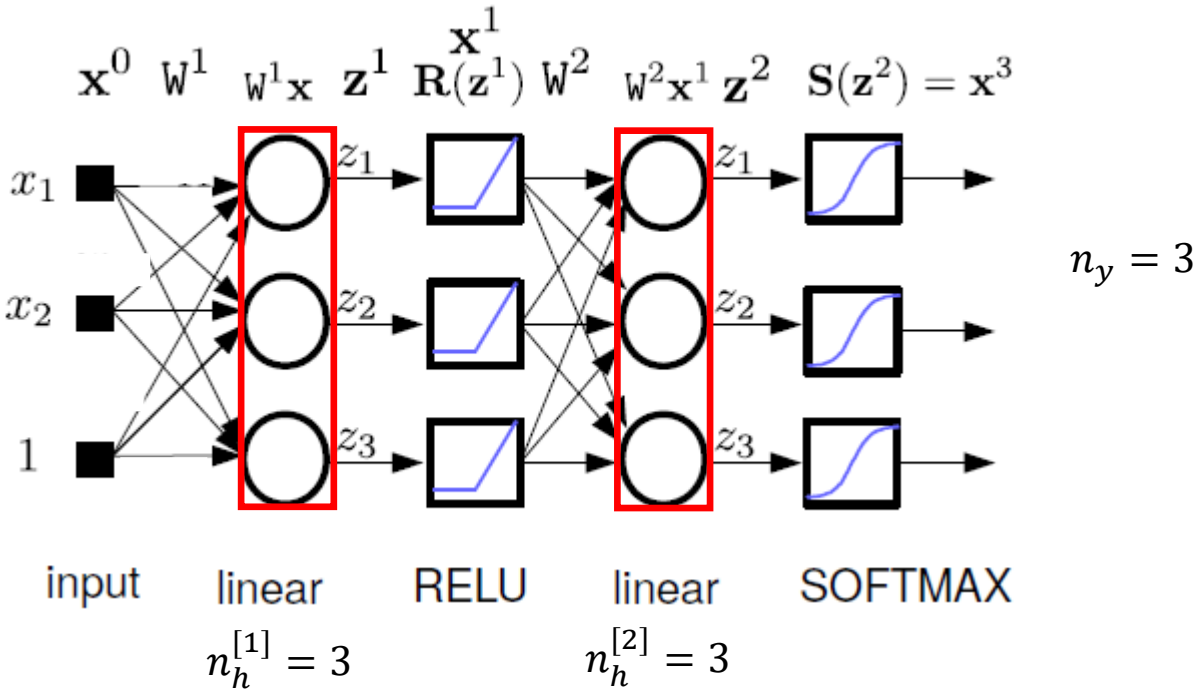
Response function

$$\mathbf{S}(\mathtt{W}^2\mathbf{R}(\mathtt{W}^1\mathbf{x}))$$

Training

$$\arg\min_{[\mathtt{W}]} \mathcal{L}([\mathtt{W}], \mathcal{D})$$

where $[\mathtt{W}] = [\mathtt{W}_1 \dots \mathtt{W}_s]$



A multi-layer perceptron consists of +1 FC layers

© https://playground.tensorflow.org/

ffNN models:

- *Solve non-linearly separable problems*
- *Hierarchical representation*
- *How do we train it?*

Example feed forward step

- Superscript $[l]$ denotes a quantity associated with the l^{th} layer.
  - Example: $a^{[L]}$ is the L^{th} layer activation. $W^{[L]}$ and $b^{[L]}$ are the L^{th} layer parameters.
- Superscript (i) denotes a quantity associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a_i^{[l]}$ denotes the i^{th} entry of the l^{th} layer's activations).

$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

$Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

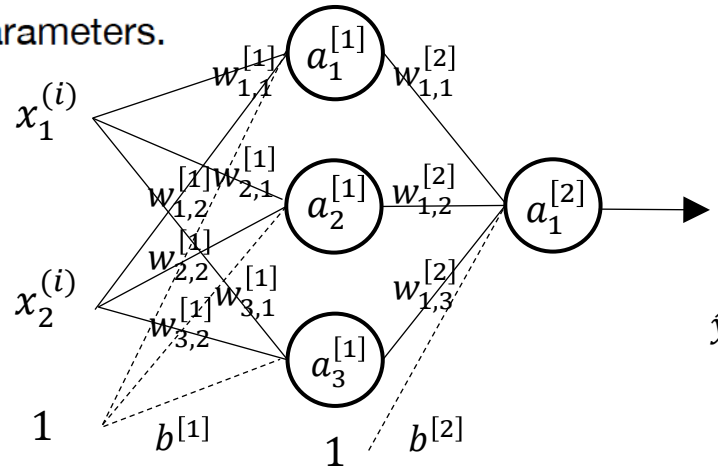
$y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

$W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$
weight matrix,superscript [l] indicates the layer

$b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

6 weights + 3 biases 3 weights + 1 bias loss function

$W^{[1]}$ $W^{[2]}$ $\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$



$\hat{y} = a^{[2]} = g(z^{[2]})$

$z^{[1]} = W^{[1]}x + b^{[1]}$ $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

$a^{[1]} = g(z^{[1]})$

activation function

$a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

Hidden layers activation functions

- *Non-linear component in a multi-layer NN required to learn arbitrarily complex non-linear functions. These are the most used ones:*
  - ***Saturating***

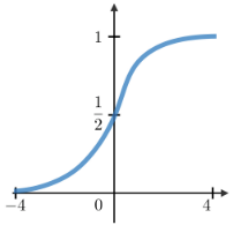
    used by initial NN models

    vanishing gradients

    poor convergence

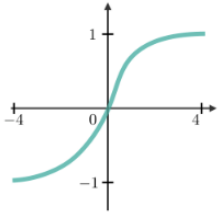
Sigmoid

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



tanh

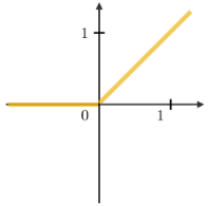
$$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$



  - ***Non-saturating***

ReLU

$$g(z) = \max(0,z)$$



new DL insights

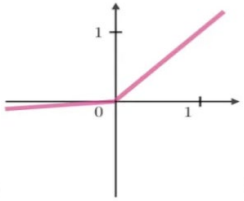
do not saturate for z > 0

dying ReLUs problem

- Leaky ReLUs never die

  parameter fixed, random or learned

$$g(z) = \max(\epsilon z, z) \quad \text{with } \epsilon \ll 1$$



- Exponential Linear Unit (ELU)

  everywhere smooth

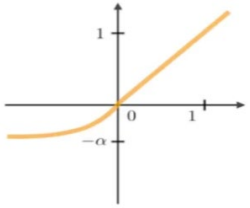
  nonzero gradient for z < 0

  average output closer to 0

  computationally expensive

ELU

$$g(z) = \max(\alpha(e^{z}-1), z) \quad \text{with } \alpha \ll 1$$



Output layer activation functions

- The **final activation layer** is special since it will work in tandem with the loss function $\mathcal{L}(\hat{\mathrm{y}}^{(\mathrm{i})}, \mathrm{y}^{(\mathrm{i})})$
- Classification problems: **Softmax**
  - Generalization of the sigmoid activation used for binary classification to c classes
  - It compresses the sum of the output vector to be 1

$$\mathbf{S}(\mathbf{z}_i) = \frac{e^{\mathbf{z}_i}}{\sum_{j=1}^{c} e^{z_i^j}} = \mathbf{S}(z) : \mathbb{R}^K \longmapsto \left\{ \mathbf{S} \in \mathbb{R}^K | S^i > 0, \sum_i S^i = 1 \right\}$$

- Regression problems: **Linear**

$$\mathbf{S}(\mathbf{z}_i) = \mathbf{z}_i$$

Activation functions in Keras

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

model = Sequential()
model.add(InputLayer(input_shape=INPUTS))
for neurons in n_neurons_per_hidden_layer:
    model.add(Dense(units=neurons, activation='relu'))
model.add(Dense(units=OUTPUTS, activation='softmax'))
```

Layer activations

- celu function
- elu function
- exponential function
- gelu function
- glu function
- hard_shrink function
- hard_sigmoid function
- hard_silu function
- hard_tanh function
- leaky_relu function
- linear function
- log_sigmoid function
- log_softmax function
- mish function
- relu function
- relu6 function
- selu function
- sigmoid function
- silu function
- softmax function
- soft_shrink function
- softplus function
- softsign function
- sparse_plus function
- sparsemax function
- squareplus function
- tanh function
- tanh_shrink function
- threshold function

© https://keras.io/api/layers/activations/

Loss functions

- *Ground-truth is "one-hot" encoding*

How do we code label values, $y_i \in \{c_1 \dots c_l\}$?

$$\mathbf{S}(\mathbf{z}_i) = \begin{bmatrix} 0.9 \\ \vdots \\ 0.03 \end{bmatrix} \in \mathbb{R}^l \quad \mathbf{y}_i = \begin{bmatrix} 1 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^l \Leftrightarrow y_i = c_1$$

$$\mathcal{L}(\mathbf{g}, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} d(\mathbf{S}(\mathbf{z}_i), \mathbf{y}_i)$$

- *How do we compare classifier responses with labels?*

*The **loss function** is the function we are trying to minimize to learn the relationship between the given inputs and the desired outputs.*

- *Classification error (CE)*
- *Average cross entropy (ACE)*

classification

$$\mathcal{L}(\mathbf{g}, \mathcal{D}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{l} y_i^j \log S^j(\mathbf{z}_i)$$

- *Mean squared error (MSE)*
- *Mean absolute error (MAE)*

regression

$$\mathcal{L}(\mathbf{g}, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} [y_i - \mathbf{g}(\mathbf{z}_i)]^2$$

Loss functions

Let us analyse one example:

CE: too coarse error measure
MSE: better than CE, but emphasizes errors less
ACE: good approach!

computed			targets			correct?
0.3	0.3	0.4	0	0	1 (democrat)	yes
0.3	0.4	0.3	0	1	0 (republican)	yes
0.1	0.2	0.7	1	0	0 (other)	no

CE = 0.33
MSE = 0.81
ACE = -2 log0.4 – log0.1 = 1.38

computed			targets			correct?
0.1	0.2	0.7	0	0	1 (democrat)	yes
0.1	0.7	0.2	0	1	0 (republican)	yes
0.3	0.4	0.3	1	0	0 (other)	no

CE= 0.33
MSE = 0.34
ACE = -2 log0.7 – log0.3 = 0.64

© https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/

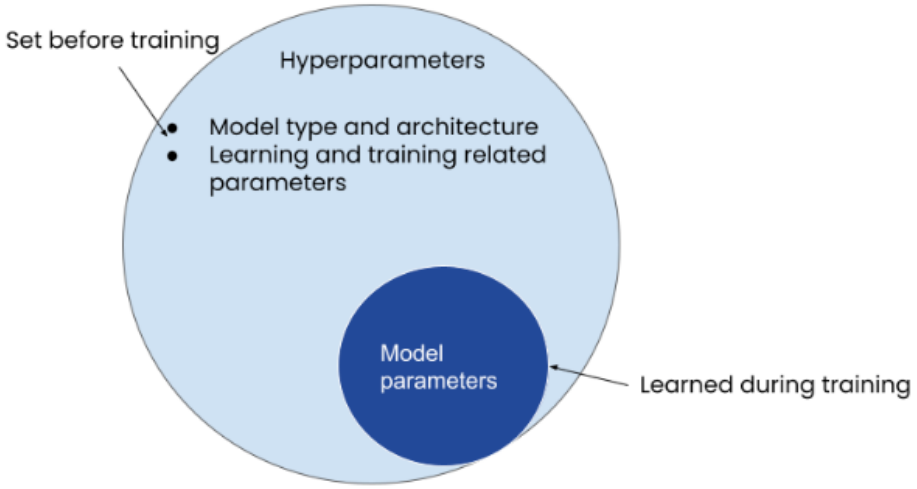
Parameters vs hyperparameters

- **Parameter:** Its value is obtained automatically by the training process.
  - Weights and biases: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}$ ...

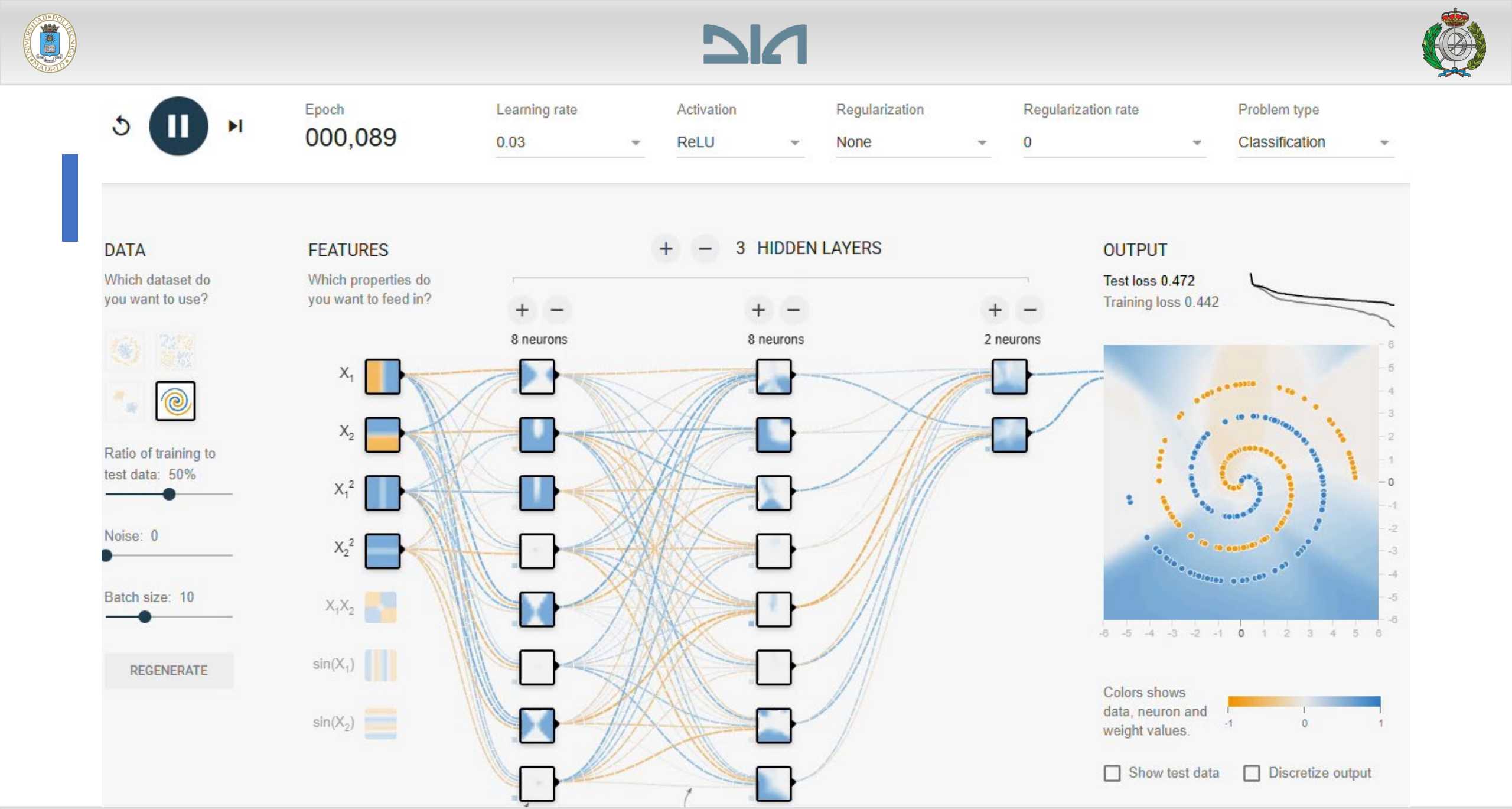
They are estimated by optimization algorithms (SGD, Adam, etc.)

- **Hyperparameter:** Its value is set manually before the training process begins.
  - Num. layers (L)
  - Num. units ($n_h^{[l]}$)
  - Activation function ($g(z)$)
  - Loss function ($\mathcal{L}(\hat{y}, y)$)
  - Number of epochs
  - Learning rate
  - Batch size

  ...



They are estimated by hyperparameter tuning


Epoch
000,089
Learning rate
0.03
Activation
ReLU
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
3 HIDDEN LAYERS
8 neurons
8 neurons
2 neurons
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
OUTPUT
Test loss 0.472
Training loss 0.442
Colors shows data, neuron and weight values.
Show test data
Discretize output

Contents

- Introduction and motivation
- *Model assessment*
  - *Training, validation and test sets*
  - *Evaluation metrics*
  - *Bias and variance trade-off*
- Initialization and normalization
- Regularization layers

Feature engineering

- Feature engineering enables you to build more complex models than you could with only raw data.
- You can think of feature engineering as helping the model to understand the data set in the same way you do.

If you had to prioritize improving one of the areas below in your machine learning project, which would have the most impact?
Using the latest optimization algorithm
The quality and size of your data
A deeper network
A more clever loss function

© https://developers.google.com/machine-learning/data-prep

Feature engineering

If you had to prioritize improving one of the areas below in your machine learning project, which would have the most impact?

Using the latest optimization algorithm

The quality and size of your data ✓

Data trumps all. It's true that updating your learning algorithm or model architecture will let you learn different types of patterns, but if your data is bad, you will end up building functions that fit the wrong thing. The quality and size of the data set matters much more than which shiny algorithm you use.

Correct answer.

A deeper network

A more clever loss function

© https://developers.google.com/machine-learning/data-prep

Data preprocessing and preparation

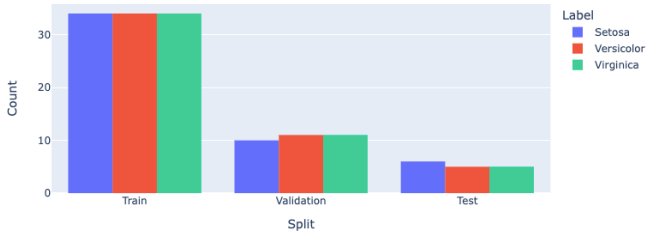
- The first step in getting your hands on machine learning is finding a suitable data set ready-made collection of images, audio, videos, texts, or tables for model training.
- Nowadays internet is full of free machine learning data sets.
- Data must be prepared before training:
  - Integrating different data sources.
  - Cleaning data.
  - Formatting data.
  - Visualizing data.
  - Dimensionality reduction.
  - Solving ethical issues (*e.g.,* anonymization).

UCI Machine Learning Repository

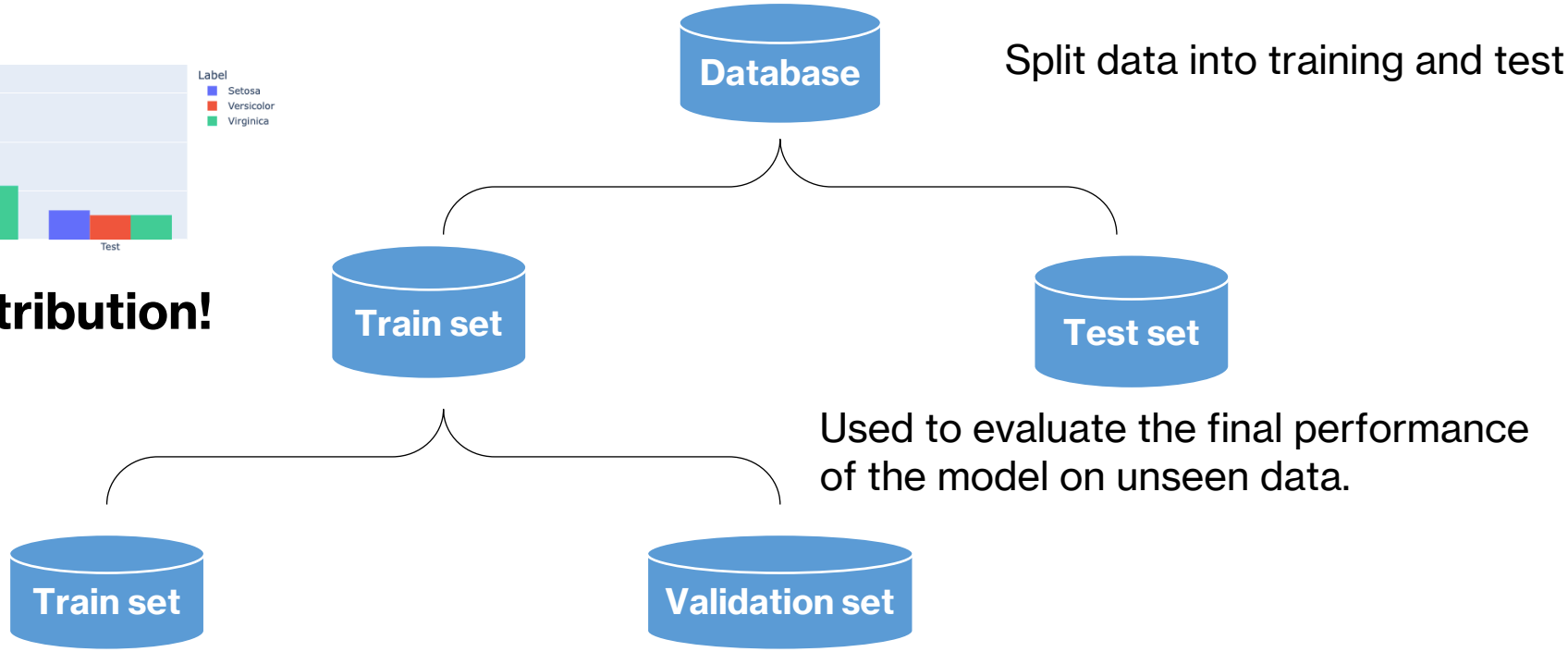
kaggle

OpenML

Divide the database into subsets



All same distribution!



Split data into training and test set

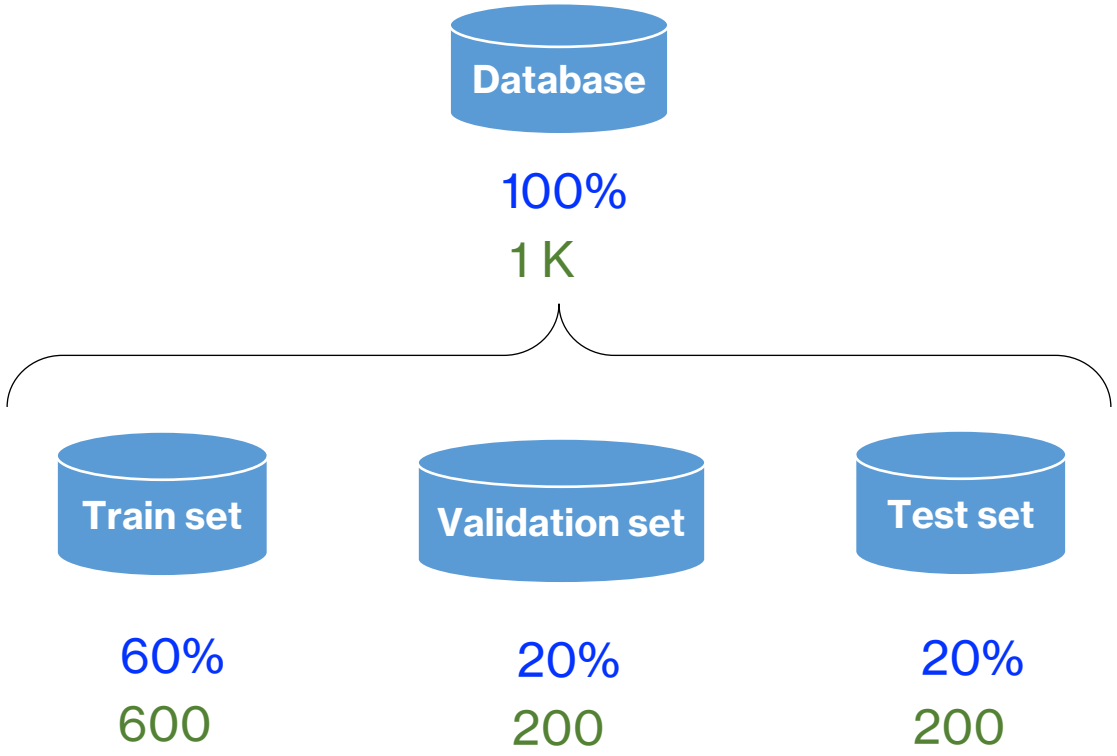
Used to evaluate the final performance of the model on unseen data.

Used to adjust the model's parameters (learning phase).

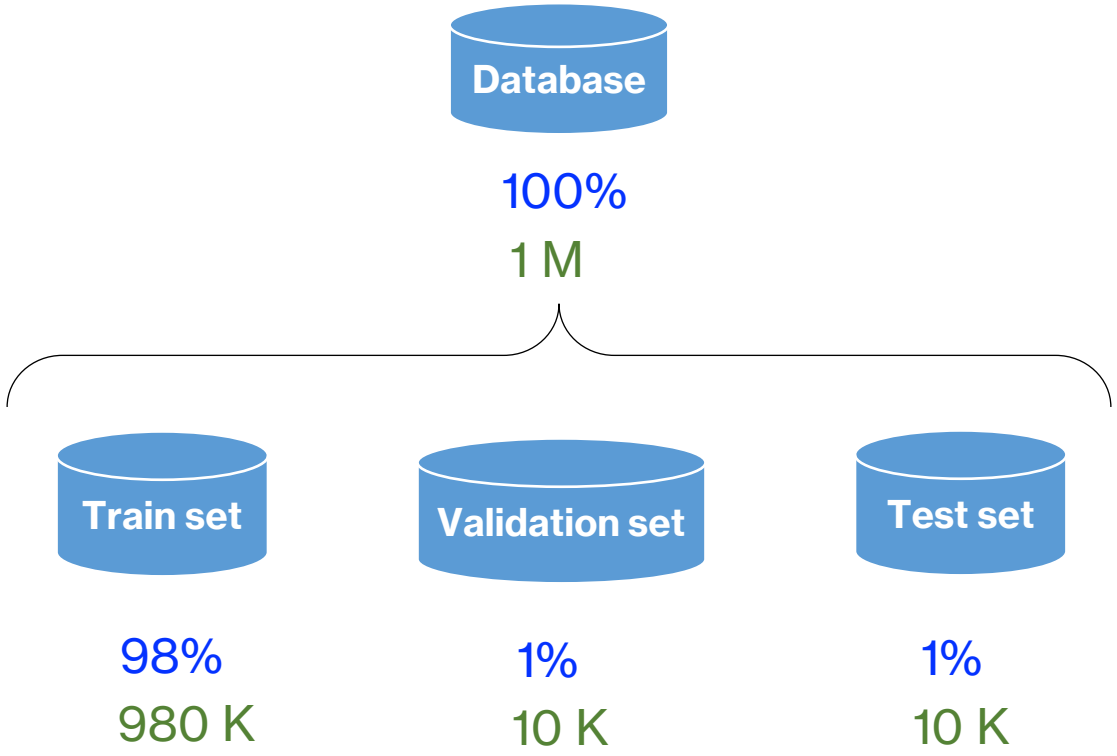
Used to fine-tune hyperparameters and prevent overfitting (model tuning).

© https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html

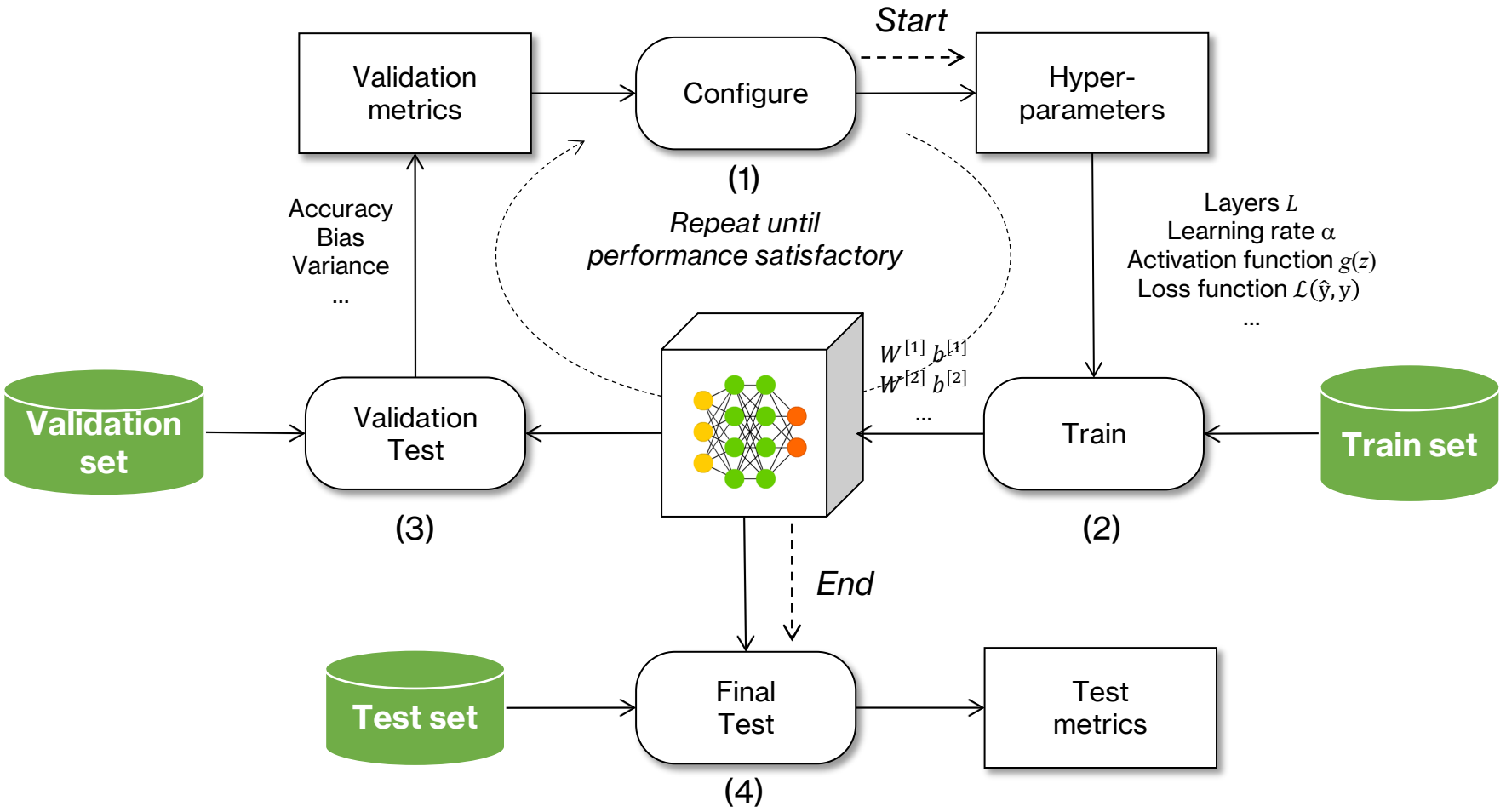
Example (60%, 20%, 20%)



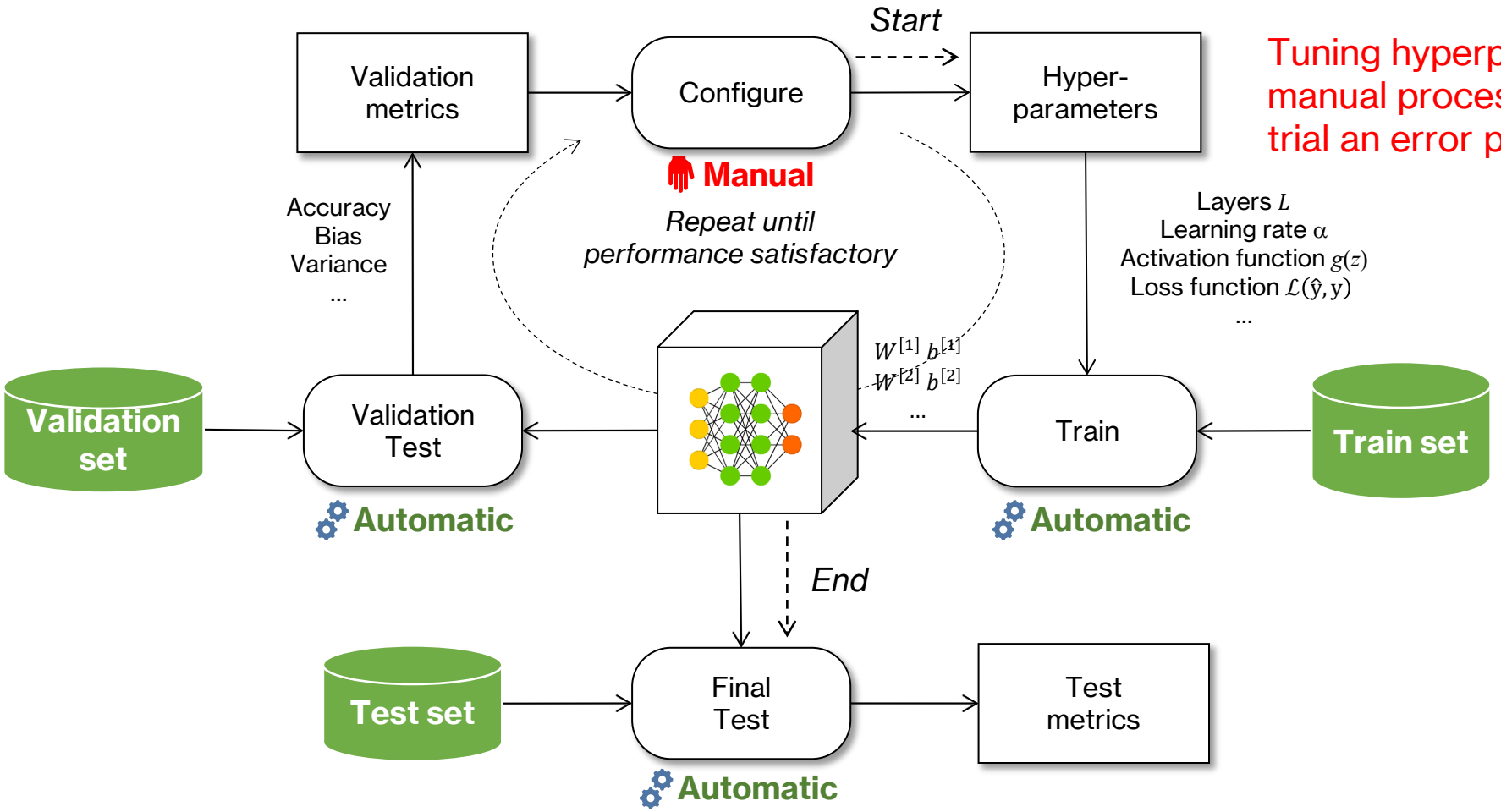
Example (98%, 1%, 1%)



Training Neural Networks



Tuning hyperparameters



Tuning hyperparameters is a manual process in an iterative trial an error procedure

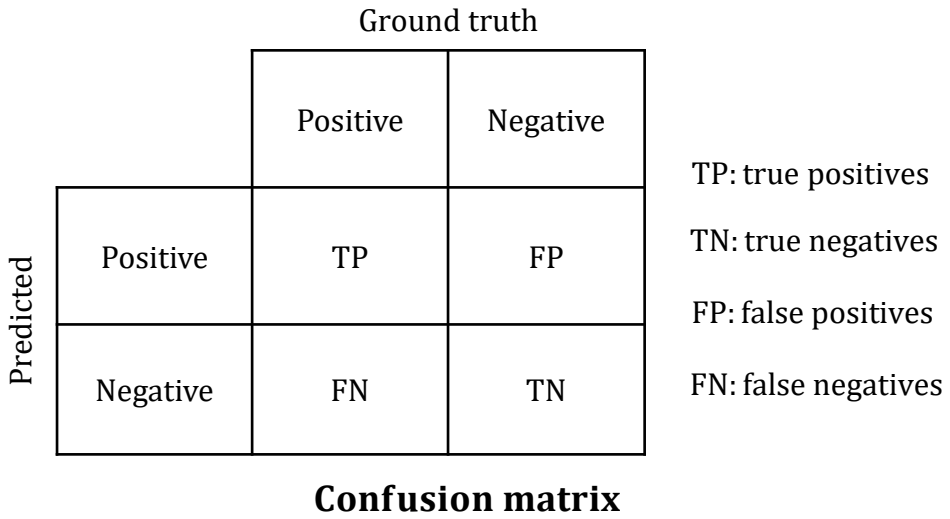
Evaluation metrics

- A metric is a function that is used to judge the performance of your model
- Metric functions are like loss functions, except that the results from evaluating a metric are not used when training the model

$$Accuracy = \frac{Correct\ predictions}{Number\ of\ examples} = \frac{TP + TN}{TP + TN + FP + FP}$$

$$Precision = \frac{TP}{TP + FP} \qquad Recall\ (sensitivity) = \frac{TP}{TP + FN}$$

$$Specifity = \frac{TN}{TN + FP}$$

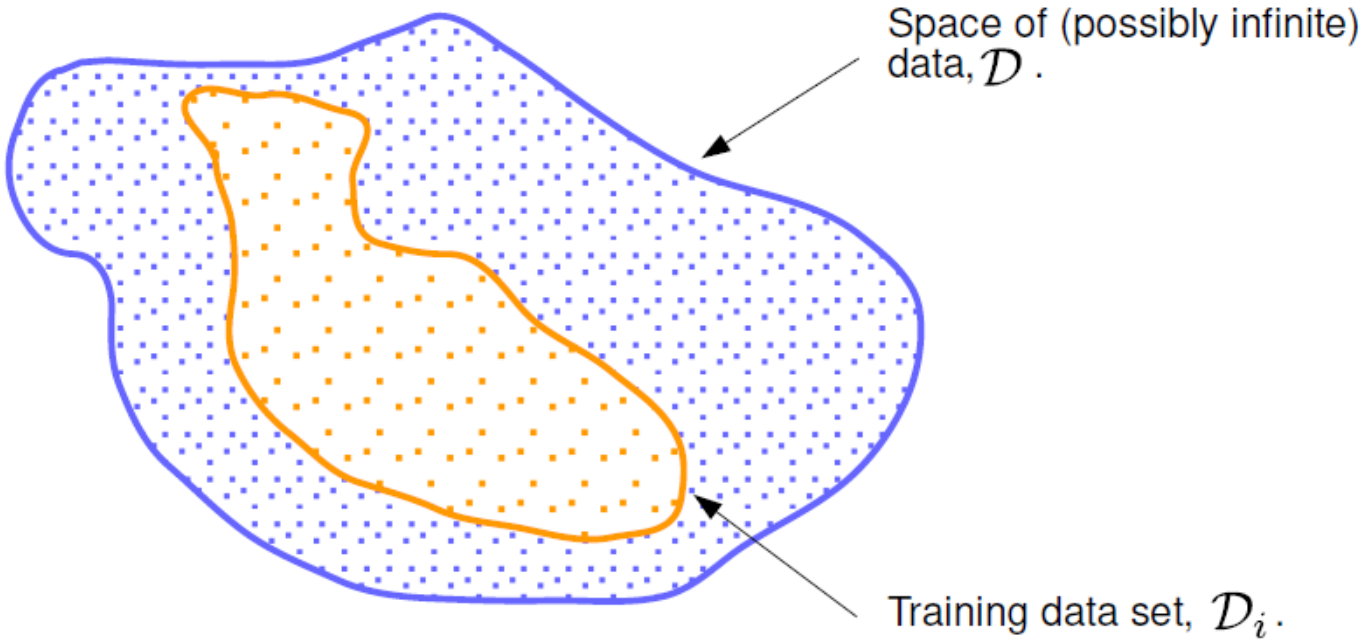



```
model.compile(optimizer='SGD', loss='binary_crossentropy', metrics=['binary_accuracy'])
```

© https://keras.io/api/metrics/

Understanding generalization error

- It emerges when you train a model with a **limited amount of data**
- The goal is to minimize the **generalization error**, *i.e.,* error on any data either seen (during training) or unseen



Bias and variance trade-off

$$Error = Bias^2 + Variance + Irreducible\_error$$

$$Error(x) = E\left[\left(Y - g(x)\right)^2\right] =$$

$$= (E[g(x)] - f(x))^2 + E\left[g(x) - E[g(x)]\right]^2 + \sigma^2 =$$

$$= Bias(g(x))^2 + Variance(g(x)) + \sigma^2$$

The prediction error for a machine learning algorithm can be expressed with the sum of three terms

$f(x)$: *true function*

$g(x)$: *predicted function using Di*

ϵ: *error normally distributed*, $\epsilon \sim N(0, \sigma)$

$Y = f(X) + \epsilon$

$E[x]$: *expected value (expectation)*

$E[g(x)]$: *average predicted value*

$Variance\left(g(x)\right) = E\left[g(x) - E[g(x)]\right]^2$

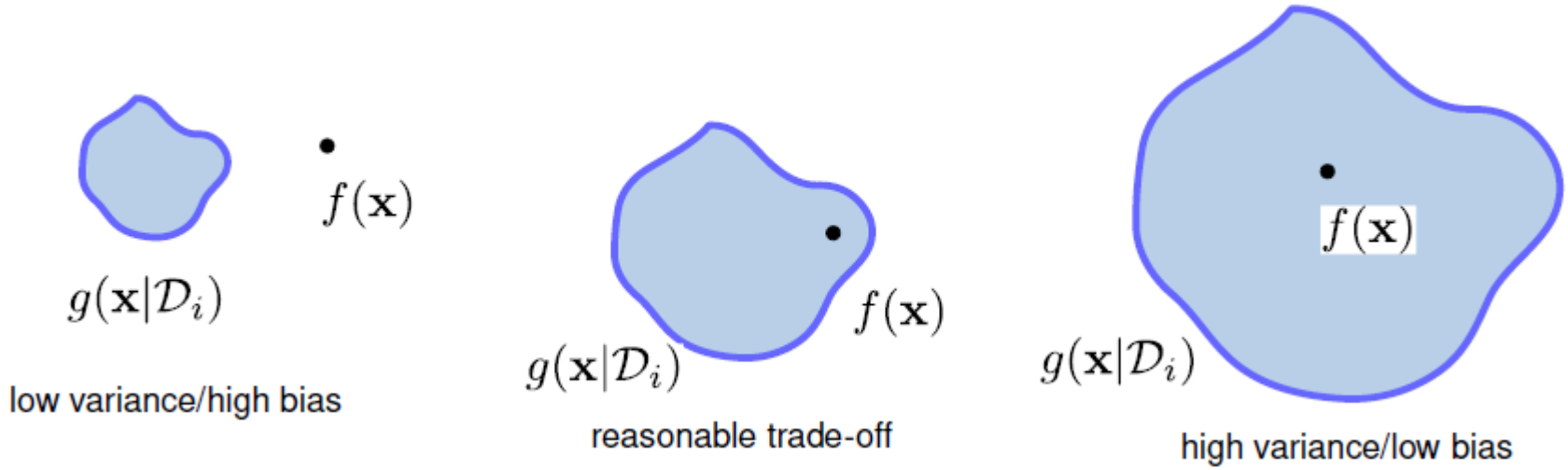
$Bias(g(x)) = E[g(x) - f(x)]$

σ^2: *irreducible error*

© Trevor Hastie et al. (2009). The Elements of Statistical Learning. Springer Series in Statistics.

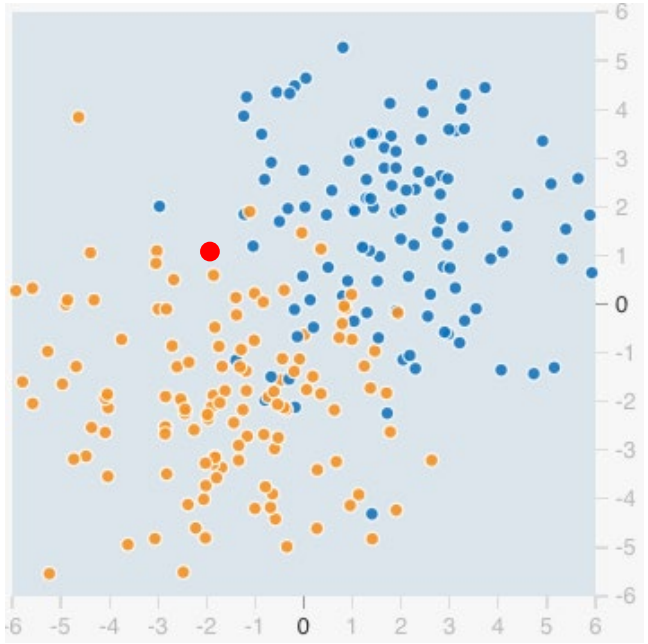
Bias and variance trade-off

- **Bias:** Distance from the average model to the solution.
  It is big if the model has too little capacity to fit the data.
  Warning: There are two different meanings for the word “bias”. Bias term b in: $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$
- **Variance:** Specificity, changes in the model for different training data sets.
  It is big if the model has so much capacity that it also fits the noise.



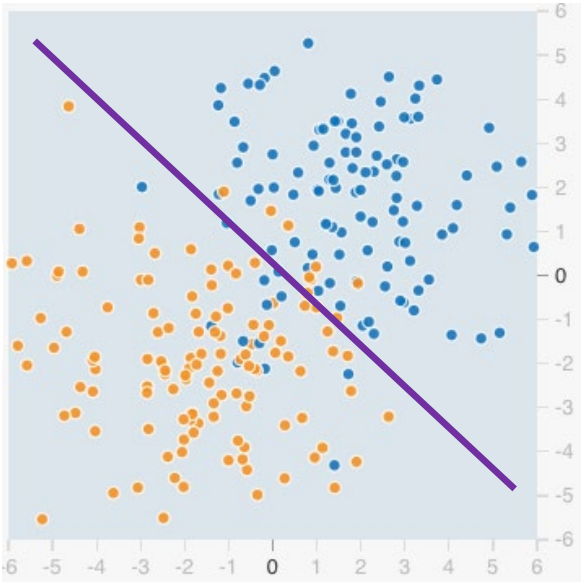
Bias and variance trade-off

- If we consider a classification problem:

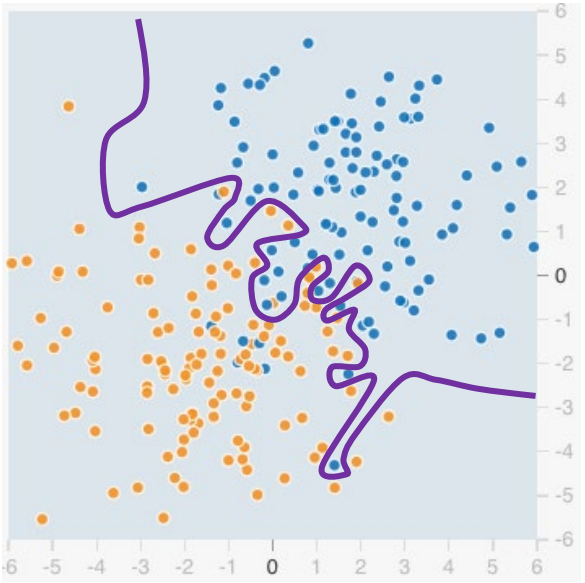


What is the category for the point $(x = -2, y = 1)$?

low variance/high bias

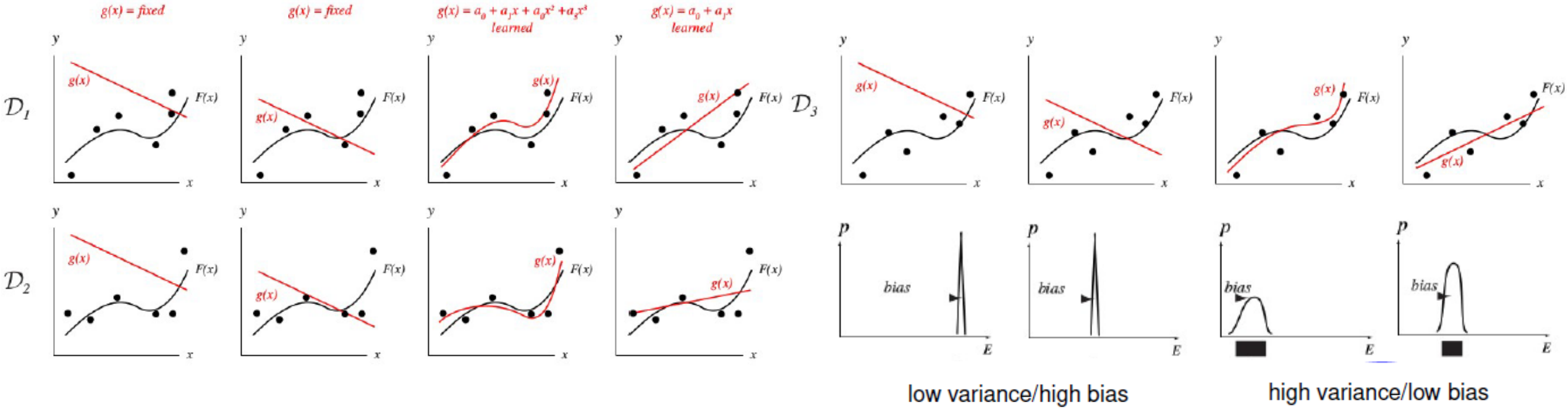


high variance/low bias



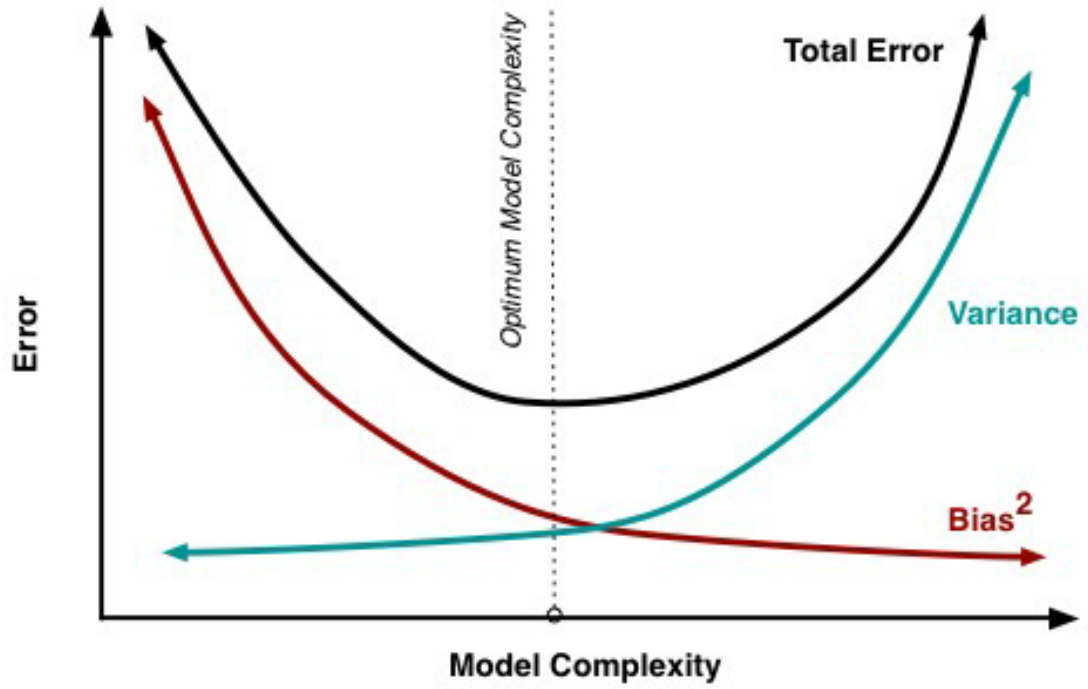
Bias and variance trade-off

- If we consider a regression problem:



Bias and variance trade-off

- Changing the number of parameters and checking the total generalization error we may find a trade-off complexity for a certain model.



However, since $f(x)$ is unknown, we cannot compute these curves

How to estimate bias and variance?

- Estimate the value of **bias** with $E_{train} - E_{human}$
- Estimate the value of **variance** with $E_{test} - E_{train}$

E_{train} Train error (with train dataset)
E_{test} Test error (with dev dataset)
E_{human} Human error (estimation of Bayes error)

Example 1

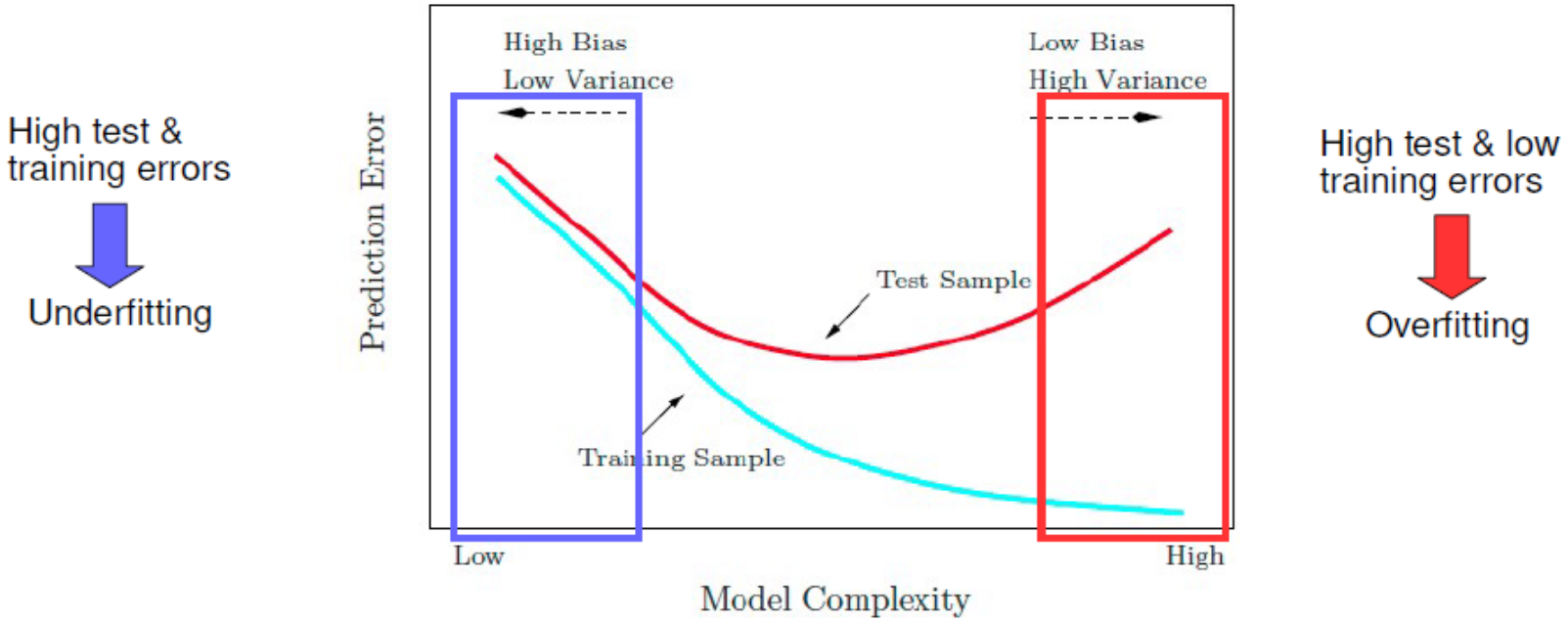
E_{human}	E_{train}	E_{test}	
0%	1 %	15%	Low-bias, high-variance
0%	15 %	16%	High-bias, low-variance
0%	15 %	30%	High-bias, high-variance
0%	0.5 %	1%	Low-bias, low-variance

Example 2

E_{human}	E_{train}	E_{test}	
1%	8 %	10%	High-bias, low-variance
7.5%	8 %	10%	Low-bias, high-variance

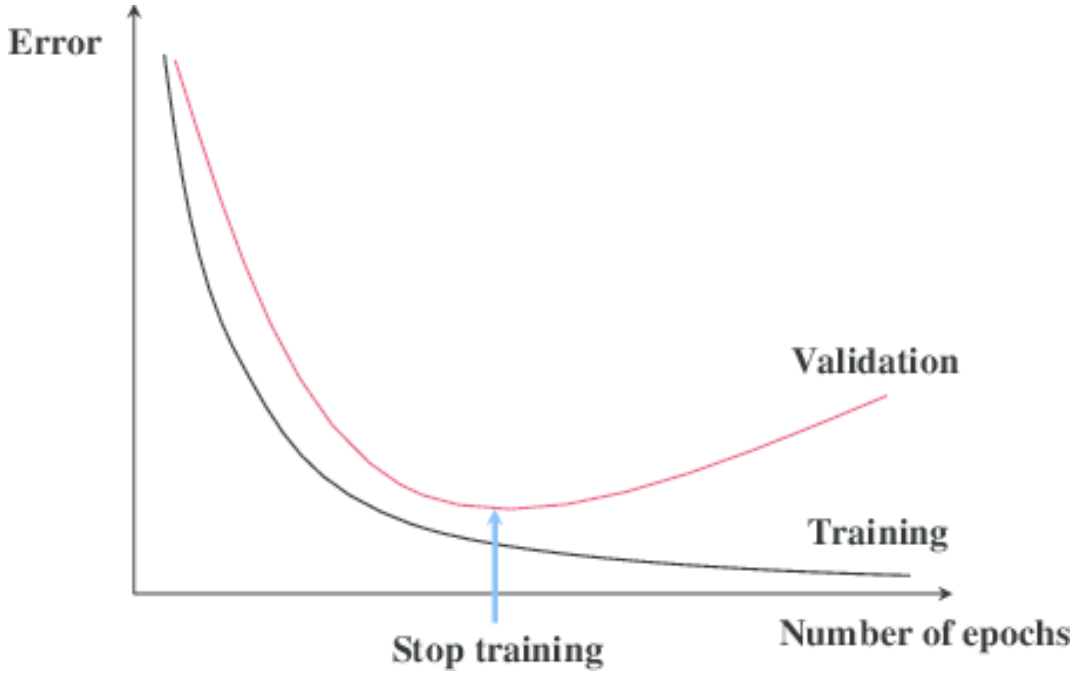
How to estimate bias and variance?

- We can estimate this trade-off from the observation that:
  - **Underfitting:** Models tend to have similar poor goodness of fit performance both on training and testing data sets
  - **Overfitting:** Models tend to have much worse goodness on fit performance on a testing data set compared to a training data set.



Early stopping

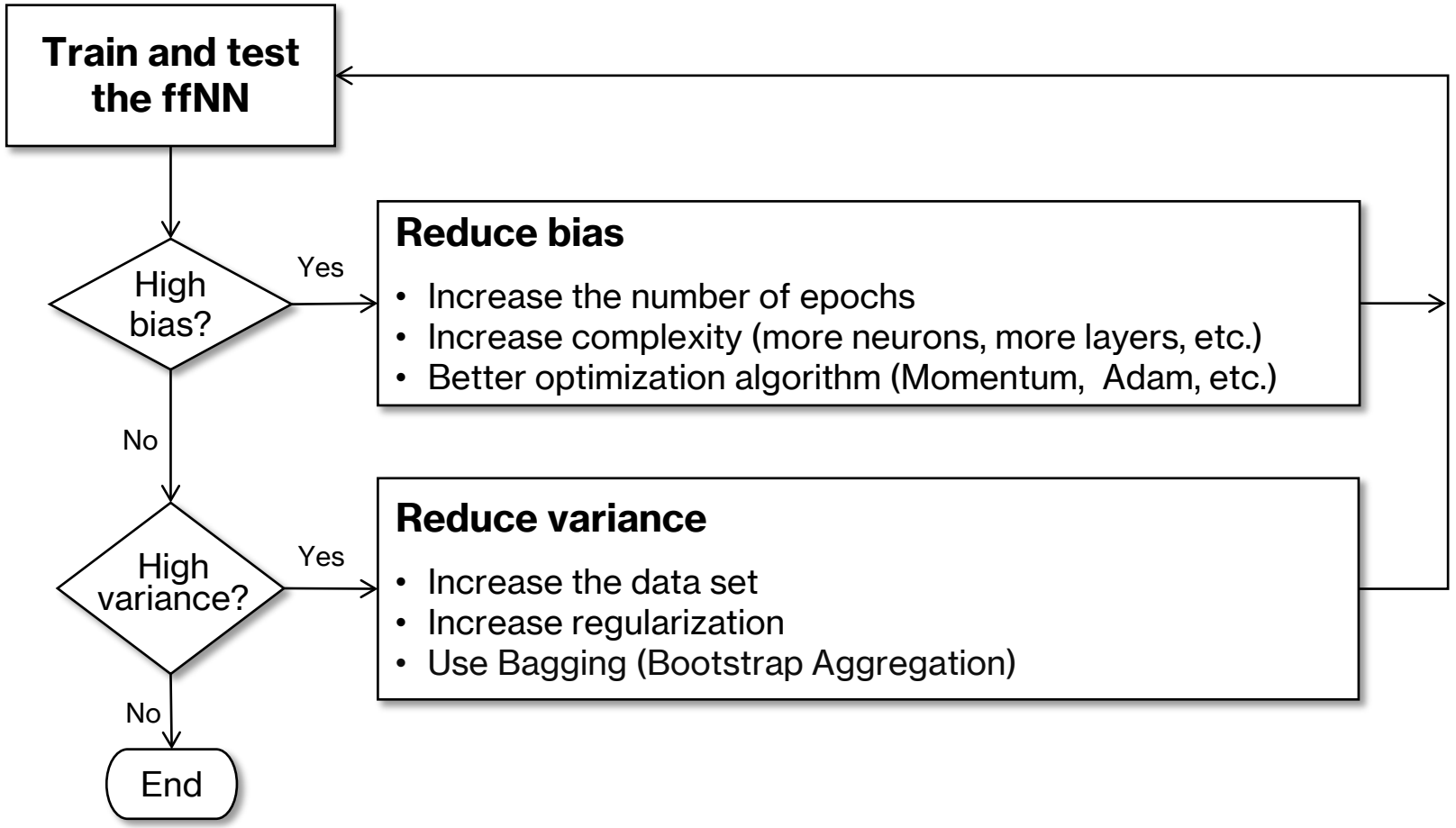
- Early stopping is a technique used while training neural networks to prevent the model from overfitting
- As the training process progresses, some weights usually grow up
- If we stop earlier, we'll keep weights smaller
  - Stops training when the validation error increases
- Early stopping requires less computational power than other methods (see Regularization layers)

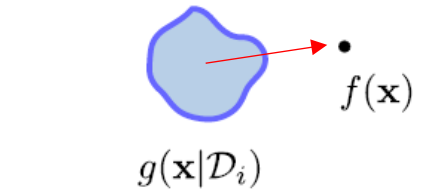


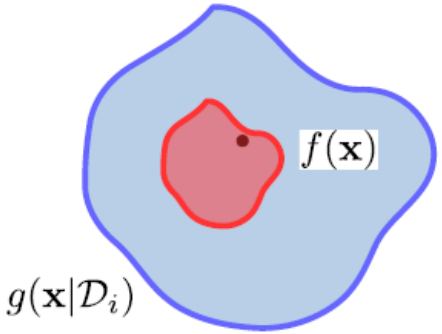

```
from tensorflow.keras.callbacks import EarlyStopping


early_stop = EarlyStopping('val_accuracy', patience=40, verbose=1)
model.fit(…, callbacks=[early_stop], verbose=1)
```

Strategies to improve model performance

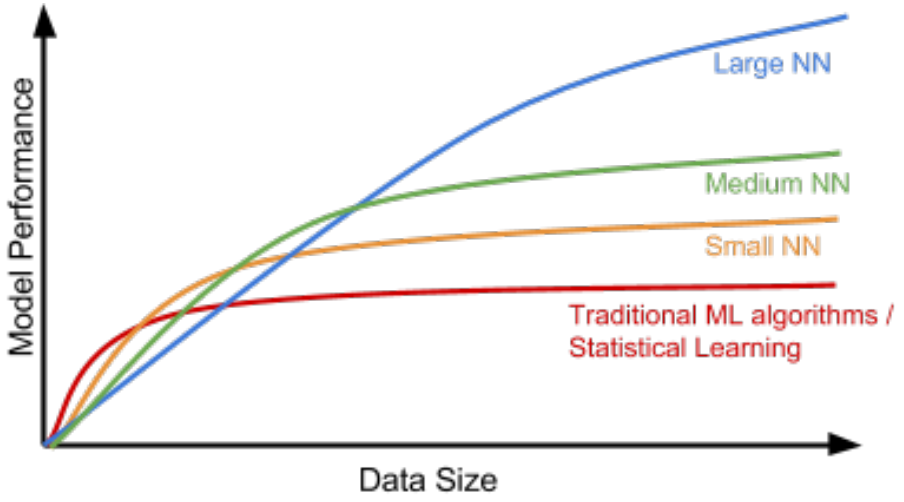






Strategies to improve model performance

- The relationship between the data scale and the model performance.
- Compared to a traditional ML model, a neural network model has many more parameters and has the capability to learn complicated nonlinear patterns.



© Proposed by Andrew Ng in his "Nuts and Bolts of Applying Deep Learning" talk.

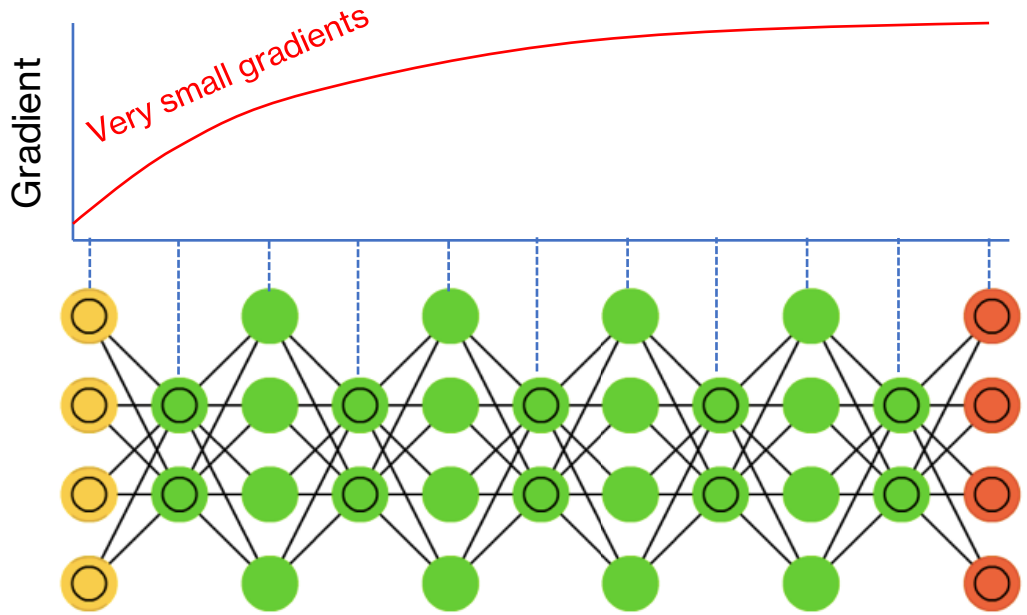
Contents

- Introduction and motivation
- Model assessment
- *Initialization and normalization*
  - *Initialization*
  - *Standardization*
  - *Batch normalization*
- Regularization layers

Vanishing gradient problem

- The output activation of a NN with L layers is given by: $\mathbf{z}^l = \mathtt{W}^l\mathbf{R}(\ldots\mathtt{W}^3\mathbf{R}(\mathtt{W}^2\mathbf{R}(\mathtt{W}^1\mathbf{x})))$
- For simplification, we assume $\mathbf{R}(\mathbf{z}) = \mathbf{z}$ then: $||\mathbf{z}^l|| \approx ||\mathtt{W}^l|| \ldots ||\mathtt{W}^1|| \cdot ||\mathbf{x}|| \approx ||\mathtt{W}||^l \cdot ||\mathbf{x}||$
- For large enough number of layers L:
  - If $||\mathtt{W}|| > 1$ then $||\mathbf{z}^l|| \longrightarrow \infty$
  - If $||\mathtt{W}|| < 1$ then $||\mathbf{z}^l|| \longrightarrow 0$
- The same happens to the gradients, so learning may diverge or become stalled.

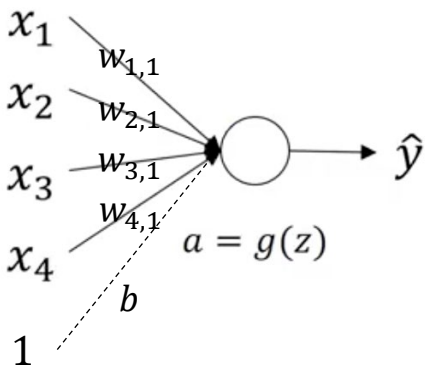
A partial solution is to carefully initialize the network weights *W*



Weight initialization

- What should be the size of the weights w in relation to the number of inputs n_x?
- We want to avoid too large values of z (or too small)
- If n is large then w should be small to avoid too large z

$n_x = 4$



$n_y = 1$

Weights W are initialized to avoid large or too small z

$$z = w_{1,1}x_1 + w_{2,1}x_2 + \ldots + w_{n,1}x_n + b$$

Normal distribution to initialize weights

- One good way is to assign the weights from a Gaussian distribution
- **Xavier (Glorot) initialization:** pick the weights from N with zero mean and a variance of $1/n$ being n the number of neurons in the weight tensor
  - In the general case, the n_x and n_y of a layer may not be equal. Use the average number of neurons (n_x+n_y)/2
  - Glorot and Bengio considered sigmoid and hyperbolic tangent activation functions, which was the default choice at that moment for their weight initialization scheme, but it is not quite as optimal for ReLU functions

$$\mathrm{W} \sim \mathcal{N}\left(0, \sigma^2 = \frac{2}{n_{inputs} + n_{outputs}}\right)$$ **Sigmoid** **tanh**

- **He initialization:** only one tiny adjustment we need to make, which is to multiply the variance of the weights by 2!

$$\mathrm{W} \sim \mathcal{N}\left(0, \sigma^2 = \frac{4}{n_{inputs} + n_{outputs}}\right)$$ **ReLU**

Initializers in Keras

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense
from tensorflow.keras.initializers import Zeros, HeNormal

model = Sequential()
model.add(InputLayer(input_shape=INPUTS))
model.add(Dense(units=neurons, activation='relu', kernel_initializer=HeNormal(),
bias_initializer=Zeros()))
model.add(Dense(units=OUTPUTS, activation='softmax'))
```

◆ Available initializers

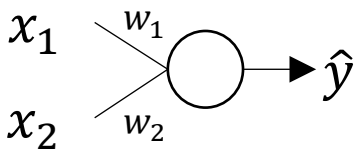
- RandomNormal class
- RandomUniform class
- TruncatedNormal class
- Zeros class
- Ones class
- GlorotNormal class
- GlorotUniform class

- HeNormal class
- HeUniform class
- Orthogonal class
- Constant class
- VarianceScaling class
- LecunNormal class
- LecunUniform class
- Identity class

© https://keras.io/initializers/

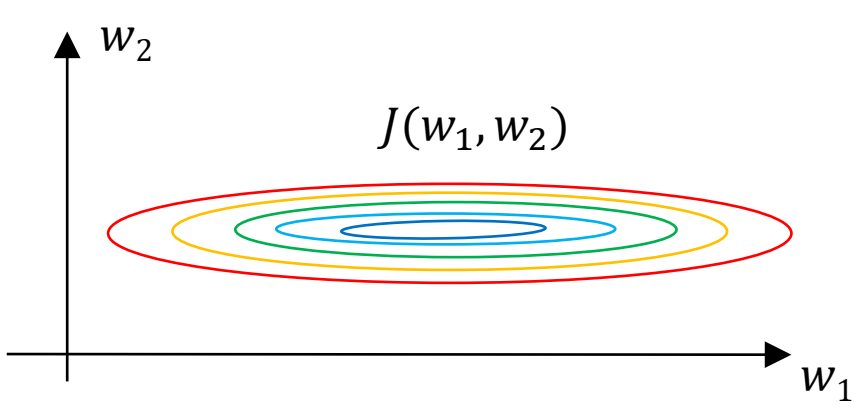
Standard normalization

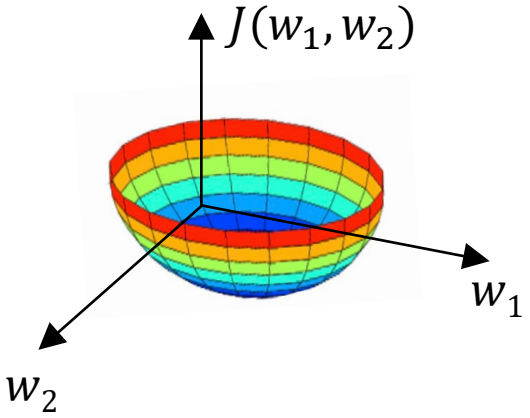
- The cost function can be visualized as surface represented with contour lines
- Input data with very different range values produce elongated surfaces for the cost function

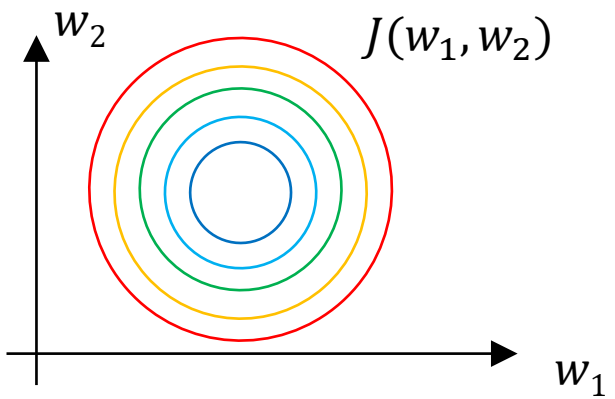


x_1: [-50,000 , … , 1,000,000]

x_2: [0.0 , … , 0.01]



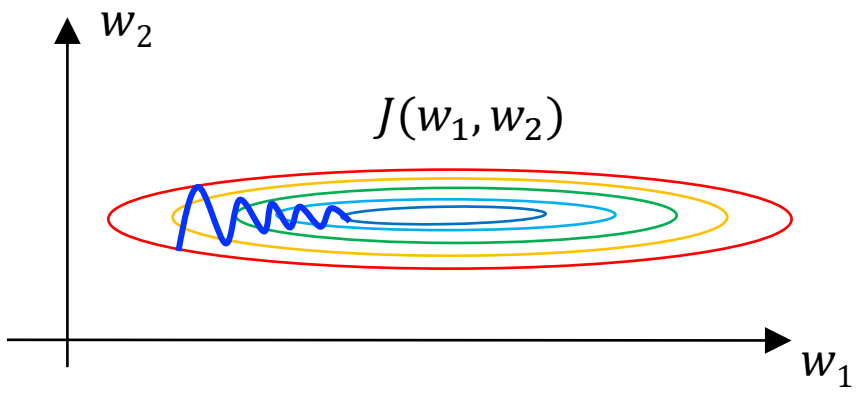




Standard normalization

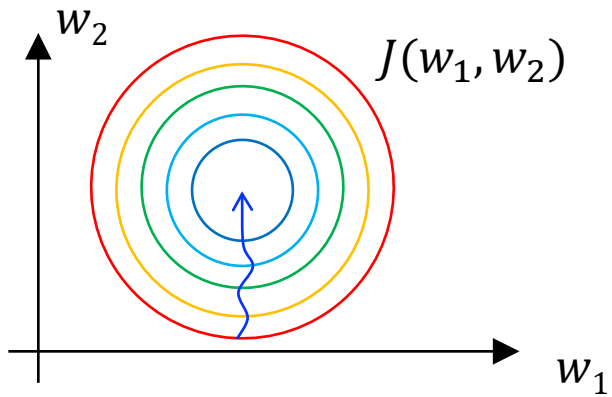
- The training process tries to reach the minimum value in an iterative progress (gradient descent)
- An elongated surface may generate an inefficient convergence to the minimum value

Elongated surface



Inefficient convergence

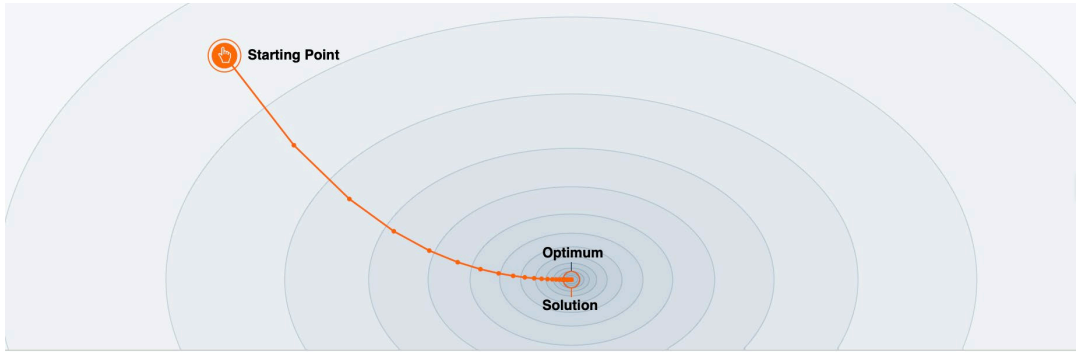
Normalized surface



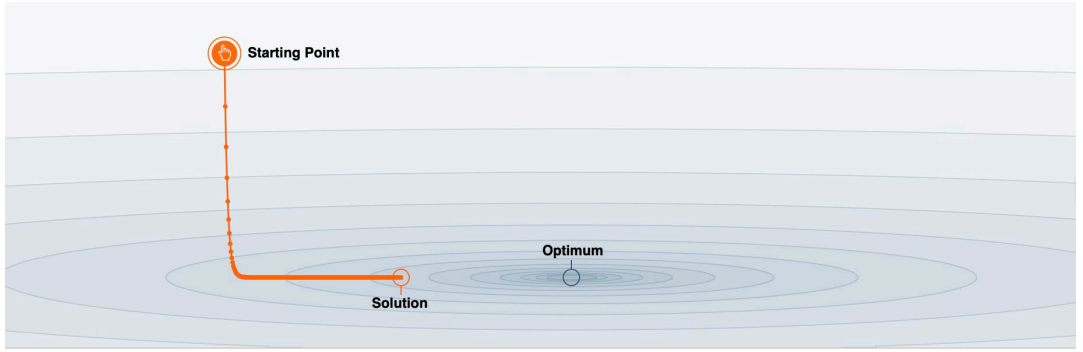
More efficient convergence

Example

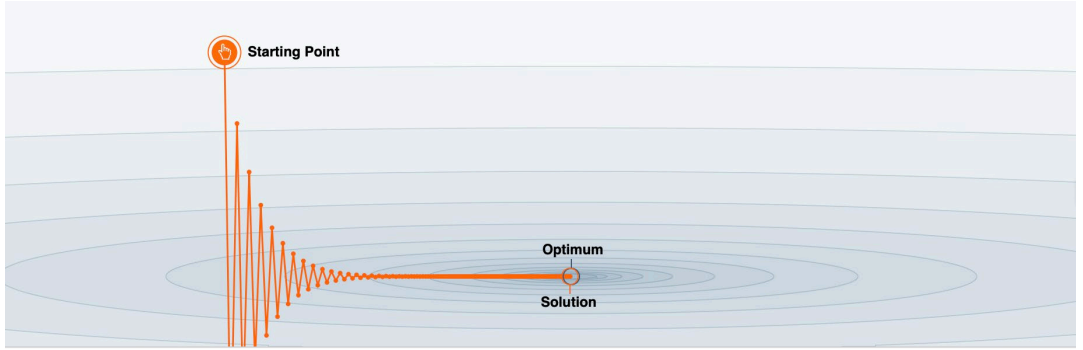
Normalized surface



Elongated surface



Lower learning rate



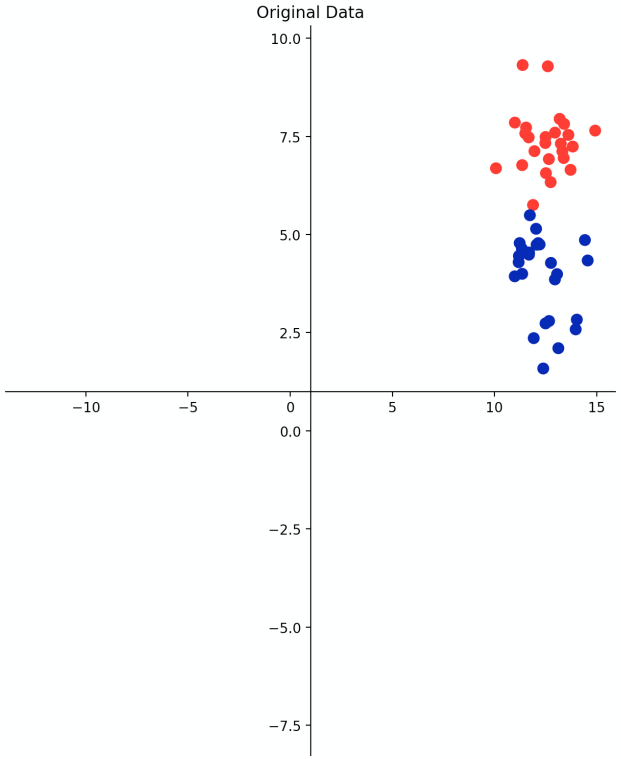
Higher learning rate

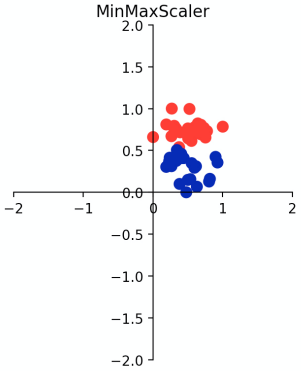
© http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

Min-max scaling

- Values are shifted and rescaled within the range [0,1]

$$\tilde{x} = \frac{x - \min(x)}{\max(x) - \min(x)}$$





Problem: min-max scaling is affected by outliers

Standardization

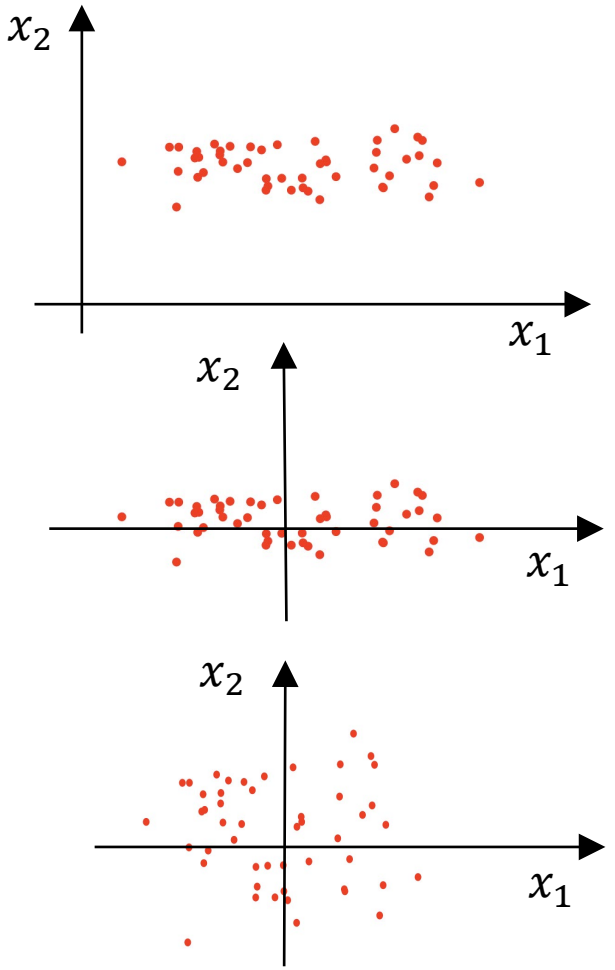
- Values have zero-mean and standard deviation of 1

$$\tilde{x} = \frac{x - \mu}{\sigma^2}$$

1. $\mu \leftarrow \frac{1}{m} \sum_{p=1}^{m} X^{(p)}$
2. $X \leftarrow X - \mu$
3. $\sigma^2 \leftarrow \frac{1}{m} \sum_{p=1}^{m} X^{(p)2}$
4. $X \leftarrow X \,/\, \sigma^2$

m number of examples

$X^{(p)}$ example p



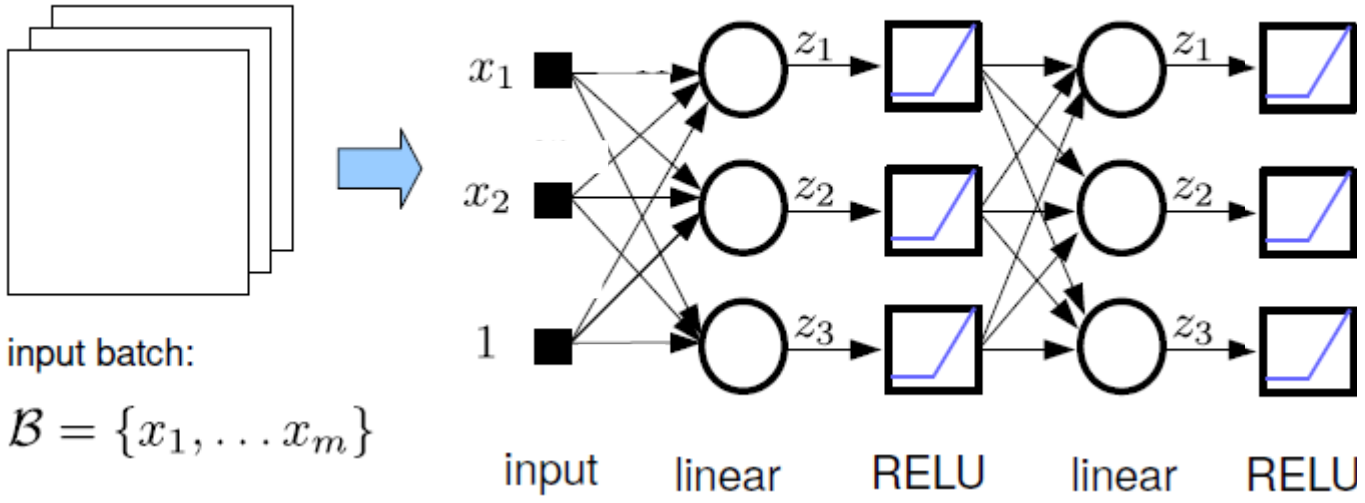
Important: use same μ and σ^2 to normalize all sets (train, validation, test)

Contents

- Introduction and motivation
- Model assessment
- Initialization and normalization
- *Regularization layers*
  - *Batch normalization*
  - *Norm penalties*
  - *Dropout*

Batch normalization

- Standardizes the inputs to all activation units for each training batch



$$\begin{cases} z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} = g(z^{[2]}) \end{cases}$$

Normalize $z^{[2]}$ to learn $W^{[3]}, b^{[3]}$ faster

$$\begin{cases} z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\ \tilde{z}^{[2]} = \mathrm{BN}(z^{[2]}) \\ a^{[2]} = g(\tilde{z}^{[2]}) \end{cases}$$

© Sergey Ioffe, Christian Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML 2015: 448-456

Batch normalization

- Standardizes the inputs to all activation units for each training batch
- Value z is “standardized”:
  - Subtract mean μ and divide by variance σ^2 of the batch
  - Hyperparameter ε avoids division by zero

$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

- However, values of z with mean 0 and variance 1 generate small activation values
- Therefore, value z is “scaled” and “shifted”:
  - Avoids mean 0 and variance 1
  - Parameters γ and β are learned like weights w

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

© Sergey Ioffe, Christian Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML 2015: 448-456

Batch normalization

- Network can learn: $\gamma = \sigma_{\mathcal{B}}; \quad \beta = \mu_{\mathcal{B}}$ so the identity may be represented
- At test time there is no mini-batch to compute μ and σ^2
  - Instead, we use the mean and standard deviation from the whole training set
- Alternative normalization schemes: Layer Normalization, Instance Normalization, Group Normalization

Advantages	Disadvantages
Improves optimization (gradients controlled by learning): allows higher learning rates	There is a runtime penalty: the neural network makes slower predictions due to extra computations required at each layer
Regularizes: improves model generalization	Does not work for very small batches.
Reduces dependence on initialization	The statistics of batches may change between train/test data sets

© Sergey Ioffe, Christian Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML 2015: 448-456

Batch normalization in Keras

Normalize **after** activation function (normalize $a^{[l]}$)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense, BatchNormalization

model = Sequential()
model.add(InputLayer(input_shape=INPUTS))
model.add(Dense(units=neurons, activation='relu', kernel_initializer='he_normal'))
model.add(BatchNormalization())
model.add(Dense(units=OUTPUTS, activation='softmax'))
```

Normalize **before** activation function (normalize $z^{[l]}$)

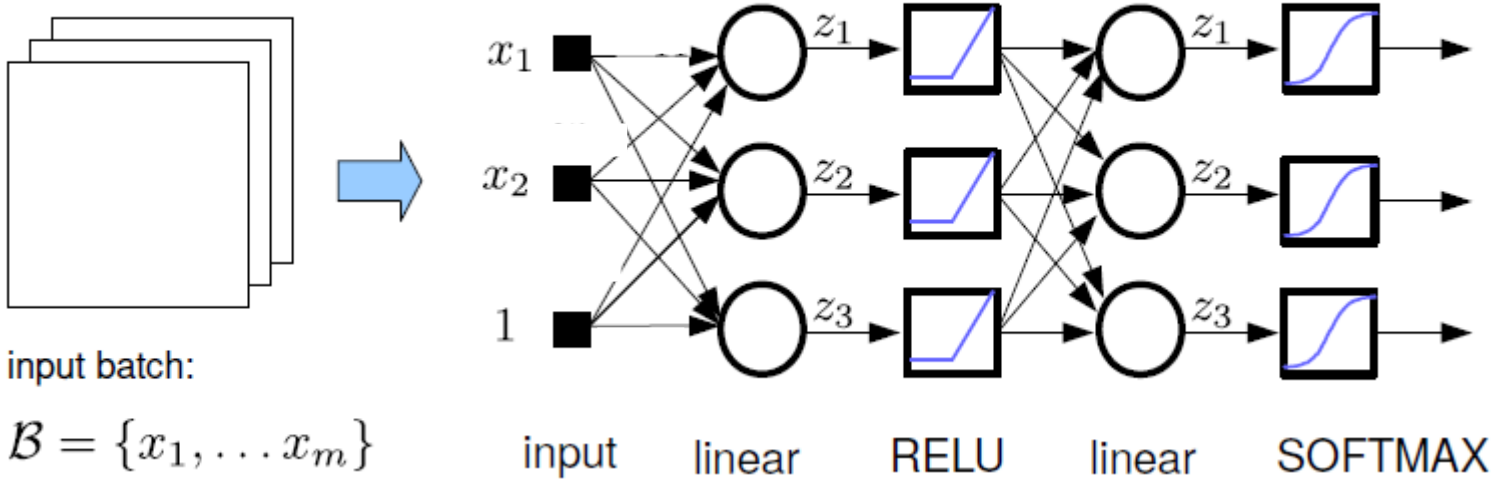
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense, BatchNormalization, Activation

model = Sequential()
model.add(InputLayer(input_shape=INPUTS))
model.add(Dense(units=neurons, kernel_initializer='he_normal', use_bias=False))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(units=OUTPUTS, activation='softmax'))
```

We do not use the bias vector b because batch normalization includes the offset parameter β

Norm penalties

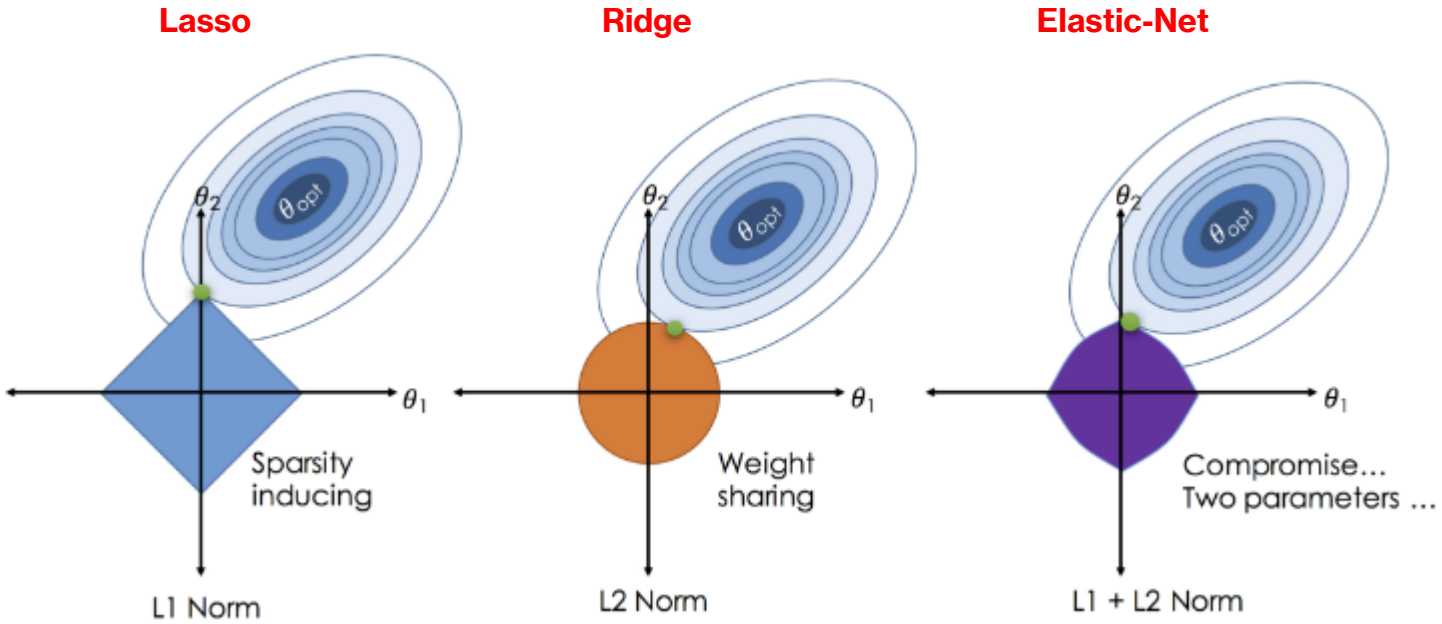
- As the training process progresses, some weights may be very large
- Some problems related to overfitting are related with large weights
- A possible solution is to keep weights small



$$\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + penalization$$

Regularization coefficients

- Regularization is implemented to avoid overfitting of the data by penalizing large coefficients (w_i)
- There are three main techniques for regularization in linear regression:



Lasso and Ridge regularization

- These penalties are summed into the loss function that the network optimizes:

- **Lasso** regression or “Least Absolute Shrinkage and Selection Operator” regularizes using the L1 norm.

$$\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_1$$

- This term pushes the model’s weights to become exactly zero.
  - Suitable when some weights are irrelevant.
  - Can eliminate some features.

- **Ridge** regression regularizes using a L2 penalty term which regulates the magnitude of the weights in the model.

$$\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

- Shrinks the weights towards zero.
  - Can prevent overfitting.

Norm penalties in Keras

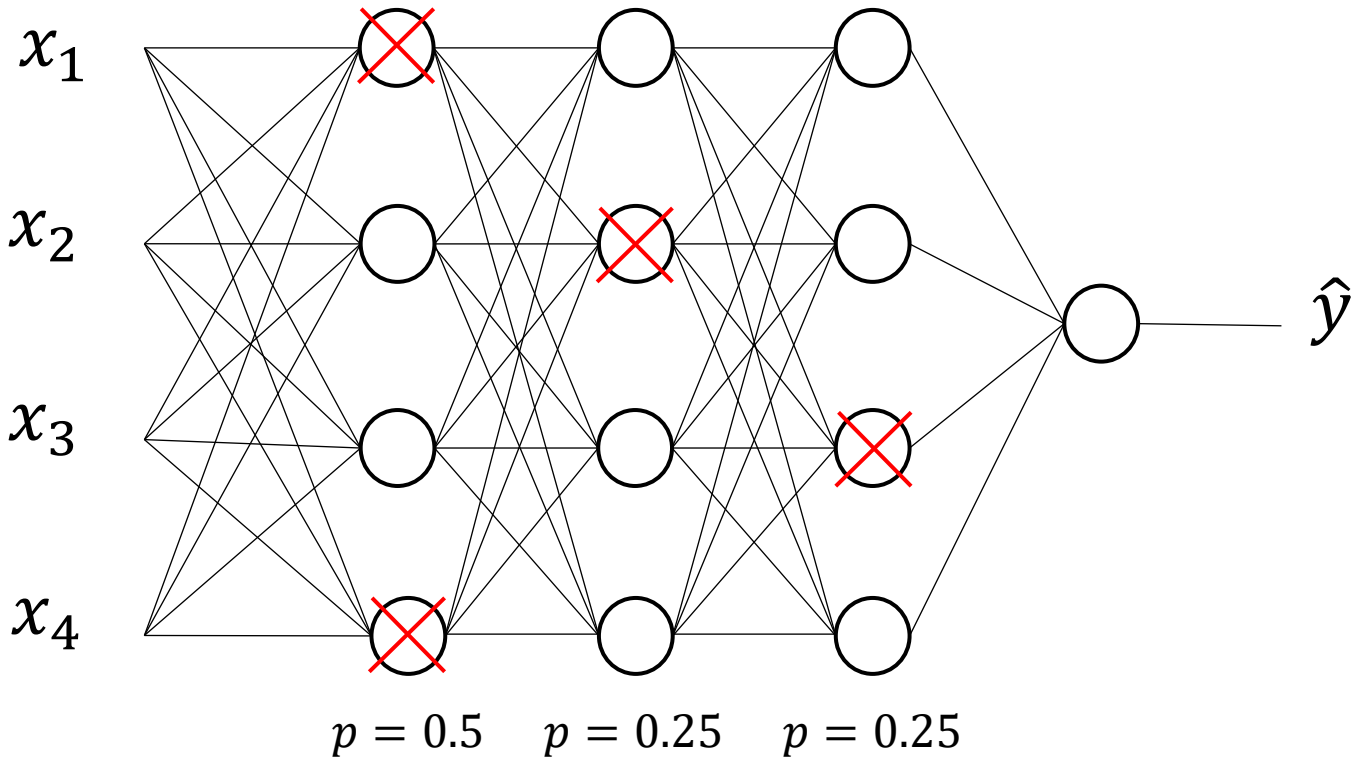
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense
from tensorflow.keras.regularizers import L2

model = Sequential()
model.add(InputLayer(input_shape=INPUTS))
model.add(Dense(units=neurons, activation='relu', kernel_regularizer=L2(1e-4),
bias_regularizer=L2(1e-4), activity_regularizer=L2(1e-5)))
model.add(Dense(units=OUTPUTS, activation='softmax'))
```

© https://keras.io/api/layers/regularizers/

Dropout

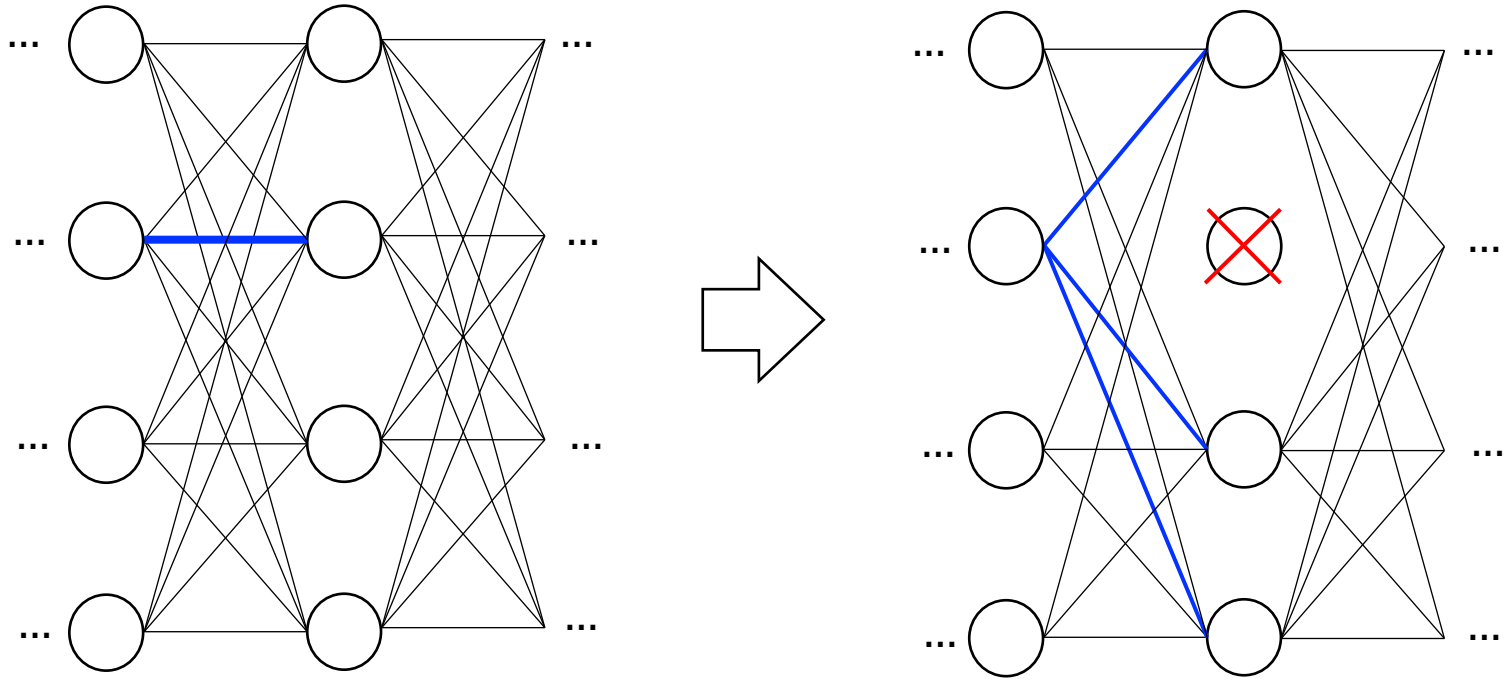
- Prune the NN by randomly discarding hidden units during training
- Each hidden unit is multiplied by 0 with a certain probability. The rate "p" represents the fraction of units to drop
  - Hidden units must learn more general features
- Changes on the training algorithm:
  - Forwprop: each layer includes a random [0,1] mask that multiplies each activation
  - Backprop: hidden units multiplied by 0 do not contribute to the gradient backprop



© Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15(1): 1929-1958 (2014)

Dropout

- The weights are distributed (shrink weights):



Inverted dropout

- If we apply dropout directly:
  - Testing loss corresponds to the complete network
  - Training loss corresponds to a network with fewer neurons and connections
- To compare them we can multiply each activation by the probability p at testing time: $a^{[l]} \leftarrow a^{[l]} \times p$
  - **Problem:** This approach adds complexity during testing
- Instead of multiplying activations by p during testing time, we divide during training time: $a^{[l]} \leftarrow a^{[l]} / p$
  - Inverted dropout is the recommended option to minimize processing time during testing

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense, Dropout

model = Sequential()
model.add(InputLayer(input_shape=INPUTS))
model.add(Dense(units=neurons, activation='relu')
model.add(Dropout(rate=0.2))
model.add(Dense(units=OUTPUTS, activation='softmax'))
```