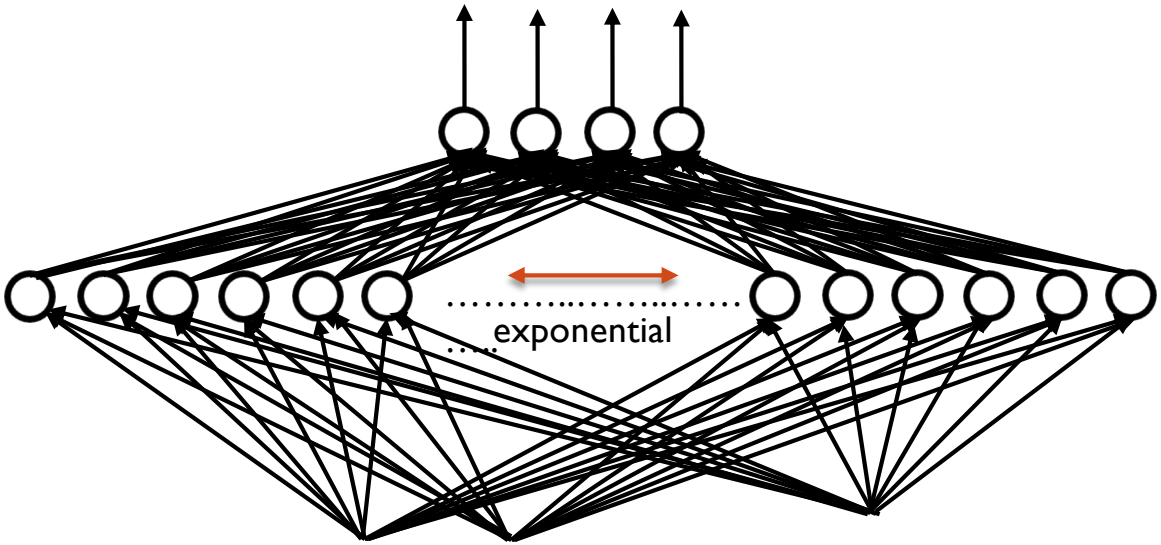
Neural networks
Hyper-parameters

Michel.RIVEILL@univ-cotedazur.fr

Main neural network architectures

Deep representation origins

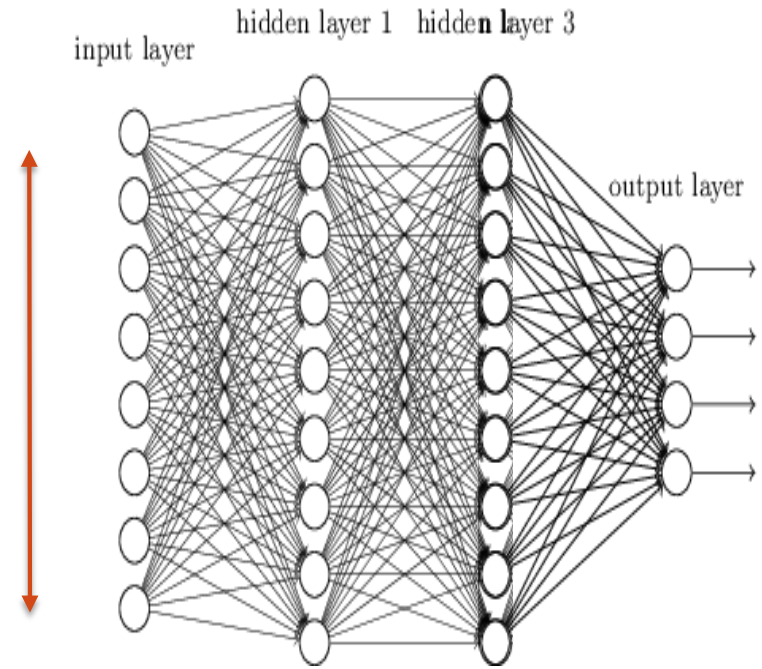
- **Theorems** (Cybenko (1989), Hornik & Stinchcombe & White (1989))
  A neural network with one single hidden layer is a universal "approximator", it can represent any continuous function on compact subsets of $\mathbf{R}^n \Rightarrow$ *2 layers are enough…* ***but*** *hidden layer size may be exponential for error* ε *(or even infinite for error 0), and there is no efficient learning rule known.*



Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems, 2(4), 303-314.
Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, 2(5), 359-366.

Deep representation origins

- **Theorem Hastad** (1986), **Bengio** et al. (2007) Functions representable compactly with *k* layers may require exponentially size with *k-1* layers



Johan T. Håstad. Computational Limitations for Small Depth Circuits. MIT Press, Cambridge, MA, 1987.
Bengio, Y., & LeCun, Y. (2007). Scaling learning algorithms towards AI. Large-scale kernel machines, 34(5), 1-41.

Enabling factors

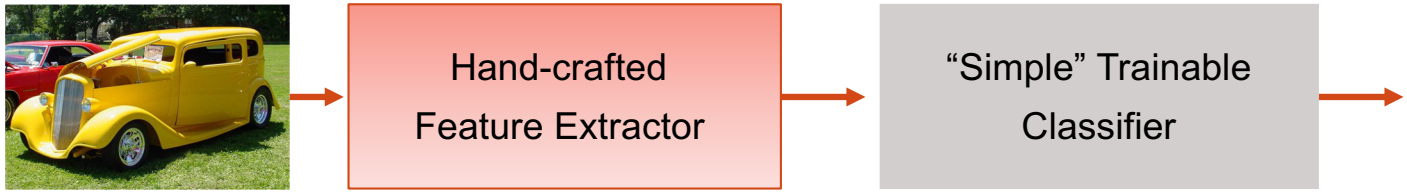
- Why do it now ? Before 2006, training deep networks was unsuccessful because of practical aspects
  - faster CPU's
  - parallel CPU architectures
  - advent of GPU computing
  - Advances in ML/Optim (1995 ->

- Hinton, Osindero & Teh « A Fast Learning Algorithm for Deep Belief Nets », *Neural Computation*, 2006
- Bengio, Lamblin, Popovici, Larochelle « Greedy Layer-Wise Training of Deep Networks », *NIPS'2006*
- Ranzato, Poultney, Chopra, LeCun « Efficient Learning of Sparse Representations with an Energy-Based Model », *NIPS'2006*

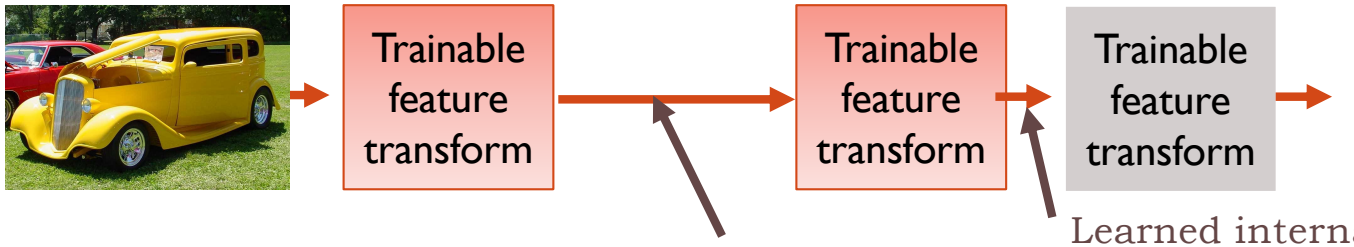
- Results…
  - 2009, sound, interspeech +~24%
  - 2011, text, +~15% without linguistic at all
  - 2012, images, ImageNet +~20%
  - 2020, molecules/graphs, AlphaFold, +~24% (protein folding)
    Repliement des protéines:
    https://www.youtube.com/watch?v=OGewxRMME8o (French)

Functions of a deep network

- The traditional model of pattern recognition (since the late 50's)
  - Fixed/engineered features (or fixed kernel) + trainable classifier

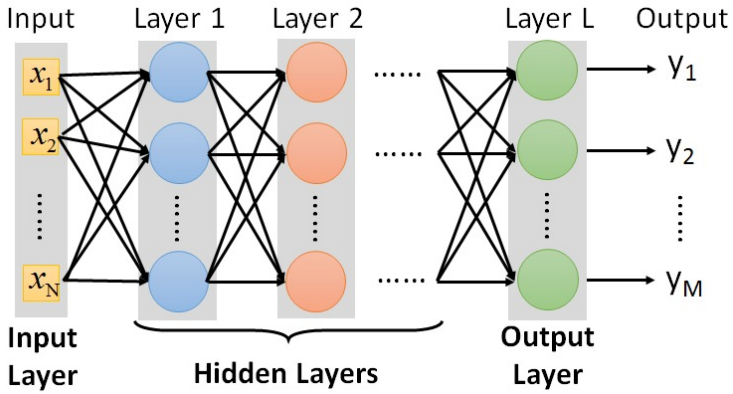


- A hierarchy of trainable feature transforms
  - Each module transforms its input representation into a higher-level one.
  - High-level features are more global and more invariant
  - Low-level features are shared among categories



FAQ

- Q: How many layers? How many neurons for each layer?



Trial and Error

Intuition

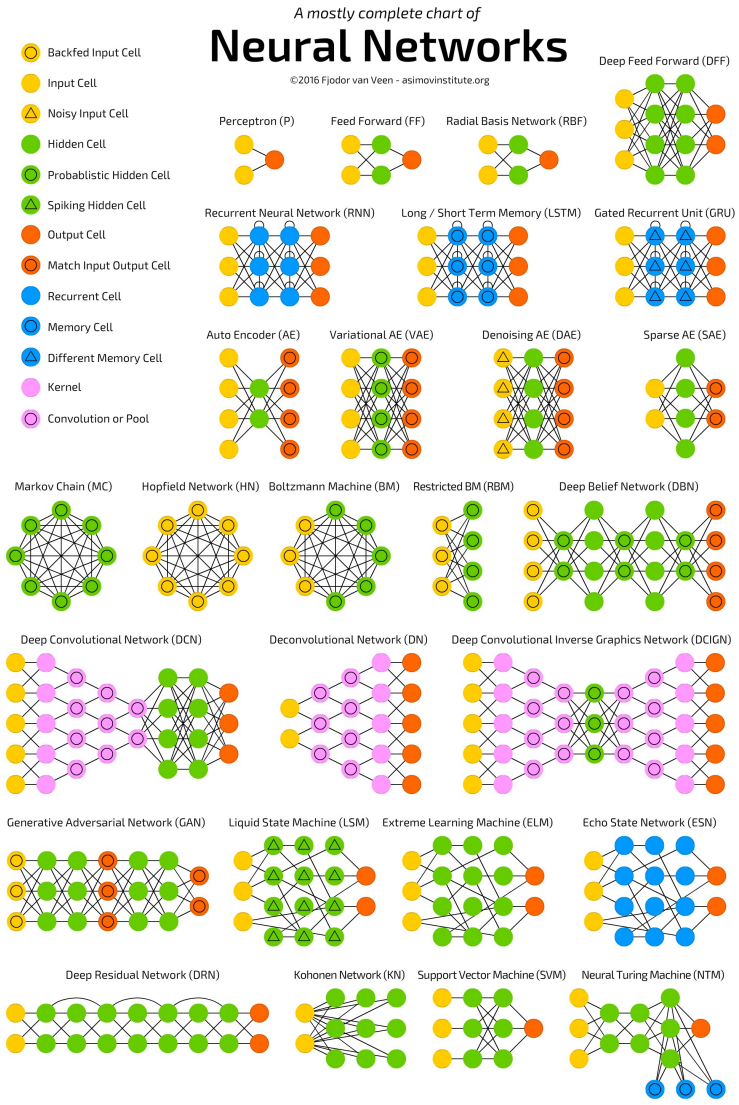
- Q: Can we design a specific network structure?

Next slides

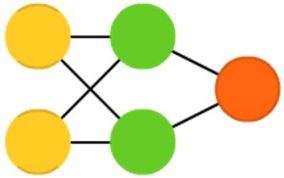
- Q: Can the structure be automatically determined?
  - Yes, intense research in the last 2 years (e.g. **AutoML, AdaNet**).

Topologies of Neural Networks

http://www.asimovinstitute.org/neural-network-zoo/



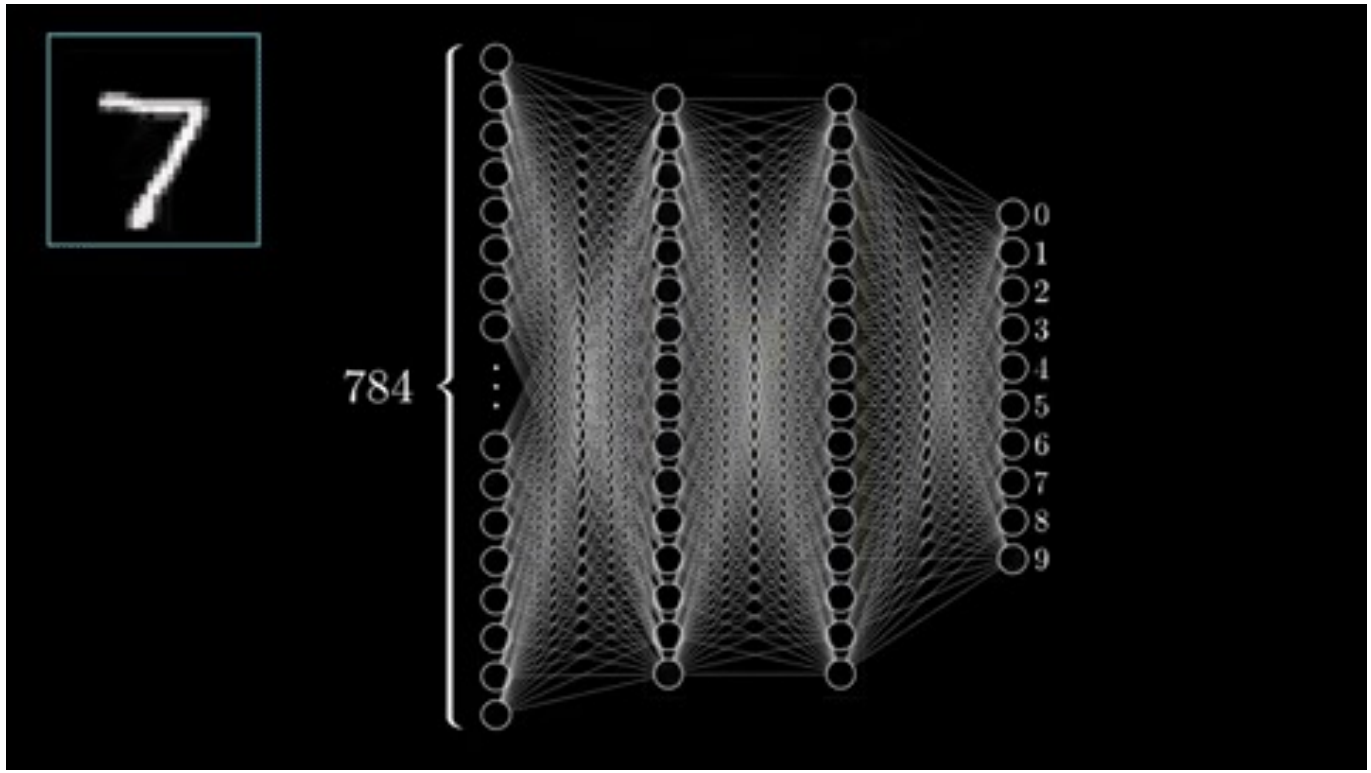
Topologies of Neural Networks



Feed forward or Perceptrons (P)
Original paper: *Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.*

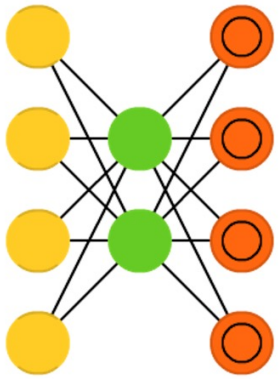
- Feedforward: No loops, input → hidden layers → output
- Supervised networks use a "teacher"
  - The desired output for each input is provided by user

Multi-layer perceptron (MLP)



Source: https://www.3blue1brown.com/neural-networks

Topologies of Neural Networks

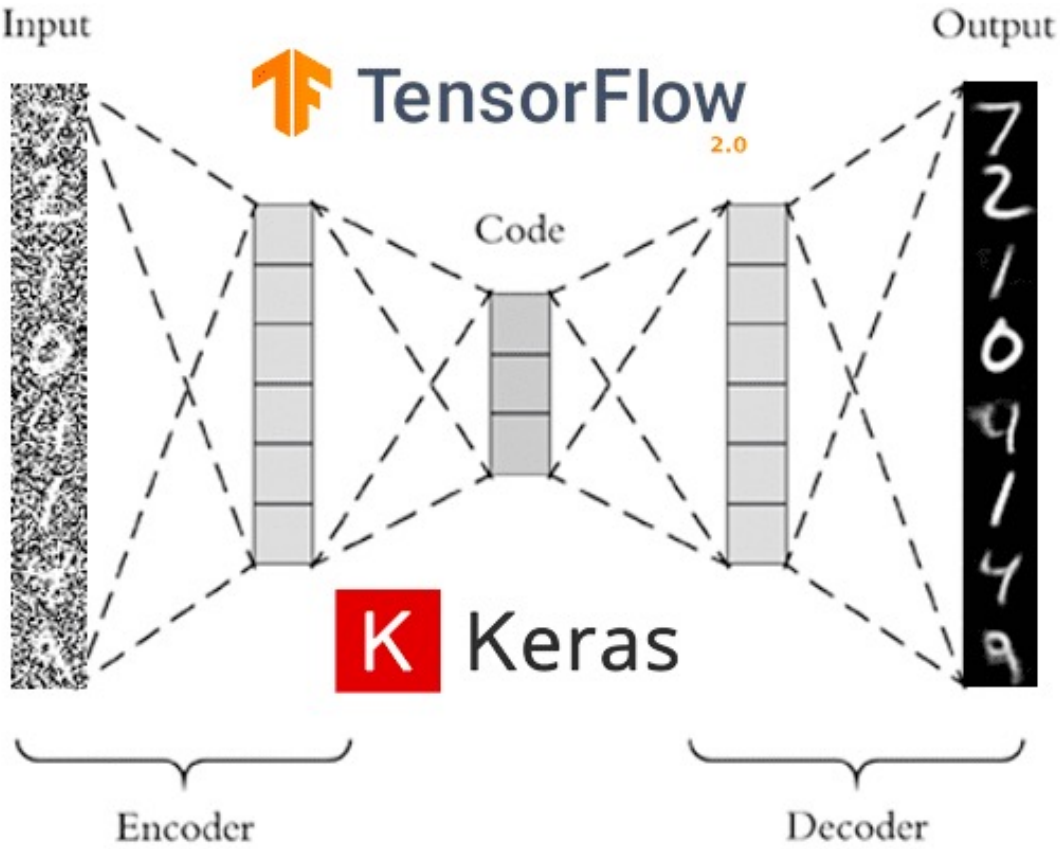


Autoencoders (AE) The basic idea behind autoencoders is to compress information automatically

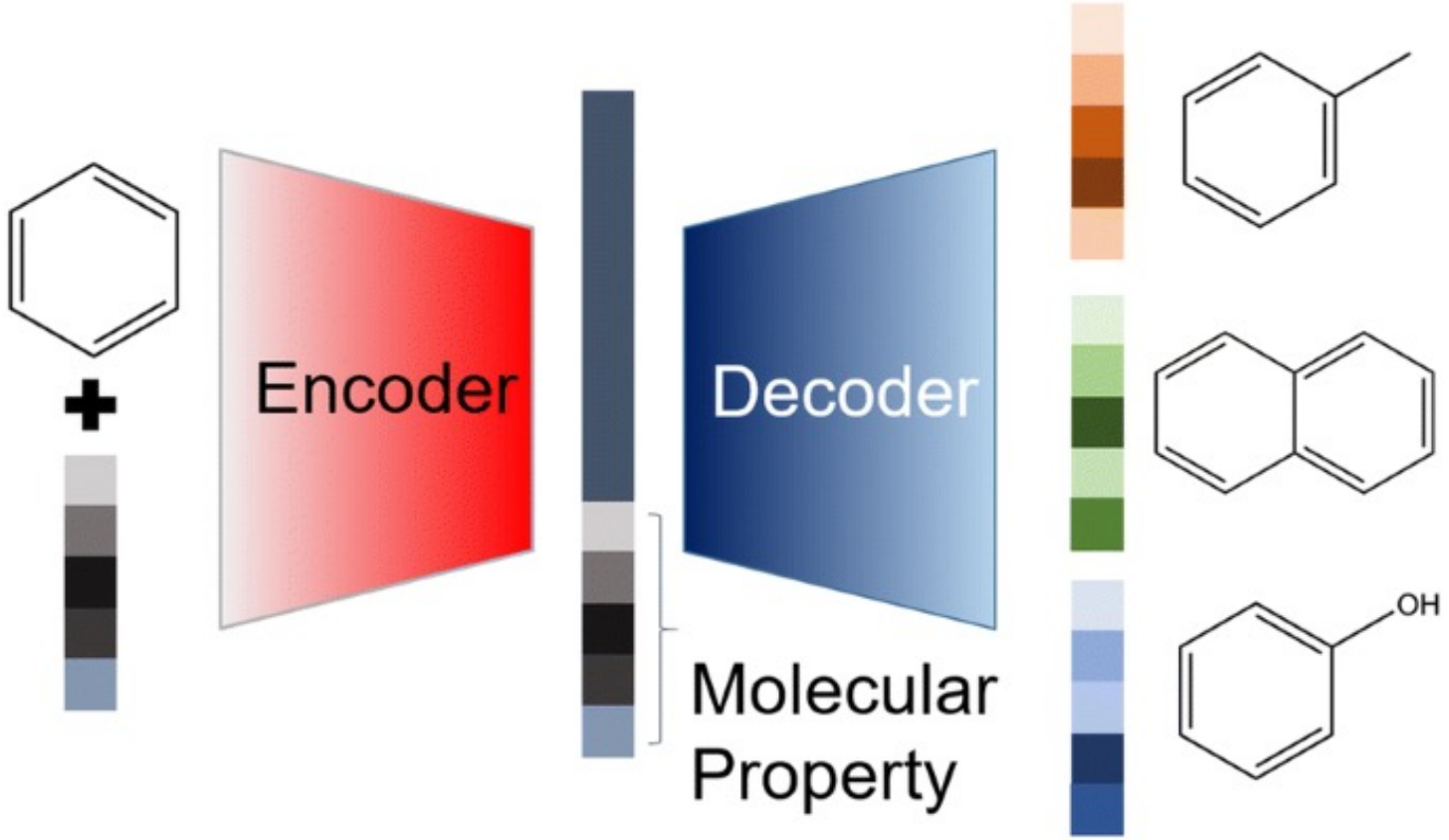
Original paper: *Bourlard, Hervé, and Yves Kamp. "Auto-association by multilayer perceptrons and singular value decomposition." Biological cybernetics 59.4-5 (1988): 291-294.*

Unsupervised networks find hidden statistical patterns in input data

Denoiser auto-encoder



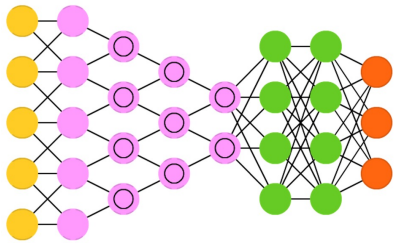
Example of use: generation of drug-like molecules with five target properties



Molecular generative model based on conditional variational autoencoder for de novo molecular design
Jaechang Lim , Seongok Ryu , Jin Woo Kim and Woo Youn Kim, arXiv:1806.05805v1 [cs.LG] 15 Jun 2018

Topologies of Neural Networks

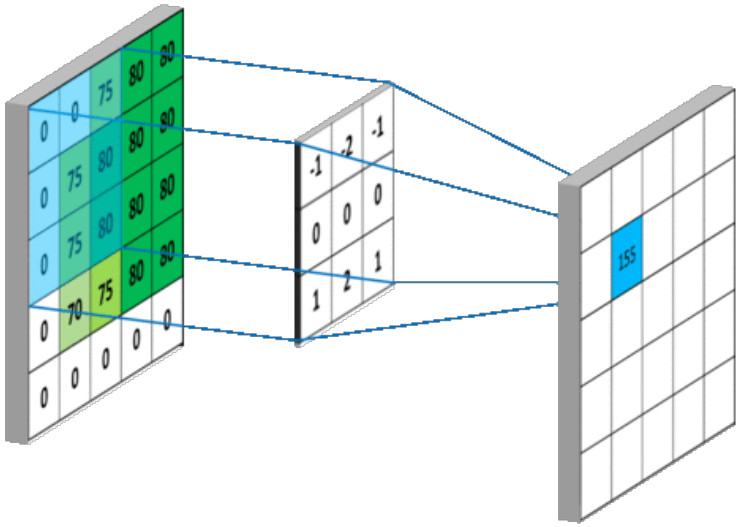
- In mathematics (in particular, functional analysis), convolution is a mathematical operation on two functions (f and g) that produces a third function (f * g) that expresses how the shape of one is modified by the other.
- The term convolution refers to both the result function and to the process of computing it.



Convolutional neural networks (CNN) are quite different from most other networks. They are primarily used for image processing but can also be used for other types of input such as audio or text. Each node only concerns itself with close neighbouring cells.

Original paper: *LeCun,Yann, et al."Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.*

Convolution

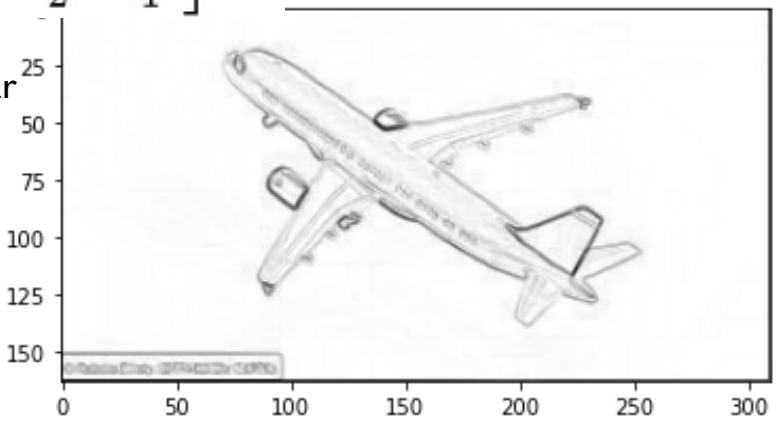


Extraction du contour d'une image : filtre de Solbel

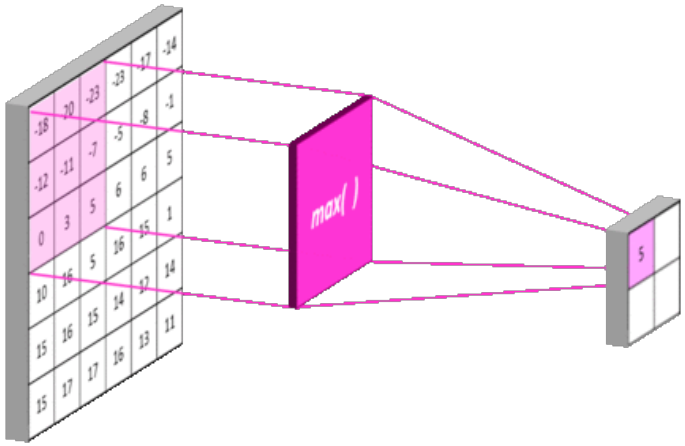
$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \mathbf{A}$$

L'image finale G, est obtenue par

$$G = \sqrt{{G_x}^2 + Gy^2}$$



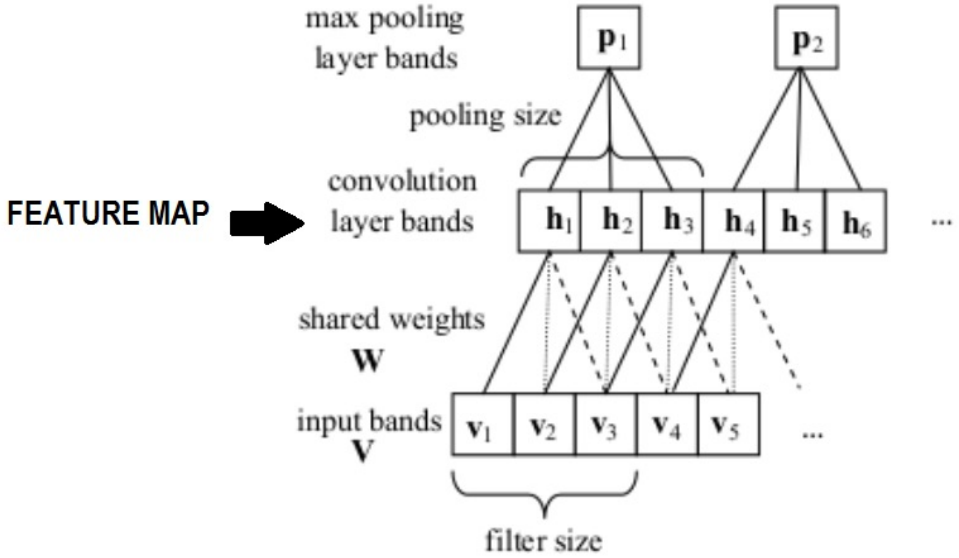
Max pooling


-18
-10
-23
-13
-17
-14
-12
-11
-7
-5
-8
-1
0
3
5
6
6
5
10
16
5
16
15
1
15
16
15
14
17
14
15
17
17
16
13
11
max( )
5

Deep representation by CNN

- Yann Lecun, [LeCun et al., 1998]

1. Subpart of the field of vision and translation invariant
2. S cells: convolution with filters
3. C cells: max pooling

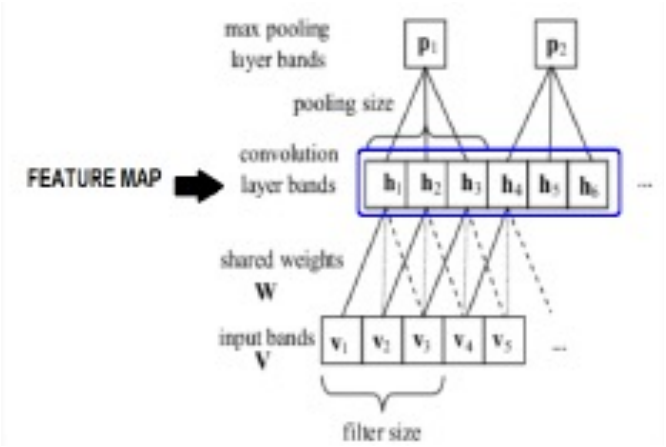


18

Deep representation by CNN

- Yann Lecun, [LeCun et al., 1998]

1. Subpart of the field of vision and translation invariant
2. S cells: convolution with filters
3. C cells: max pooling

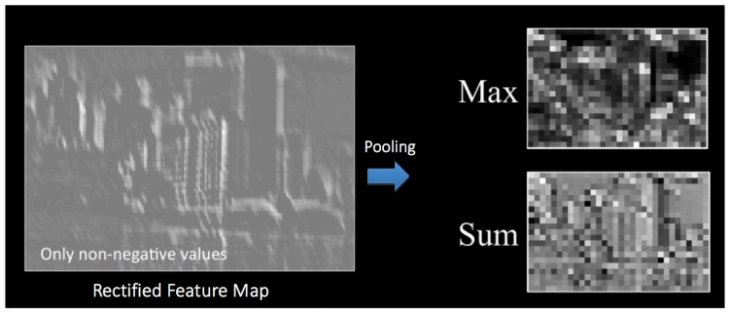


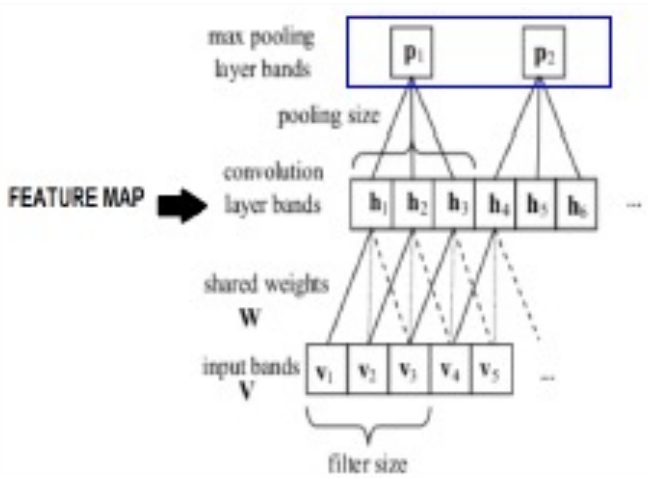
19

Deep representation by CNN

- Yann Lecun, [LeCun et al., 1998]

1. Subpart of the field of vision and translation invariant
2. S cells: convolution with filters
3. C cells: max pooling

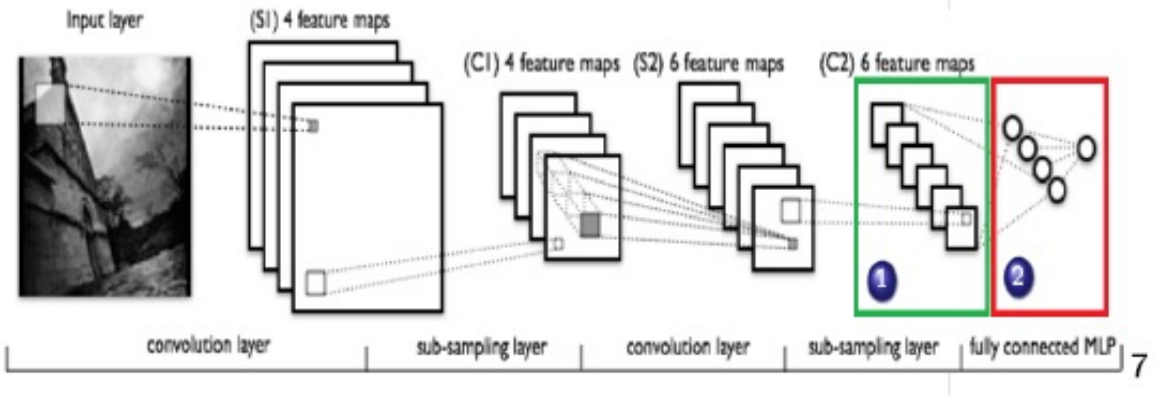




20

Deep representation by CNN

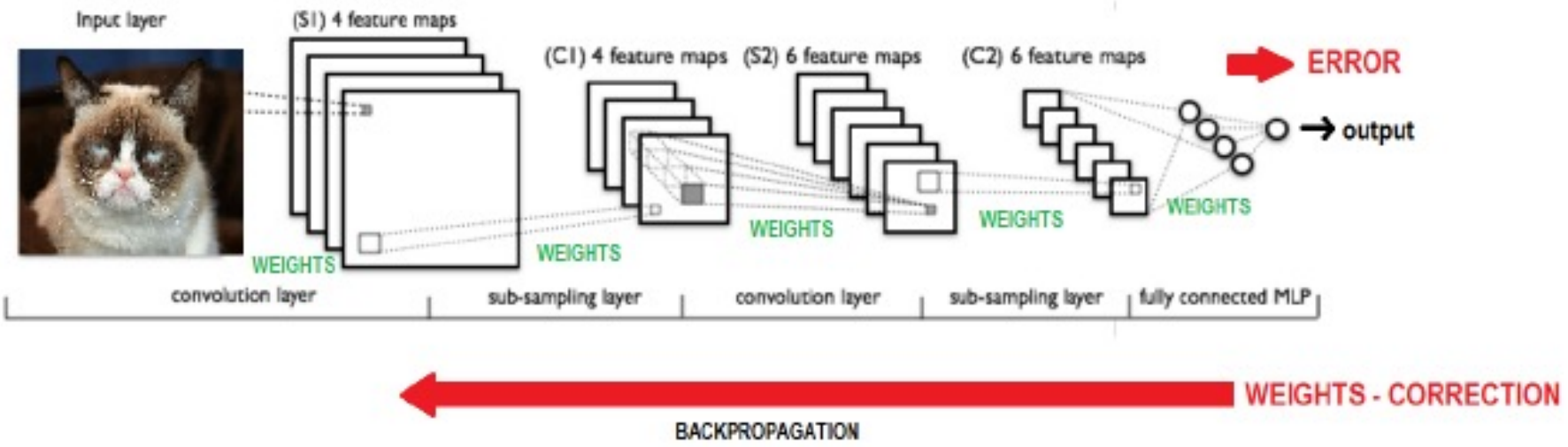
- feature map = result of the convolution
- convolution with a filter extract characteristics (*edge detectors*)
- extract parallelised characteristics at each layer



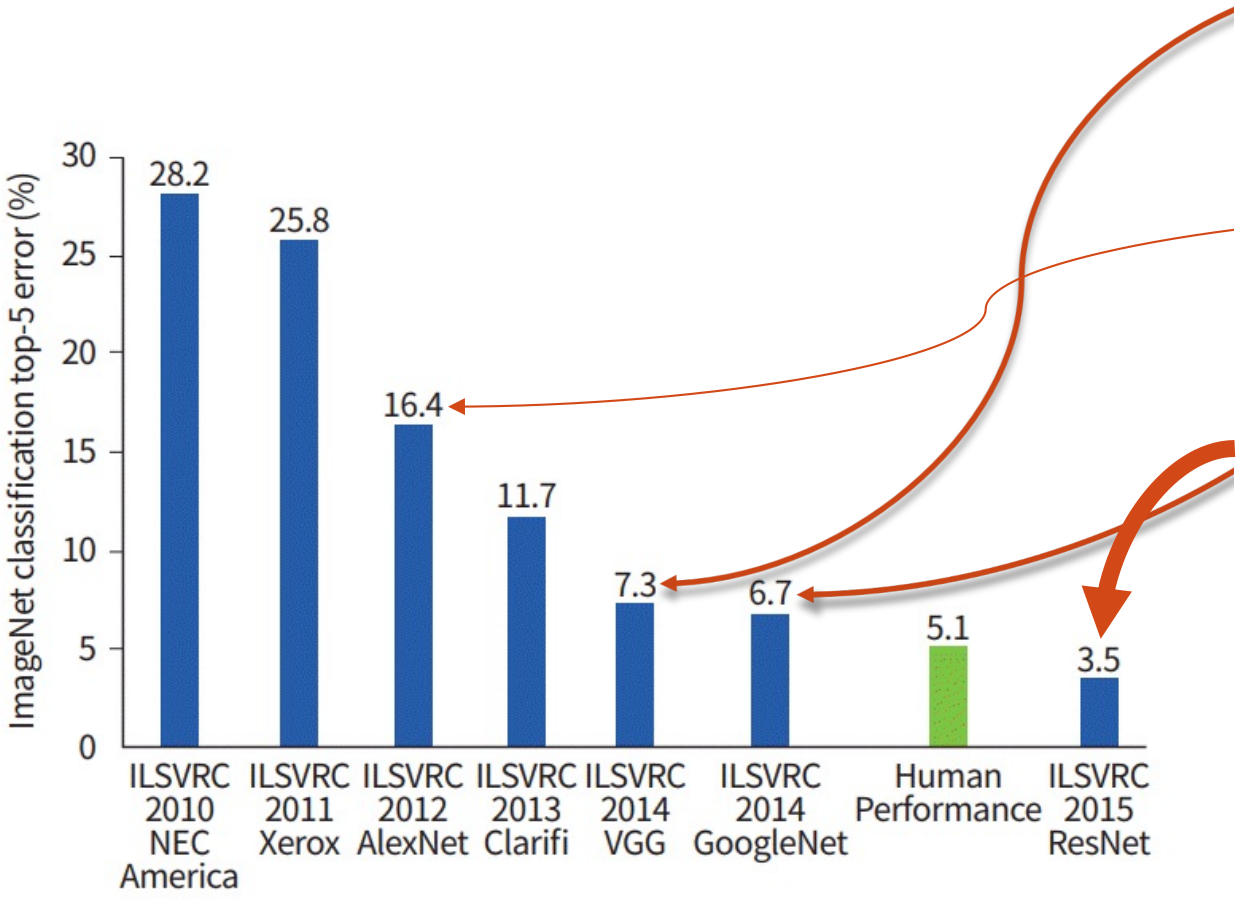
1. final representation of our data
2. classifier (MLP)

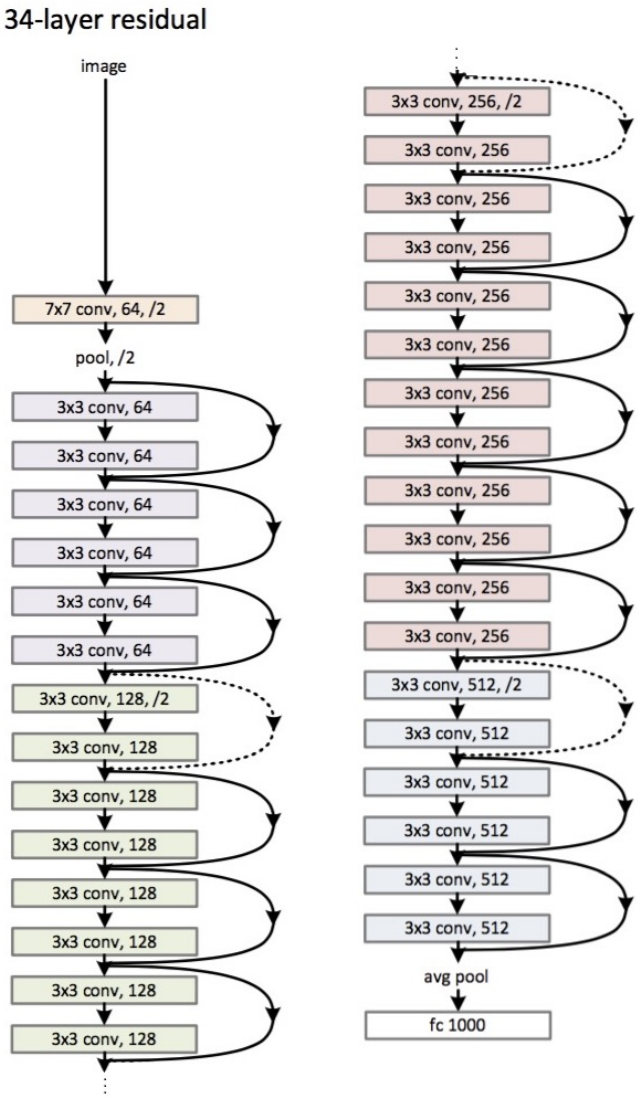
[redacted]

Deep representation by CNN

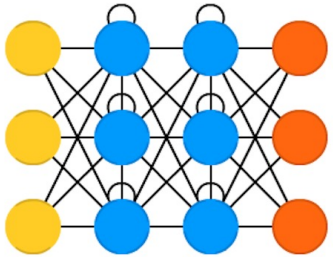


Deep = Many hidden layers



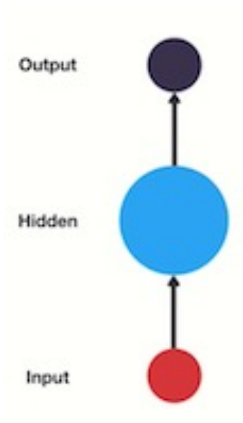


Topologies of Neural Networks



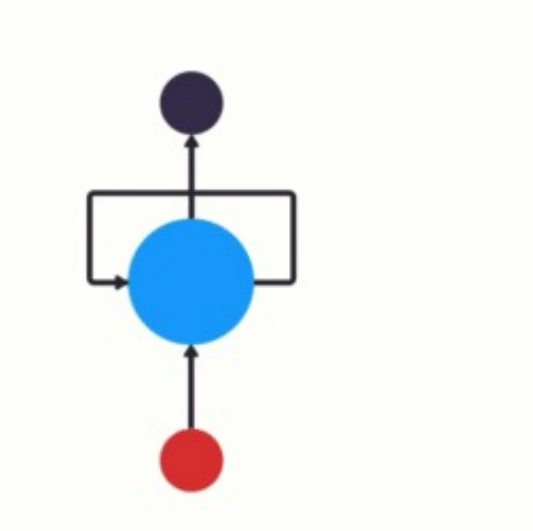
Recurrent neural networks (RNN) they have connections between passes, connections through time. Neurons are feed information not just from the previous layer but also from themselves from the previous pass.
Original paper: *Elman, Jeffrey L. "Finding structure in time." Cognitive science 14.2 (1990): 179-211.*

Traditional neural network



Recurrent neural network = add memory to a cell

RNN representation



Network architecture

Main parameters for a neural network

1. Architecture of the network
   - Input layer
   - Hidden layer (number of layers / number of neurons / activation function)
   - Dropout
   - Output layer
2. Loss and metric function
   - Metric: model evaluation criteria
   - Cost/Loss: Function minimized during learning, must be derivable
   - Ideally: Metric = Cost
3. Batch size
4. Optimization algorithm
   - Objective to converge as quickly as possible to the minimum of the cost function
   - Regularization

- Exercice sur le playground.

Input / Output layers

- Input layer
  - Size: depends on the number of features
- Output layer
  - Regression
    - Activation: linear
    - Number of neurons: depends on the number of variables to predict
      - 1 neuron by variable
  - Classification
    - Activation: softmax
    - Number of neurons: depends on the number of class to predict
      - 2 classes: 1 neuron
      - N>2 classes: 1 neuron by classes
        - Classes are one hot encoded

Loss / Metrics

- Loss : uses during learning process
  - Gradient of the loss → correction of the weigths
  - Regression: MAE, MSE, RMSE, etc.
  - Classification: cross entropy
    - Binary cross entropy: $-\sum\ (yi\log(\hat{y}_i) + (1-yi)\log(1-\hat{y}_i))$
    - Categorical cross entropy: $-\sum y_i(\log(\hat{y}_i)$
    - Interpret the result as a conditional probability: Log(P(x=c|y))
- Metrics : uses for model evaluation
  - Regression: MAE, MSE, RMSE, etc.
  - Classification: accuracy, recall, precision, F1, etc.

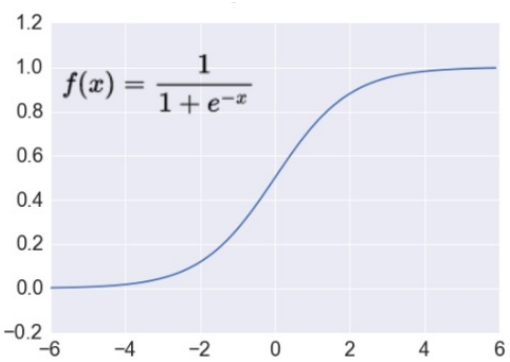
Softmax

- Softmax takes an N-dimensional vector of real numbers and transforms it into a vector of real number with a sum equal to 1
  - $pi = \frac{e^i}{\sum_j e^j}$
  - It outputs could be interpreted as a probability distribution in classification tasks.

Category	Scoring function	unnormalized probabilities $\mathbf{UP} = exp^{ULP}$	normalized probabilities $\mathbf{P} = \frac{UP}{\sum UPj}$	normalized log loss $\mathbf{LL} = -\ln(P)$
Dog	-3.44	0.0321	0.0006	7.4186
Cat	1.16	3.1899	0.0596	2.8201
Boat	-0.81	0.4449	0.0083	4.7915
Airplane	3.91	49.8990	0.9315	0.0709
			predict(X)	predict_proba(X)

Activation function

Activation: Sigmoid



http://adilmoujahid.com/images/activation.png

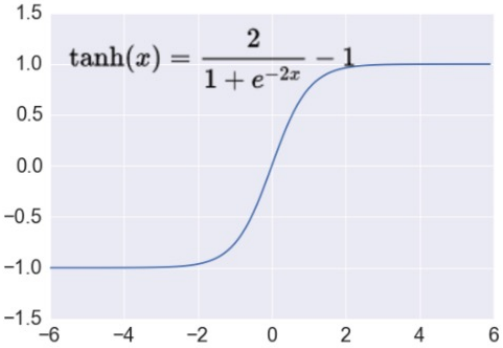
Takes a real-valued number and
"squashes" it into range between 0 and 1.

$$R^n \rightarrow [0,1]$$

\+ Nice interpretation as the **firing rate** of a neuron
- 0 = not firing at all
- 1 = fully firing

\- Sigmoid neurons **saturate** and **kill gradients**, thus NN will barely learn
- when the neuron's activation are 0 or 1 (saturate)
  gradient at these regions almost zero
  almost no signal will flow to its weights
  if initial weights are too large then most neurons would saturate

Activation: Tanh



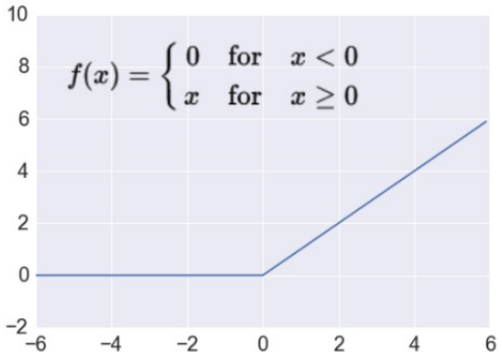
http://adilmoujahid.com/images/activation.png

Takes a real-valued number and "squashes" it into range between -1 and 1.

$$R^n \rightarrow [-1,1]$$

- 🙁 Like sigmoid, tanh neurons **saturate**
- 🙂 Unlike sigmoid, output is **zero-centered**
- Tanh is a **scaled sigmoid**: $\tanh(x) = 2sigm(2x) - 1$

Activation: ReLU



http://adilmoujahid.com/images/activation.png

Takes a real-valued number and thresholds it at zero $f(x) = \max(0, x)$

Most Deep Networks use ReLU nowadays

Trains much **faster**
- accelerates the convergence of SGD
- due to linear, non-saturating form

Less expensive operations
- compared to sigmoid/tanh (exponentials etc.)
- implemented by simply thresholding a matrix at zero

More **expressive**

Prevents the **gradient vanishing problem**

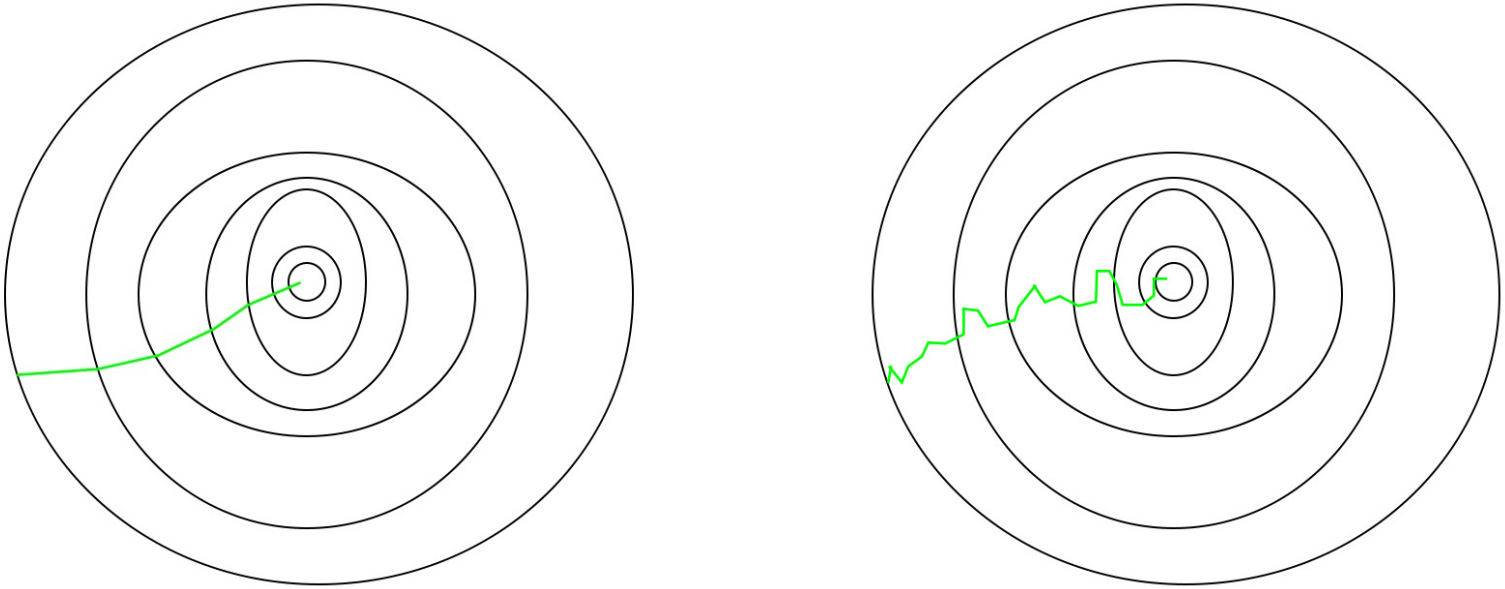
How to choose batch size and learning rate

Epoch vs Batch Size vs Iterations

- **Epochs**
  - One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE.
- **Batch Size**
  - Total number of training examples present in a single batch.
- **Iterations**
  - Iterations is the number of batches needed to complete one epoch.
  - $Iterations = \frac{nb\_items}{batch\_size}$

- There are three types of Gradient Descent:
  - Batch Gradient Descent
    - batch size = number of items
    - 1 iteration by epoch
  - Stochastic Gradient Descent
    - Batch size = 1
    - Nb_items iterations by epoch
  - **Mini-batch Gradient Descent**
    - **1 < batch size << number of items**

Impact of the batch size



in which figure:

- Batch size == 1 ?
- Batch size == nb of items ?

Batch size selection

- SGD (batch size=1 or <<N) is generally noisier than a Gradient Descent per epoch
  - more iterations to reach the minima, due to the random nature of the descent
  - but it is generally faster because it requires fewer epochs.
- Generally use batch size around
  - 32, 64, 128
  - it depends also on the use case and the system memory,
  - i.e., we should ensure that a single mini-batch should be able to fit in the system memory.

Gradient descent algorithm

for i in range$(Number\ of\ training\ steps)$:
  $batches = miniBatchGenerator(X, Y,$ batch_size$)$
  for j in range$(Number\ of\ batches)$:
    $minibatchX, minibatchY = batches[j]$
    Forward Propagation using $minibatchX$to calculate y'
    Calculate cost using $minibatchY$
    Backward Propagation to calculate derivative dW and dB
    Parameter Updation using following rule:
      $W = W - \alpha.dW$
      $b = b - \alpha.db$

How to choose learning rate

- This is probably one of the most important hyper-parameter
  - Set the learning rate too small and your model might take ages to converge
  - Make it too large and within initial few training examples, your loss might shoot up to sky

- Generally
  - Use large learning rate at the beginning
  - Use small learning rate when you reach the minima

  - But, today, a lot of optimizer use adaptative learning rate

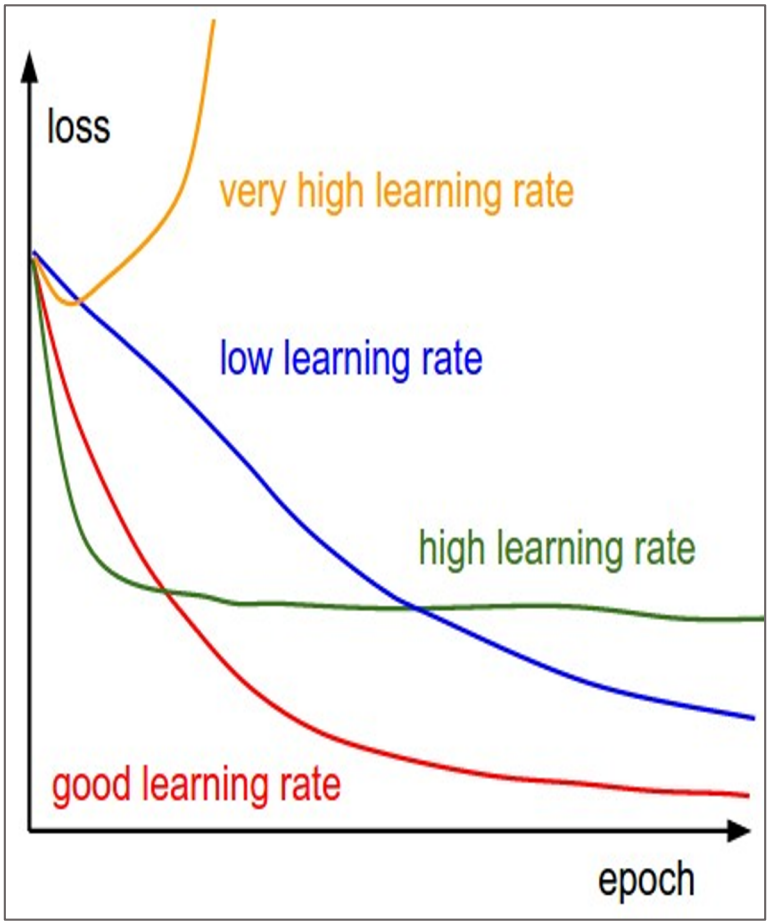
Ajust Learning Rate

Use decline learning rate

- Goal: make rapid progress when you are far from the minimum, be sure to reach it when you are close

Several possibilities

- **step decay:** e.g. decay learning rate by half every few epochs.
  - $\mu_t = \frac{\mu_0}{2^{\lfloor \frac{t}{T} \rfloor}}$
  - μ_0 (initial leaning rate) and T are hyper-parameter
- **exponential decay:**
  - $\mu_t = \mu_0 e^{-kt}$
  - μ_0 (initial leaning rate) and k are hyper-parameter
- **1/t decay:**
  - $\mu_{t+1} = \frac{\mu_0}{1+\frac{t}{T}}$
  - μ_0 (initial leaning rate) and T are hyper-parameter

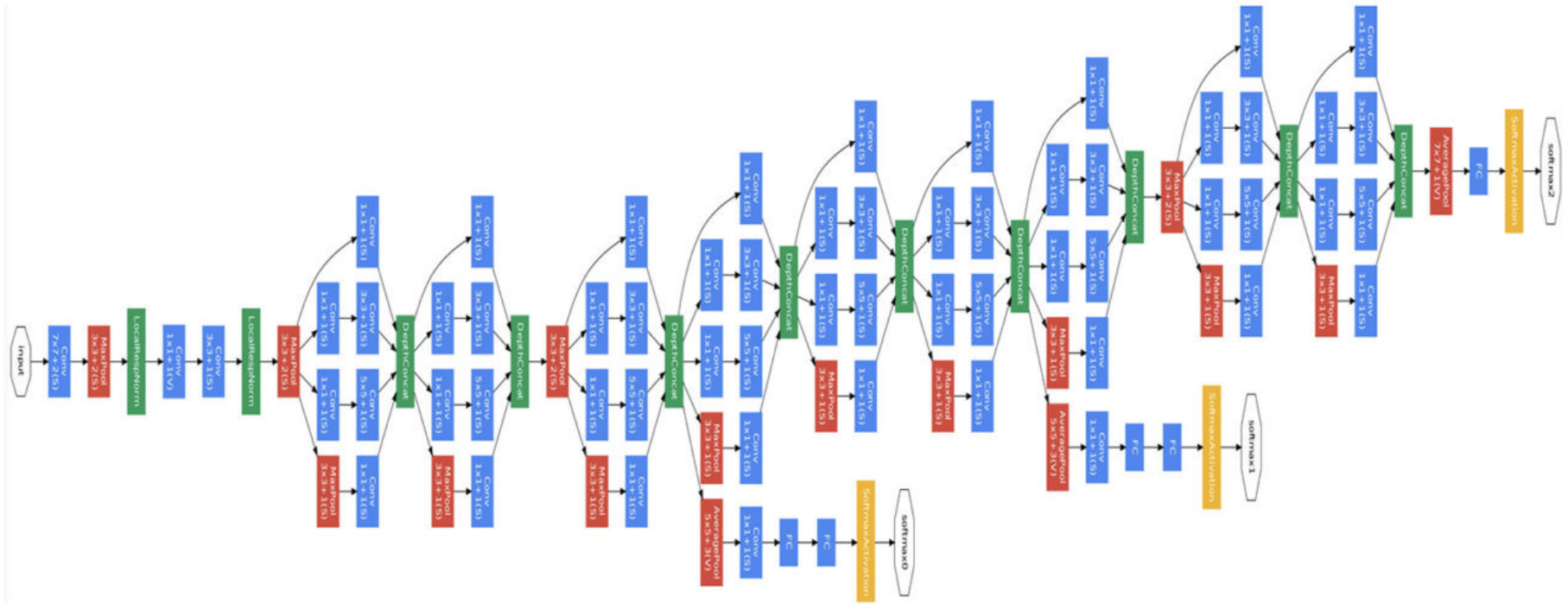


Ajust Learning Rate: Keras

- *With default parameter*
  - model.compile(loss='mean_squared_error', optimizer='sgd')
- **If you want to adapt parameters**
  - **from** keras **import** optimizers
  - sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=**True**)
    - **learning_rate**: float >= 0. Learning rate.
    - **momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
    - **decay:** $lr = init\_lr * \frac{1.0}{1.0 + decay * iterations}$
      - It's also possible to give you own decay function
    - **nesterov**: boolean. Whether to apply Nesterov momentum.
  - model.compile(loss='mean_squared_error', optimizer=sgd)

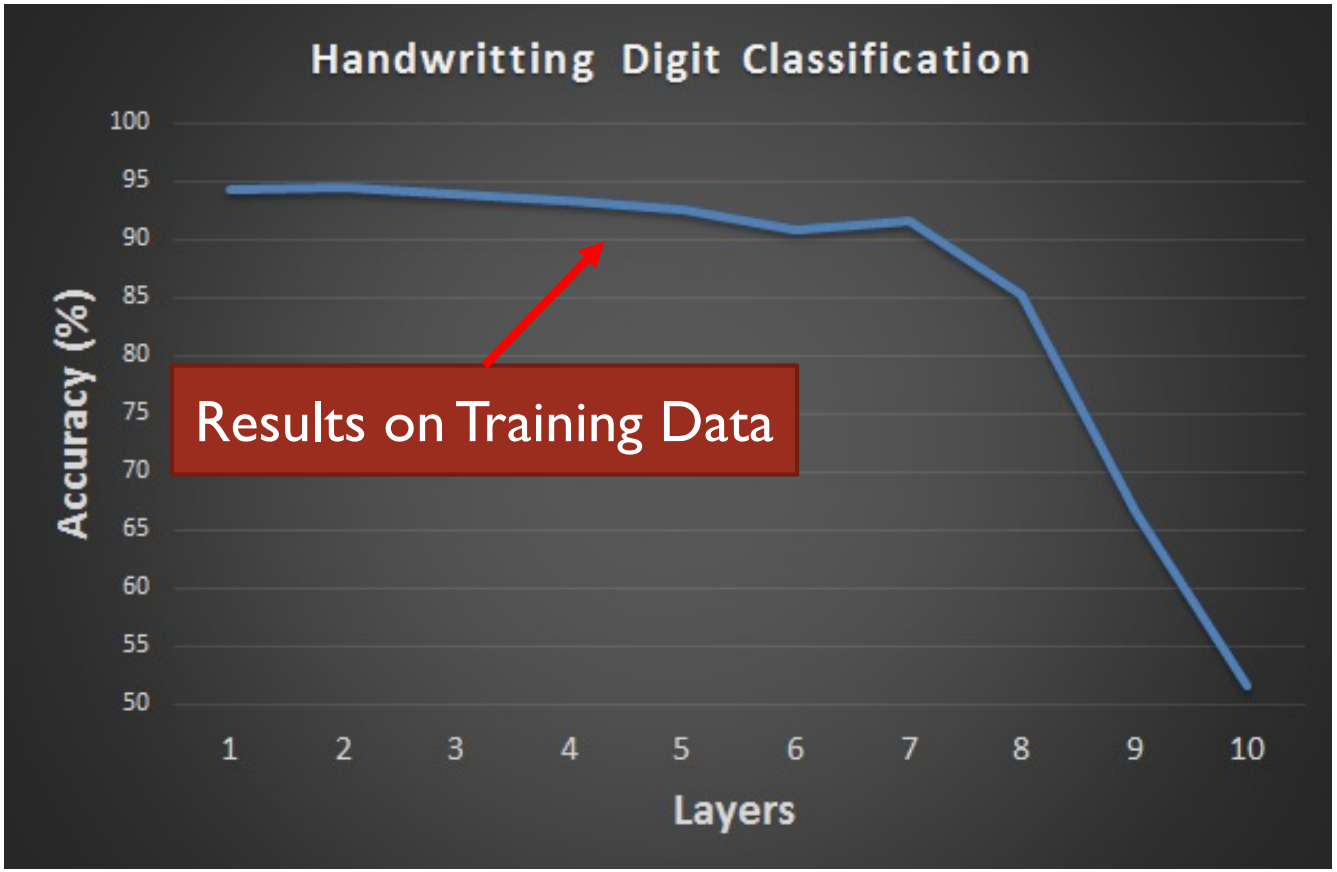
The vanishing gradient problem

Very deep network



Recipe of Deep Learning

- Hard to get the power of Deep …

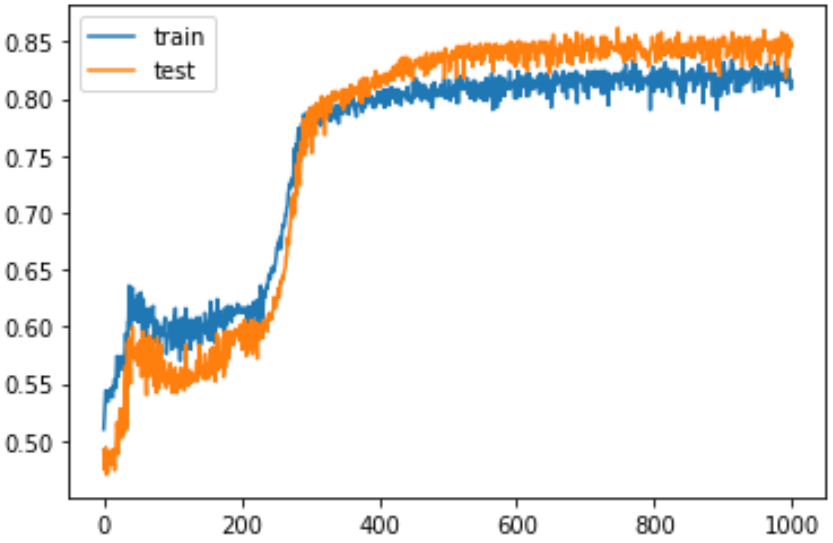


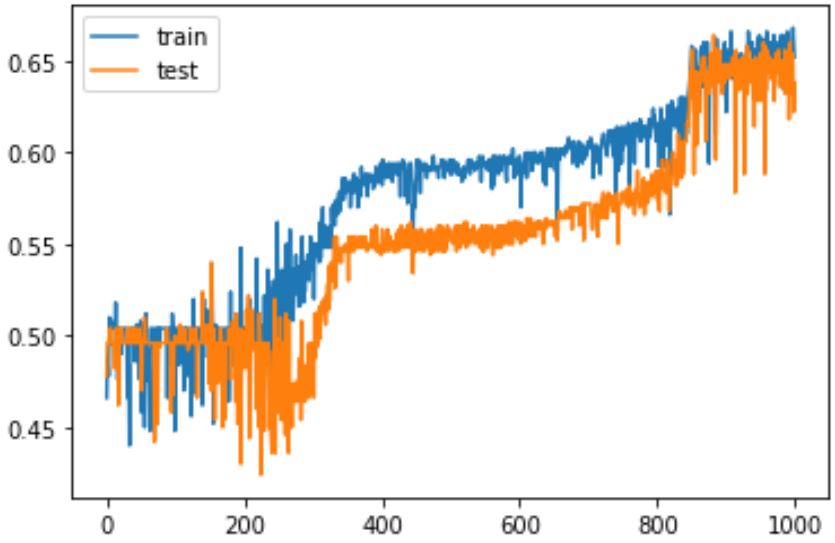
Deeper usually does not imply better.

The problem

- When the gradient becomes negligible, subtracting it from original matrix doesn't makes any sense and hence the model stops learning.

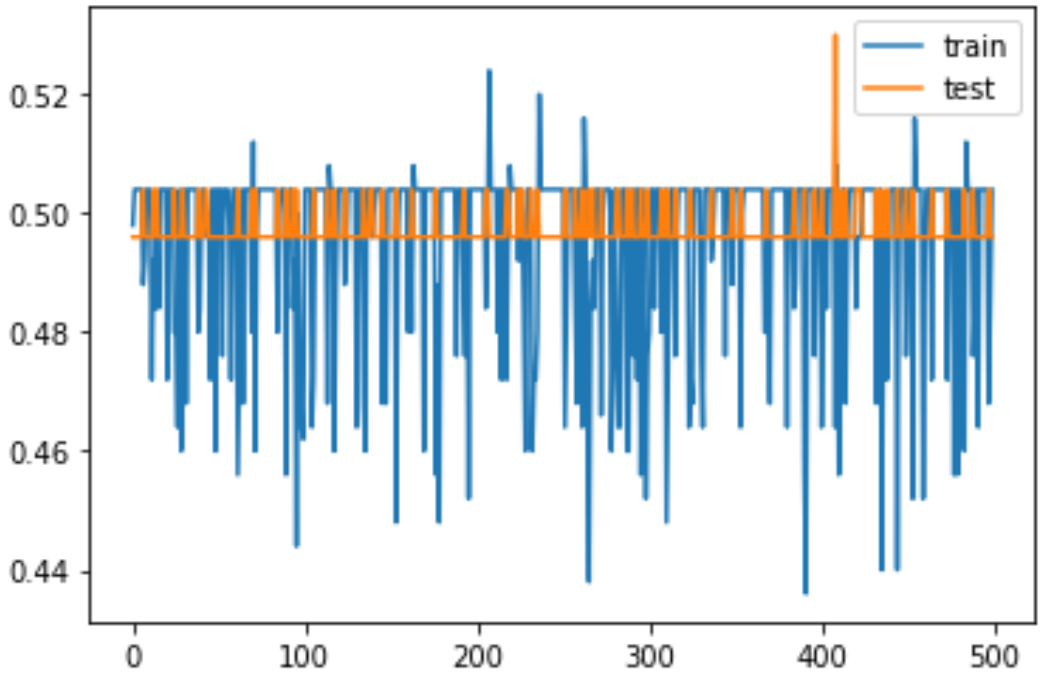
1 hidden layer tanh, random uniform (-0.5, +0.5)	10 hidden layers tanh, random uniform (-0.5, +0.5)





The problem

But for 20 hidden layers
tanh, random uniform (-0.5, +0.5)



The disease of deep networks: vanishing gradient

- Suppose you have a very deep network with several dozens of layers.
- Each layer multiplies the input x with a weight w and sends the result through an activation function.
  - For the sake of simplicity, we will ignore the bias term b.

- Assume the weights are all equal to 0.6. This means with every additional layer; we multiply the weight by itself.

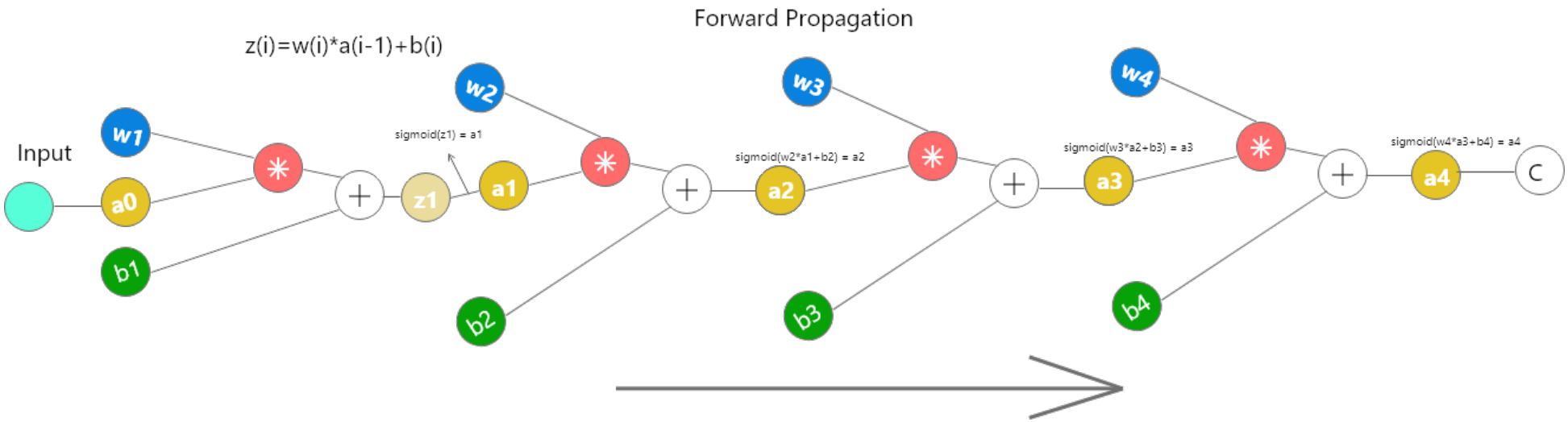
The disease of deep networks: vanishing gradient or exploding gradient

- By the time you get to the third layer, you need to take the weight to the power of 3.
  - $w^3=0.6^3=0.21$
- Now assume you have a network with 15 layers. Now your third weight equals w to the power of 15.
  - $w^{15}=0.6^{15}=0.00047$
- As you can see, the weight is now vanishingly small, and it further shrinks exponentially with every additional layer.

- The reverse is true if your initial weight is larger than 1, let's say 1.6.
  - $w^{15}=1.6^{15}=281.47$

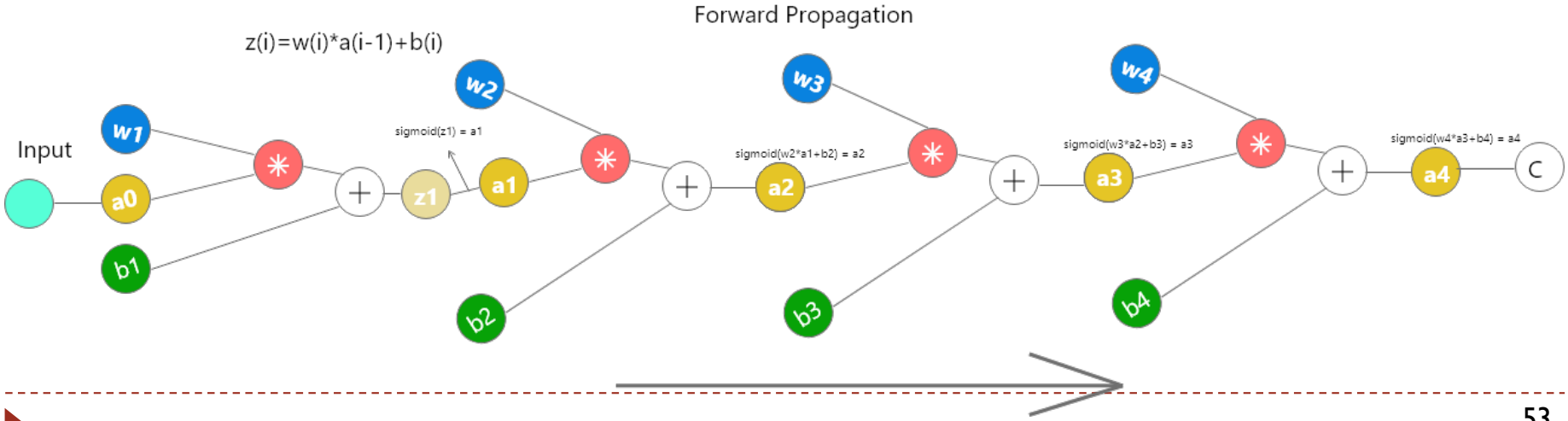
The disease of deep networks: vanishing gradient or exploding gradient

- The previous example was mainly intended to illustrate the principle behind vanishing and exploding gradients
- In practice, it affects the gradients of the non-linear activation functions that are calculated during backpropagation.



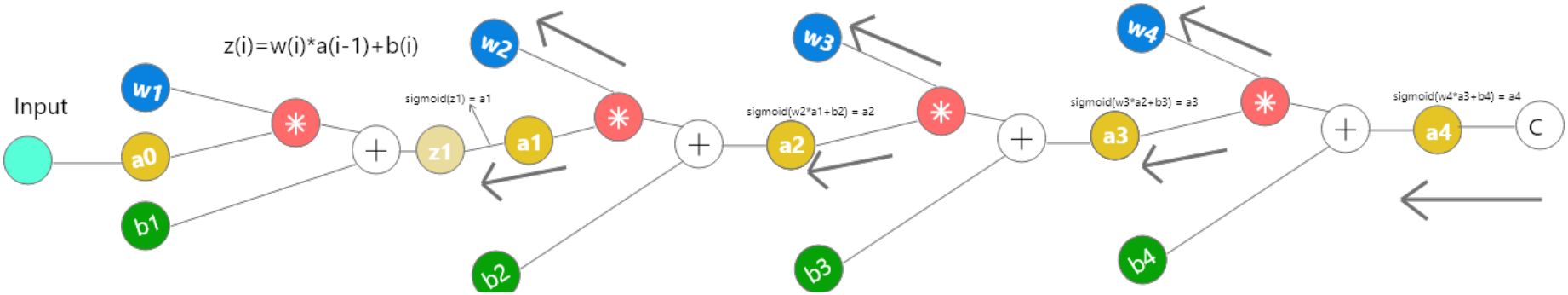
Forward (4 hidden layers)

- Consider a neural network with 4 hidden layers with a single neuron in each matrix.
  - $a1 = \sigma(z1) = \sigma(w1 * a0 + b1)$
  - $a2 = \sigma(z2) = \sigma(w2 * a1 + b2) = \sigma(w2 * \sigma(z1) + b2)$
    $= \sigma(w2 * \sigma(w1 * a0 + b1) + b2)$
  - $a3 = \sigma(z3) = \sigma(w3 * a2 + b3) = \sigma(w3 * \sigma(z2) + b3)$
    $= \sigma(w3 * \sigma(w2 * \sigma(w1 * a0 + b1) + b2) + b3)$
  - $C = a4 = \sigma(z4) = \sigma(w4 * a3 + b3) = \sigma(w4 * \sigma(z3) + b4)$
    $= \sigma(w4 * \sigma(w3 * \sigma(w2 * \sigma(w1 * a0 + b1) + b2) + b3) + b4)$

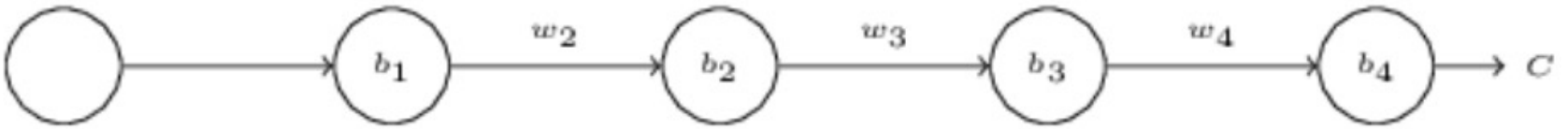


Backpropagation

- During backpropagation, we calculate the derivative of the output with respect to the different weights
- Suppose that we want to modify the matrix of weight w_1
  - $C = \sigma(w4 * \sigma(w3 * \sigma(w2 * \sigma(w1 * a0 + b1) + b2) + b2) + b1)$
  - $\frac{\partial C}{\partial w1} = \sigma'(z1) * w2 * \sigma'(z2) * w3 * \sigma'(z3) * \sigma'(z4) * \frac{\partial C}{\partial a4}$

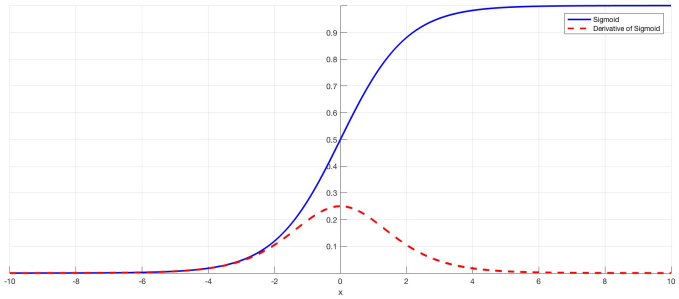


$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



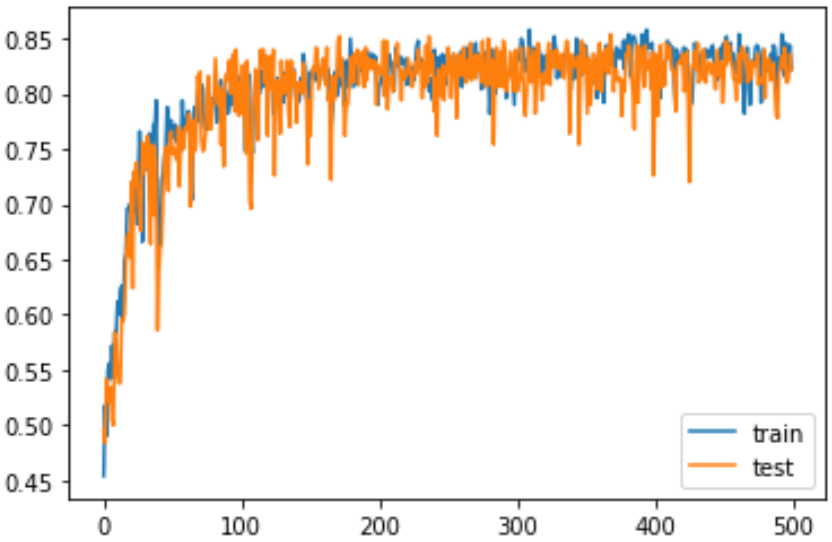
Backpropagation

- Each sigmoid'(z1),sigmoid'(z2).. Etc. are less than ¼
  - Because derivative of sigmoid function is less than ¼
- Each weight matrices w1,w2,w3,w4 are generally initialized using gaussian method to have a mean of 0 and standard deviation of 1.
  - $||w_i|| \leq 1$
- Each term of $\frac{\partial C}{\partial w1}$ are ≤1 or ≤1/4
- Multiplying such small terms for a huge number of times we get very small gradient which makes the model to almost stop learning.
  - $(¼)^2 = 0{,}0625$
  - $(1/4)^4 = 0{,}00390625$
  - $(1/4)^8 = 0{,}00001526$

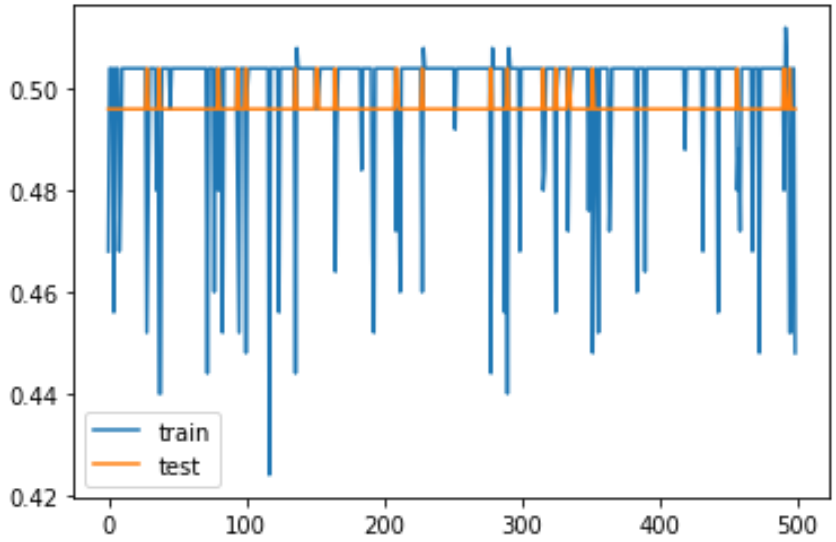


How do you fix « the vanishing gradient problem »

- First step: replace 'sigmoid' by 'relu'
  - With relu use he_uniform as initializer



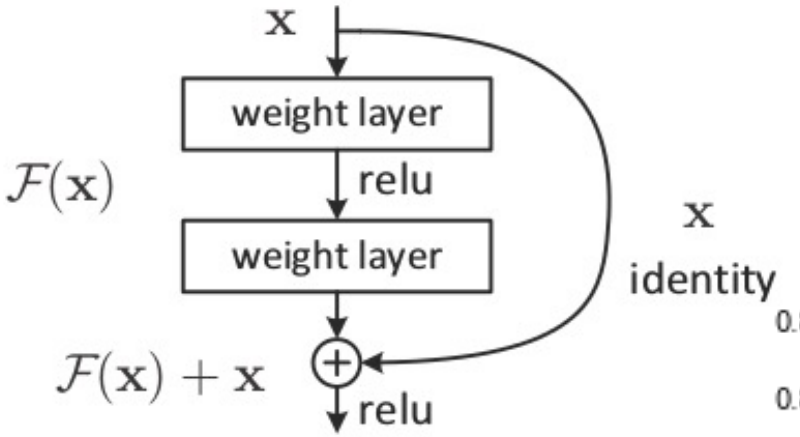
10 hidden layers
relu, he_uniform



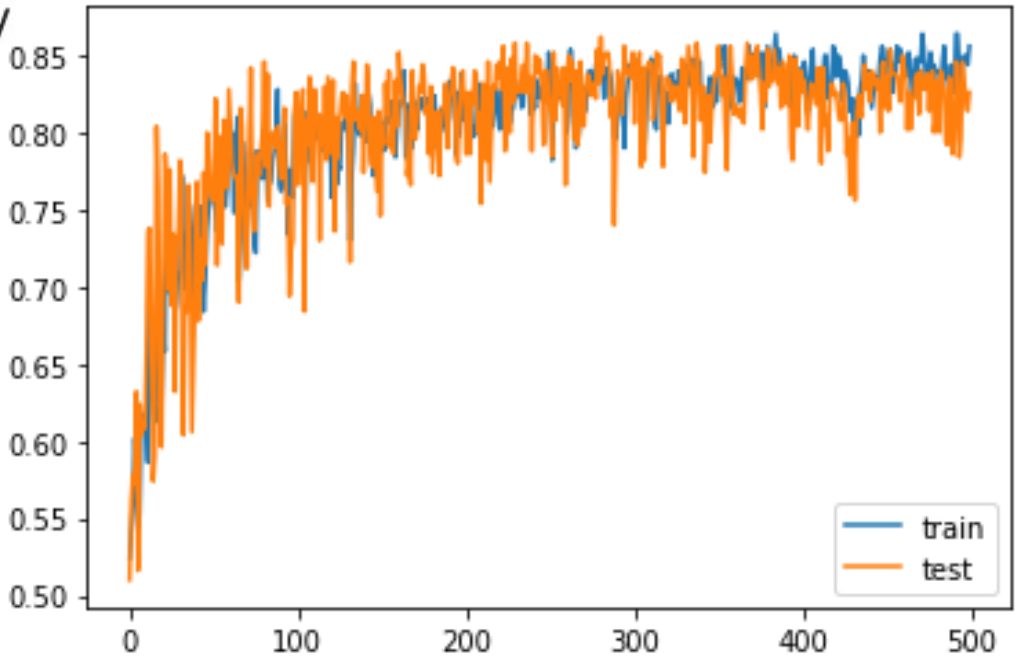
20 hidden layers
relu, he_uniform

How do you fix « the vanishing gradient problem »

- Improve solution: use residual

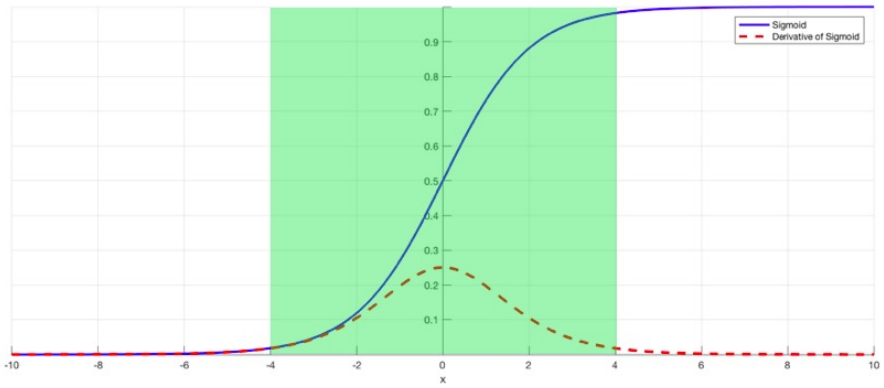


Residual, 20 hidden layers
relu, random_uniform (-0,5, +0,5)

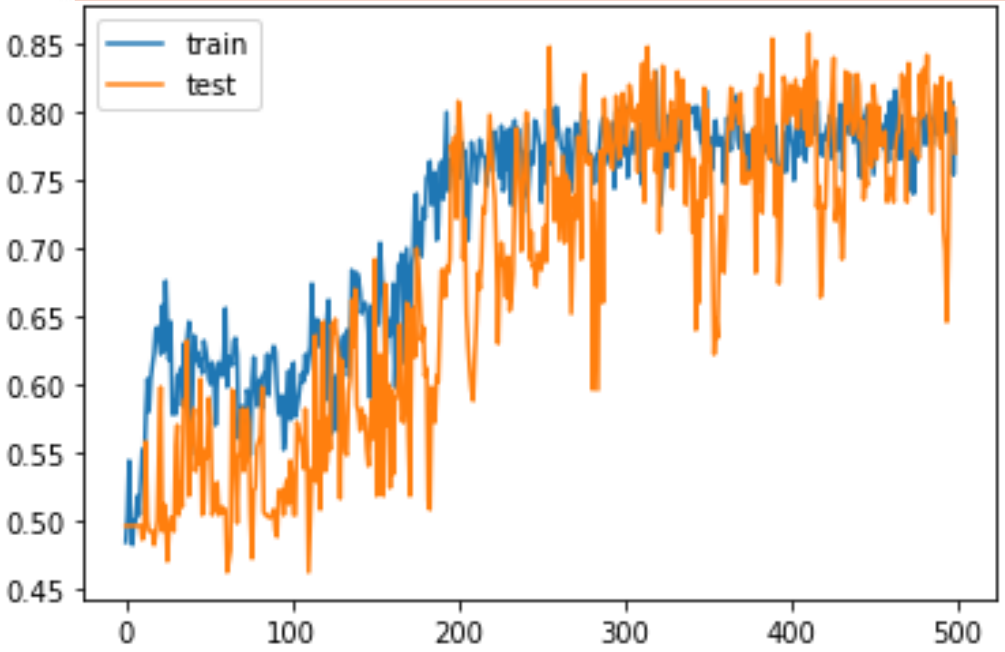


How do you fix « the vanishing gradient problem »

- Improve solution: use batch normalisation

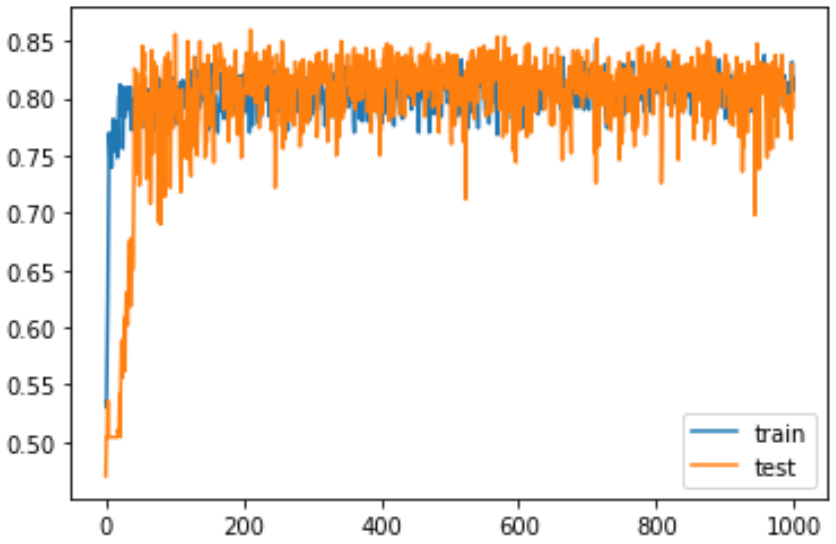


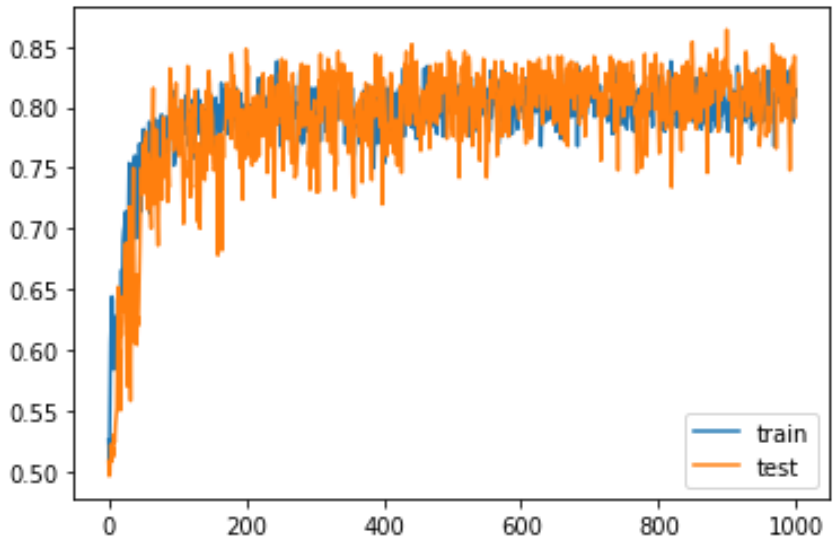
Batch normalization, 20 hidden layers tanh, random_uniform (-0,5, +0,5)



How do you fix « the vanishing gradient problem »

- You can combine the solution : Residual + Batch normalization

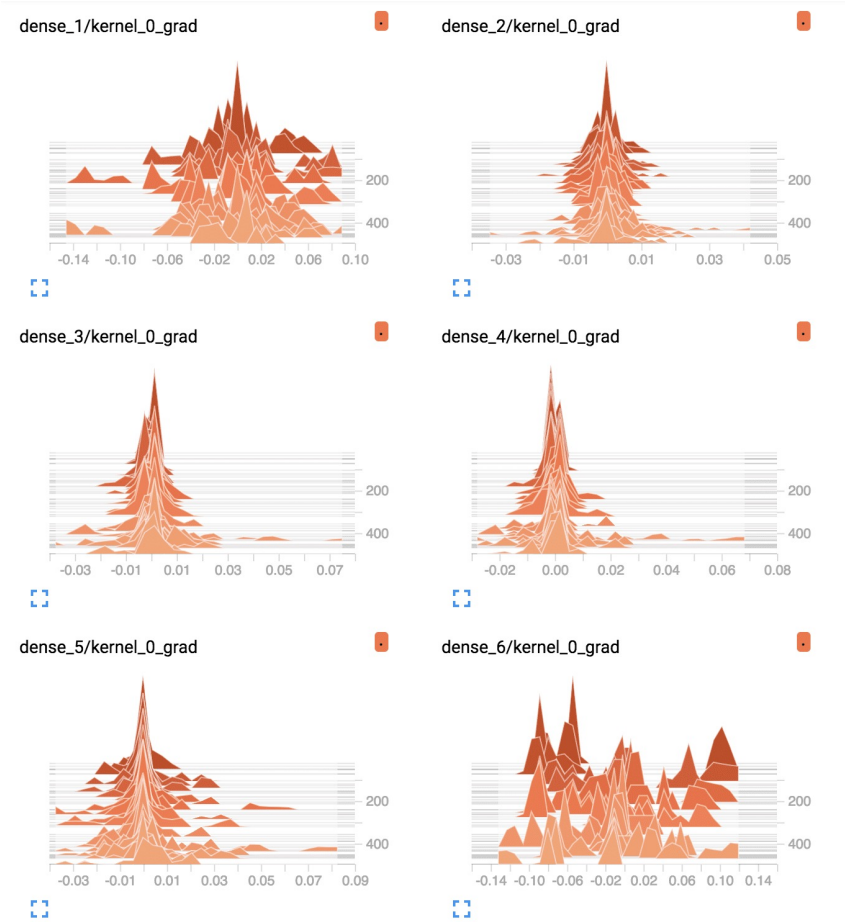




20 hidden layers Residual + Batch normalization tanh, random uniform (-0.5, +0.5)	20 hidden layers Residual + Batch normalization relu, he_uniform

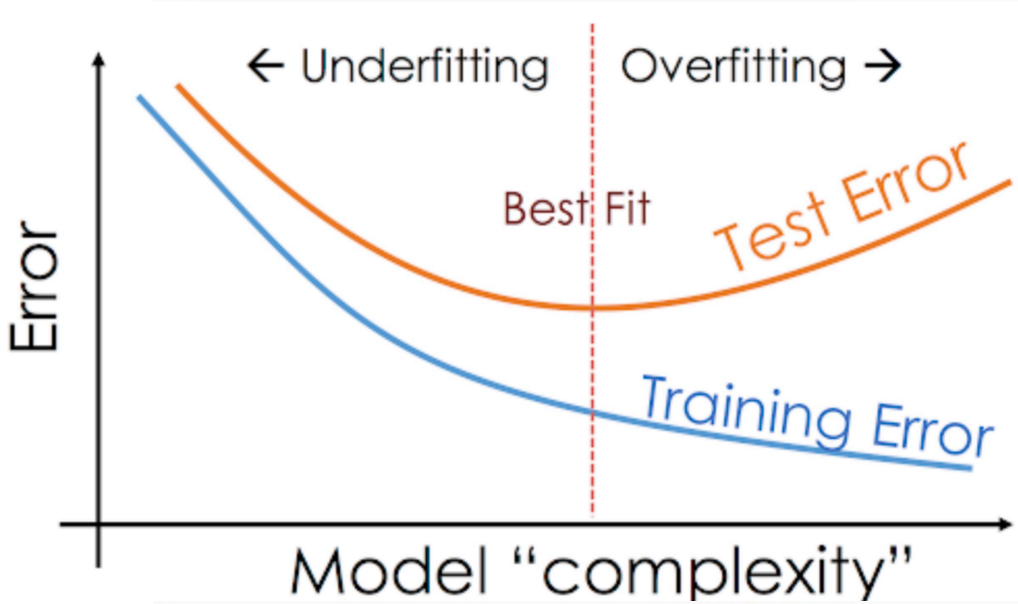
How to solve vanishing gradient problem

- Babysit your neural network (inspect weight and gradient)
- Replace sigmoid by relu
- Adapt weigh initialisation
  - He initialization and Xavier initialization ensure that the weights are close to 1.
- For not deep [1, 3]
  - Tanh with random uniform (-0,5, +0,5)
- For medium deep]3, 8]
  - Relu with he_uniform
- For deep]8, 25]
  - Residual + batch normalization



How to avoid overfitting

Babysit your network!

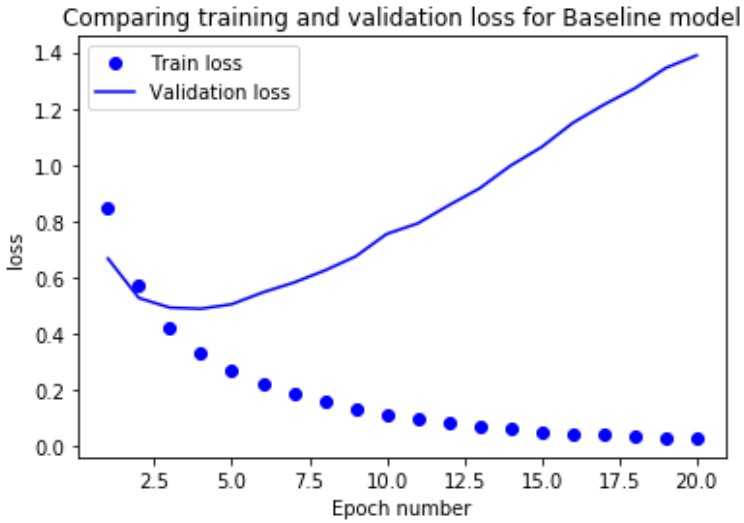


babysitting your deep network

1. overfitting and underfitting
2. check accuracy before training [Saxe et al., 2011]
3. Y. Bengio : "*check if the model is powerful enough to overfit, if not then change model structure or make model larger*"

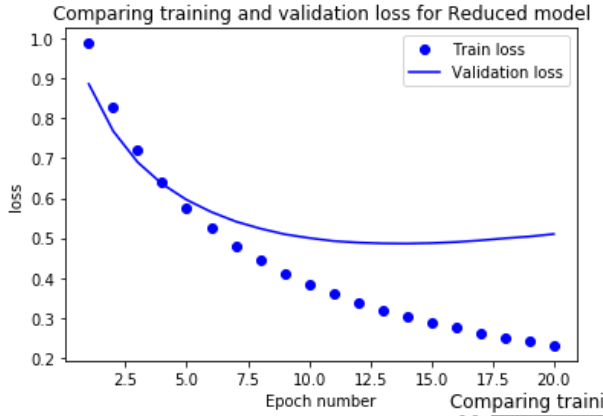
Overfitting in neural networks

- Neural Networks are especially prone to overfitting
- Zero error is possible, but so is more extreme overfitting

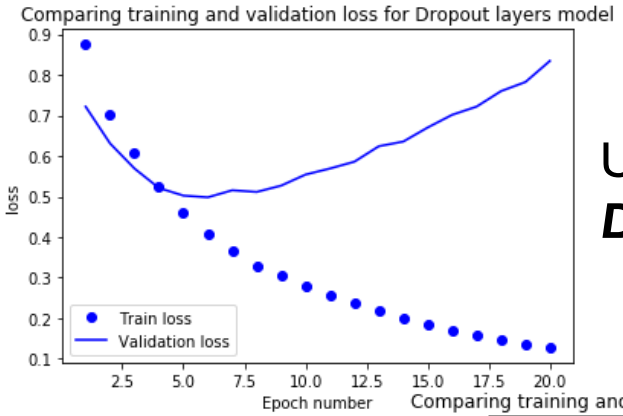


- **Handling overfitting**
  - ***Reduce the network's capacity*** by removing layers or reducing the number of elements in the hidden layers
  - Apply ***regularization***, which comes down to adding a cost to the loss function for large weights
  - Use ***Dropout layers*** *(add noise to data)*, which will randomly remove certain features by setting them to zero

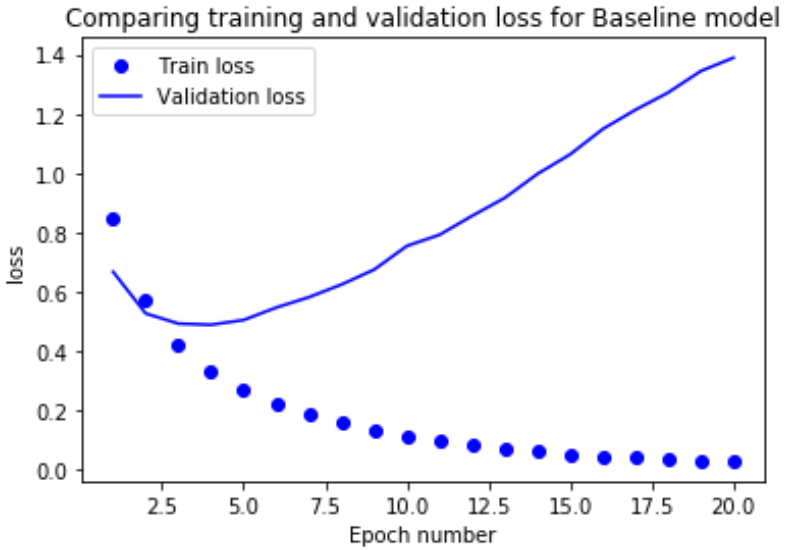
Overfitting in neural networks



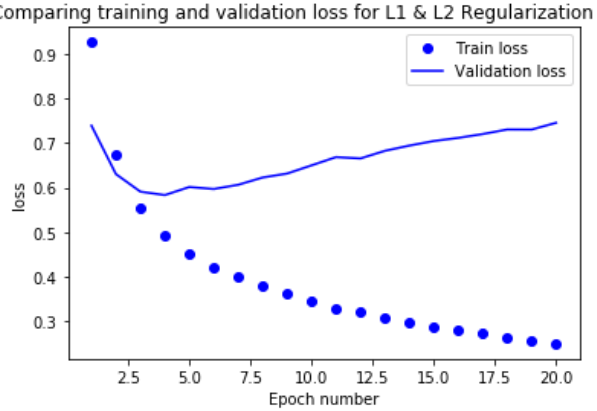
Reduce
network's capacity



Use
Dropout layers



Apply
regularization

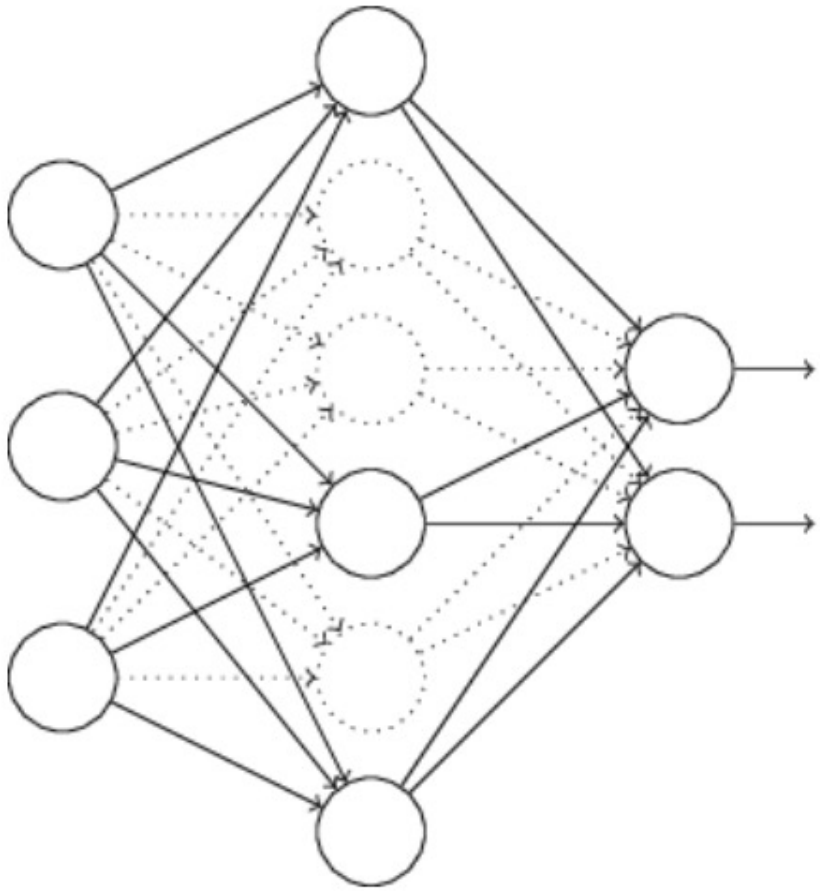


Add dropout

- The networks are usually initialized with random weights.
  - At each training ➔ the result is slightly different.
- Why random weights?
  - In order to have different weights
  - If the weights are all equal, the response of the filters will be equivalent.
    - The network does not train
- Why not train 5 different networks with random starts and vote on their outcome?
  - That's pretty much what dropout does
  - This works well!
  - It helps with generalization because the errors are averaged.

Add dropout

- At each mini-batch:
  - Randomly select a subset of neurons.
  - Ignore them.
- On test: part of weights outgoing to compensate for training on part of neurons.
- Effect:
  - Neurons become less dependent on output of connected neurons.
  - Forces network to learn more robust features that are useful to more subsets of neurons.
  - Like averaging over many different trained networks with different random initializations.
  - Except cheaper to train.

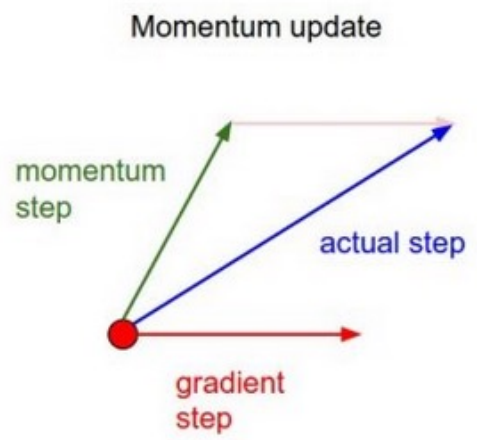


Add regularisation

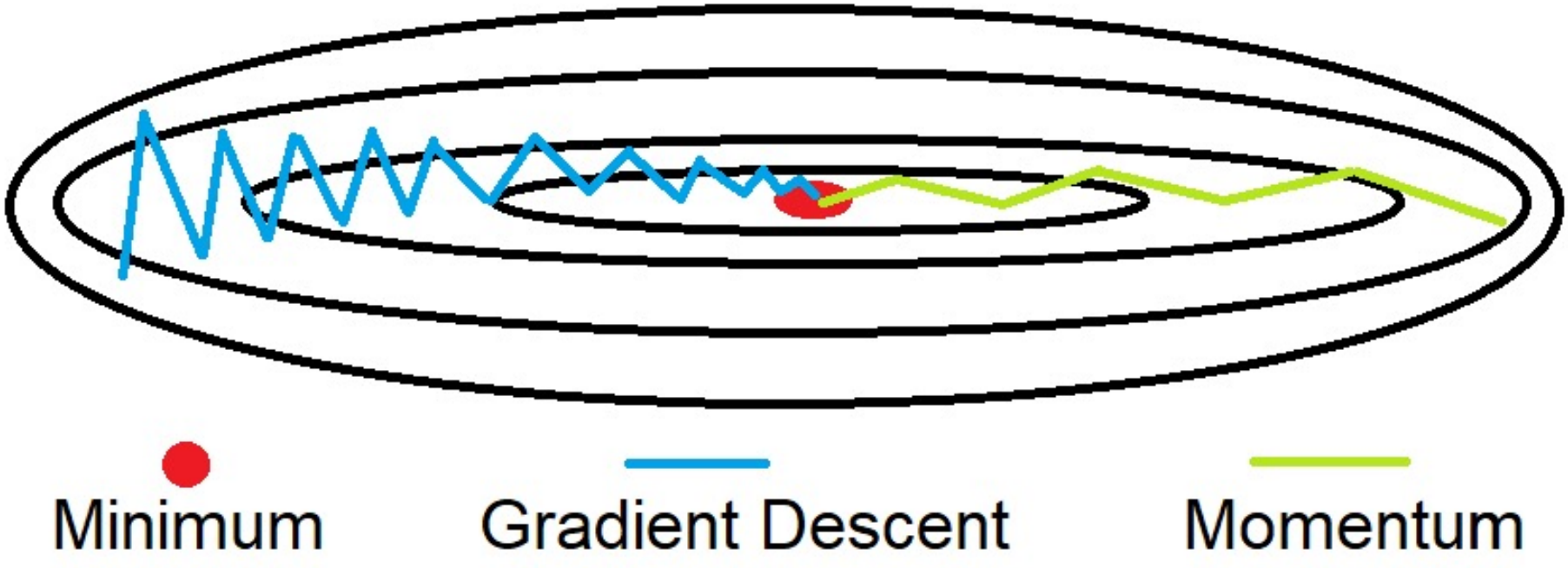
- Regularization, in general, is a mechanism that penalizes model complexity
  - Add a term to the loss function that represents model complexity.
- L2 regularization
  - Penalize the large weights: indicate over fitting to the training data.
  - $J_{new}(X, y) = J(X, y) + \lambda \sum \sum w_{ij}^2$
- Momentum regularization
  - Adds a fraction of the past weight update to the current weight update
  - Prevent the model from getting stuck in local minima, even if the current gradient is 0
  - $W_{t+1} = W_t - \beta \nabla W - \alpha \nabla J$

Momentum
(past weight update)

Gradient cost function



Add momentum

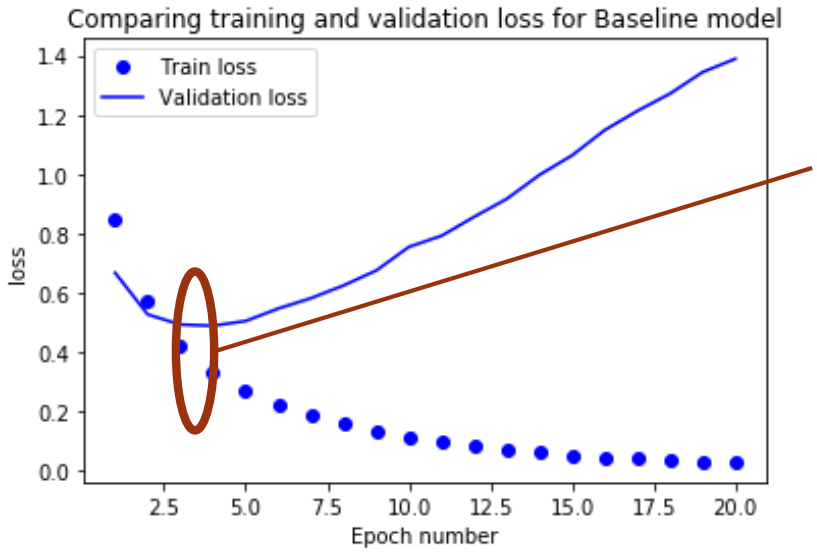


How to add regularisation in Keras

- Dropout
  - Add output layer or fixed dropout on each layer
  - The default interpretation of the dropout hyperparameter is the probability of training a given node in a layer, where 1.0 means no dropout, and 0.0 means no outputs from the layer.
  - A good value for dropout in a hidden layer is between 0.5 and 0.8.
- L1+L2 regularization
  - Add regularizer on layer
  - regularizers = tf.keras.regularizers.l1() or tf.keras.regularizers.l2()
- Momentum
  - Add momentum to the optimizer
  - Momentum paramenter of SGD

How to stop learning

- **Early stopping** is a form of regularization used to avoid overfitting
  - the main idea is to continue training as long as the generalization of the model persists
  - and to stop training as soon as the learner's adjustment to the training data is made at the expense of an increased generalization error



Try to stop learning in this area

How to stop learning: Keras

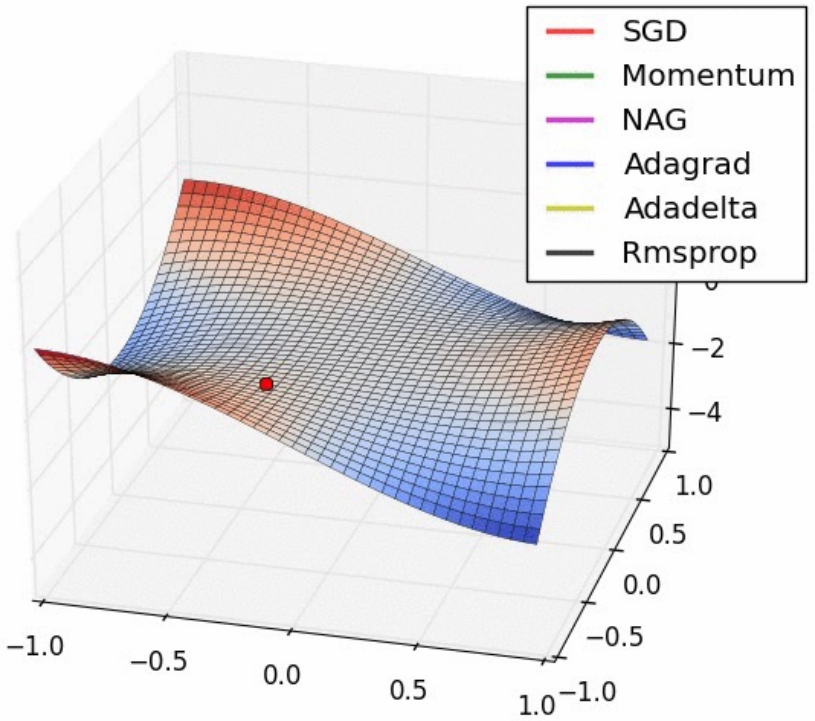
- In Keras use callback
  - es = EarlyStopping(monitor='val_accuracy', mode='max', min_delta=0.001)
    - monitor: Quantity to be monitored.
    - min_delta: Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
    - patience: Number of epochs with no improvement after which training will be stopped.
    - mode: One of {"auto", "min", "max"}.
      - In min mode, training will stop when the quantity monitored has stopped decreasing;
      - In max mode it will stop when the quantity monitored has stopped increasing;
      - In auto mode, the direction is automatically inferred from the name of the monitored quantity.
  - mc = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min', save_best_only=True)
  - model.fit(…, epochs=4000, callbacks=[es, mc])


How to choose
Optimizer

Choose optimizer algorithms

- It is crucial to train deep learning model trains in a shorter time without penalizing the accuracy
  - Several optimizers have been proposed
  - They differ in the way the weights are modified

- Some optimizer
  - SGD (Stochastic Gradien Descent)
  - RMSProp
  - SGD with Momentum
  - Nesterow Adaptative Gradient
  - Adam



Batch gradient descent (SGD)

for i in range$(Number\ of\ training\ steps)$:

Forward Propagation using X to calculate y'

Calculate cost using Y

Backward Propagation to calculate derivative dW and dB

Parameter Updation using following rule:

$$W = W - \alpha.dW$$

$$b = b - \alpha.db$$

Mini-Batch Gradient Descent (SGD)

for i in range$(Number\ of\ training\ steps)$:
  $batches = miniBatchGenerator(X, Y,$ batch_size$)$
  for j in range$(Number\ of\ batches)$:
    $minibatchX, minibatchY = batches[j]$
    Forward Propagation using $minibatchX$ to calculate y'
    Calculate cost using $minibatchY$
    Backward Propagation to calculate derivative dW and dB
    Parameter Updation using following rule:
      $W = W - \alpha.dW$
      $b = b - \alpha.db$

Momentum

for i in range$(Number\ of\ training\ steps)$:
  $batches = miniBatchGenerator(X, Y, \text{batch\_size})$
  for j in range$(Number\ of\ batches)$:
    $minibatchX, minibatchY = batches[j]$
    Forward Propagation using $minibatchX$ to calculate y'
    Calculate cost using $minibatchY$
    Backward Propagation to calculate derivative dW and dB
    Parameter Updation using following rule:

$$V_{dW} = \beta V_{dW} + (1 - \beta) dW$$
$$V_{db} = \beta V_{db} + (1 - \beta) db$$
$$W = W - \alpha . V_{dW}$$
$$b = b - \alpha . V_{db}$$

RMSprop

Root-Mean-Square Propagation

Similar to momentum, it is a technique to dampen out the motion for the weigths/bias

for i in range$(Number\ of\ training\ steps)$:

$batches = miniBatchGenerator(X, Y, \text{batch\_size})$

for j in range$(Number\ of\ batches)$:

$minibatchX, minibatchY = batches[j]$

Forward Propagation using $minibatchX$ to calculate y'

Calculate cost using $minibatchY$

Backward Propagation to calculate derivative dW and dB

Parameter Updation using following rule:

$$S_{dW} = \beta S_{dW} + (1 - \beta) dW^2$$

$$S_{db} = \beta S_{db} + (1 - \beta) db^2$$

$$W = W - \alpha . \frac{dW}{\sqrt{S_{dW}} + \epsilon}$$

$$b = b - \alpha . \frac{db}{\sqrt{S_{db}} + \epsilon}$$

AdaM
Adaptive Momentum

Combines the Momentum and RMS prop in a single approach making AdaM a very powerful and fast optimizer

for i in range($Number\ of\ training\ steps$) :

$batches = miniBatchGenerator(X, Y, \text{batch\_size})$

for j in range($Number\ of\ batches$) :

$minibatchX, minibatchY = batches[j]$

Forward Propagation using $minibatchX$ to calculate y'

Calculate cost using $minibatchY$

Backward Propagation to calculate derivative dW and dB

Parameter Updation using following rule:

$$V_{dW} = \beta_1 V_{db} + (1 - \beta_1) dW ; V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$$

$$V_{dW}^{corrected} = \frac{V_{dW}}{1 - \beta_1^i} ; V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^i}$$

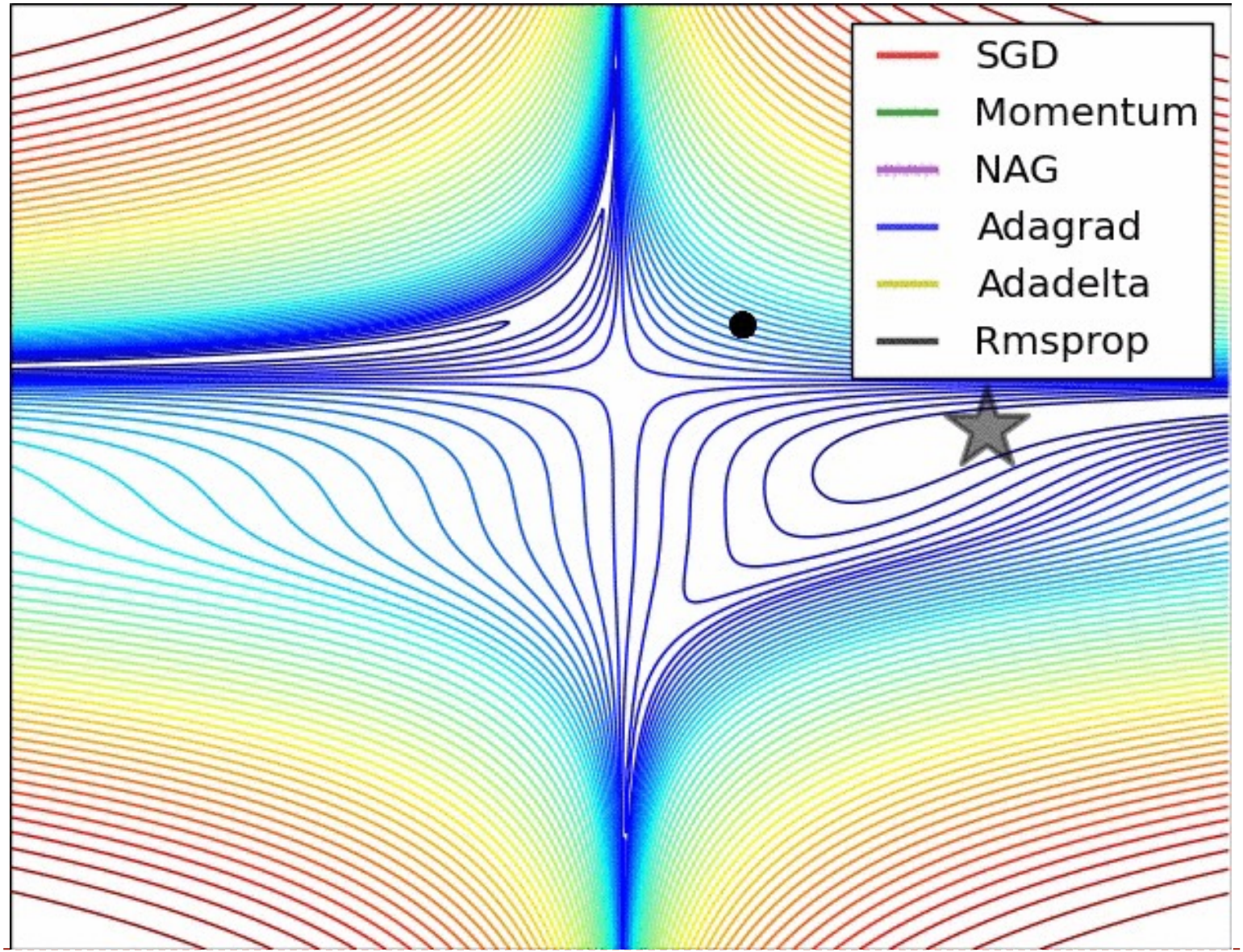
$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2 ; S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$$

$$S_{dW}^{corrected} = \frac{S_{dW}}{1 - \beta_2^i} ; S_{db}^{corrected} = \frac{S_{db}}{1 - \beta_2^i}$$

$$W = W - \alpha . \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \epsilon}$$

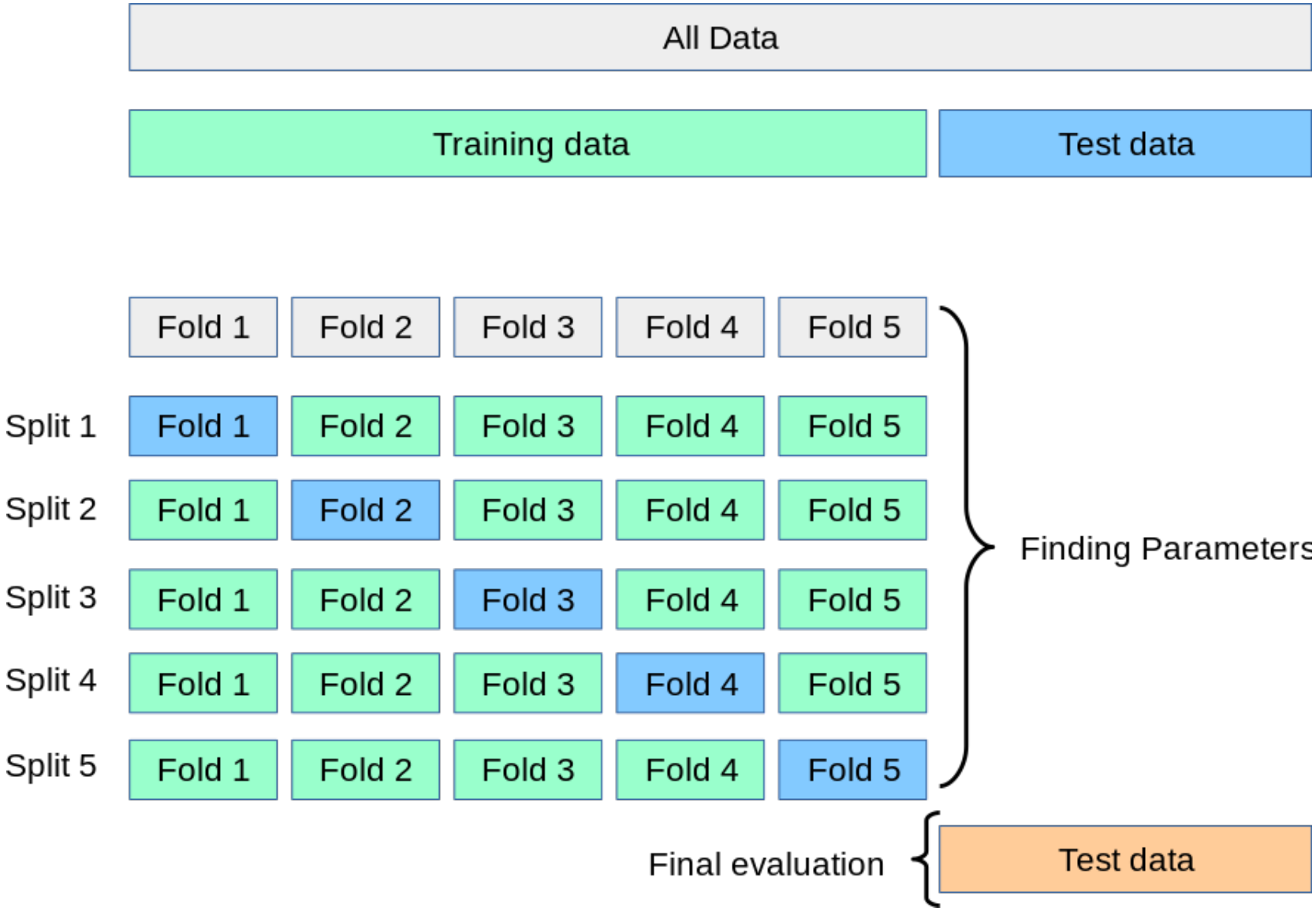
$$b = b - \alpha . \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$$

Performance Comparison



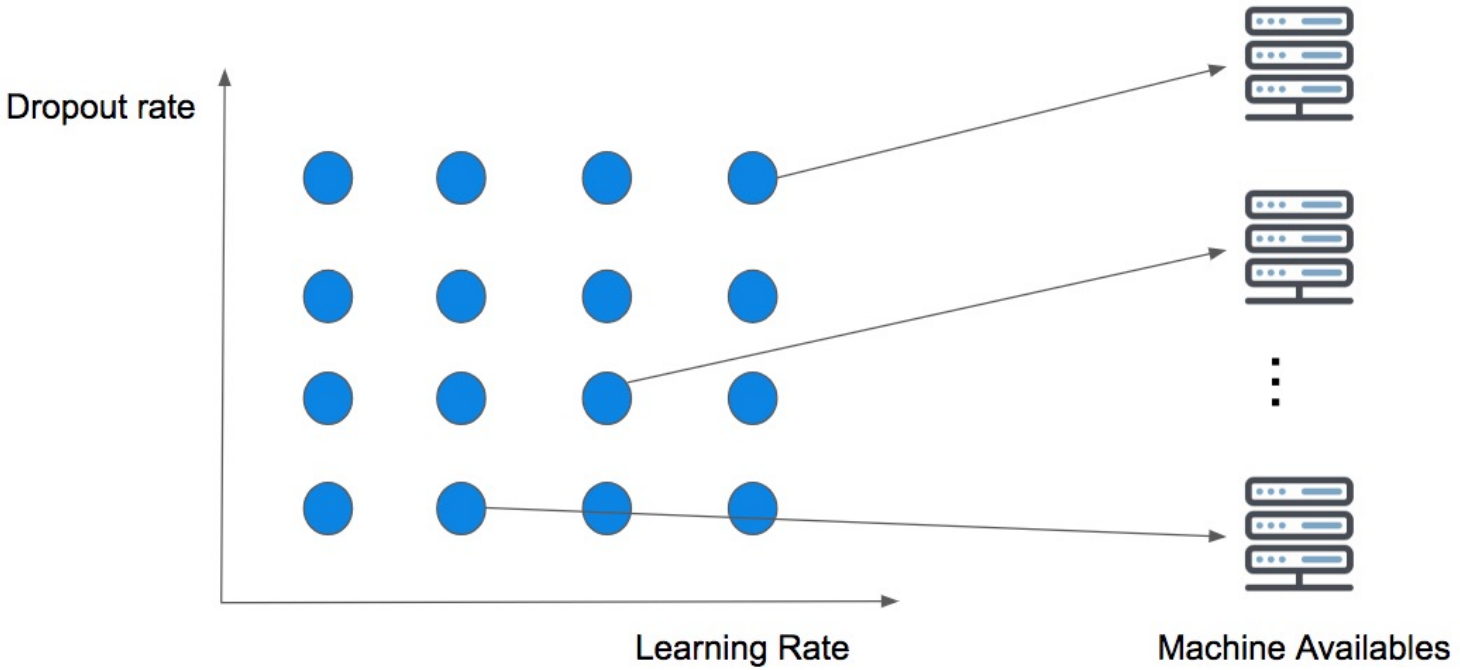
How to find the best hyper-parameters Keras-Tuner

Cross validation



GridSearch

This strategy is embarrassingly parallel because it doesn't take into account the computation history. But what it does mean is that the more computational resources you have available, then the more guesses you can try at the same time!



The real pain point of this approach is known as the curse of dimensionality. This means that more dimensions we add, the more the search will explode in time complexity (usually by an exponential factor), ultimately making this strategy unfeasible!

RandomSearch

Nice tutorial: https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/

based on sklearn GridSearch or RandomSearch

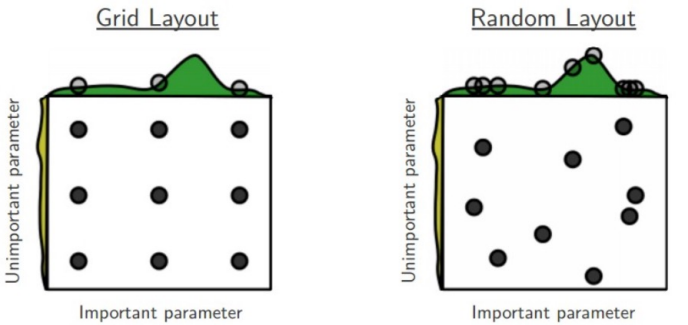


Image from http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf

Grid

- Bad on high spaces
+ It will find the best (but with high cost!)

Random

- It doesn't guarantee to find the best hyperparameters
+ Good on high spaces
+ Give better results (w.r.t. Grid) in less iterations

Keras tuner

```
def build_model(hp):                                    # Wrap a model in a function
    NUM_LAYERS = hp.Int("num_layers", 1, 3)             # Define hyper-parameters
    NUM_DIMS = hp.Int("num_dims", min_value=32, max_value=128, step=32)
    ACTIVATION = hp.Choice("activation", ["relu", "tanh"])
    DROPOUT = hp.Boolean("dropout")
    DROP_RATE = hp.Choice("drop_rate", values=[0.2, 0.25, 0.5])

    text_input = Input(shape=(input_dim,), name='input')  # replace static value
    h = text_input                                        # with hyper-parameters
    for i in range(NUM_LAYERS):
        h = Dense(NUM_DIMS//(2*i+1), activation=ACTIVATION)(h)
        if DROPOUT:
            h = Dropout(rate=DROP_RATE)(h)
    ouputs = Dense(output_dim, activation='softmax', name='output')(h)

    model.compile(
        optimizer='adam',
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

Keras Tuner

```
# Build tuner → Grid, Random, Bayesan, Hyperban
tuner = kt.BayesianOptimization(build_model,
                objective="val_accuracy",
                max_trials=10,
                overwrite=True,
                directory='my_dir',
                project_name='BOW_MLP')

# Search for hyper-parameters
ea = EarlyStopping(monitor='val_accuracy', mode='max',
       patience=2, restore_best_weights=True)
tuner.search(X, y, epochs=10,
                validation_split=0.1,
                callbacks=[ea])
```

Keras Tuner

```
# grab the best hyperparameters
bestHP = tuner.get_best_hyperparameters()

# build the best model and train it
model = tuner.hypermodel.build(bestHP)
H = model.fit(trainX, trainY,
        validation_split=0.1, callbacks=[es])

# evaluate the network
predictions = model.predict(testX)
print(classification_report(testY.argmax(axis=1),
        predictions.argmax(axis=1)))

# Babysit
pd.DataFrame({'val_accuracy':H.history['val_accuracy'],
              'accuracy':H.history['accuracy']}).plot()
```