Introduction to Machine Learning
Hyper-parameter in neural networks

Michel.RIVEILL@univ-cotedazur.fr

Main parameters for a neural network

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

- Exercice sur le playground.

Network architecture

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 and Metrics

- 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
    - Loss: MAE, MSE, RMSE, etc.
    - Metrics: generally the same
      - A metric is used to judge the performance of your model.
      - This is only for you to look at (or evaluate the model) and has nothing to do with the optimization process.
  - Classification
    - Activation: softmax
    - Loss: cross entropy
      - Binary cross entropy: $-\sum\ (yi\log(\widehat{y}_i) + (1-yi)\log(1-\widehat{y}_i))$
      - Categorical cross entropy: $-\sum y_i(\log(\widehat{y}_i)$
      - Interpret the result as a conditional probability: Log(P(x=c|y))
    - Metrics: 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)

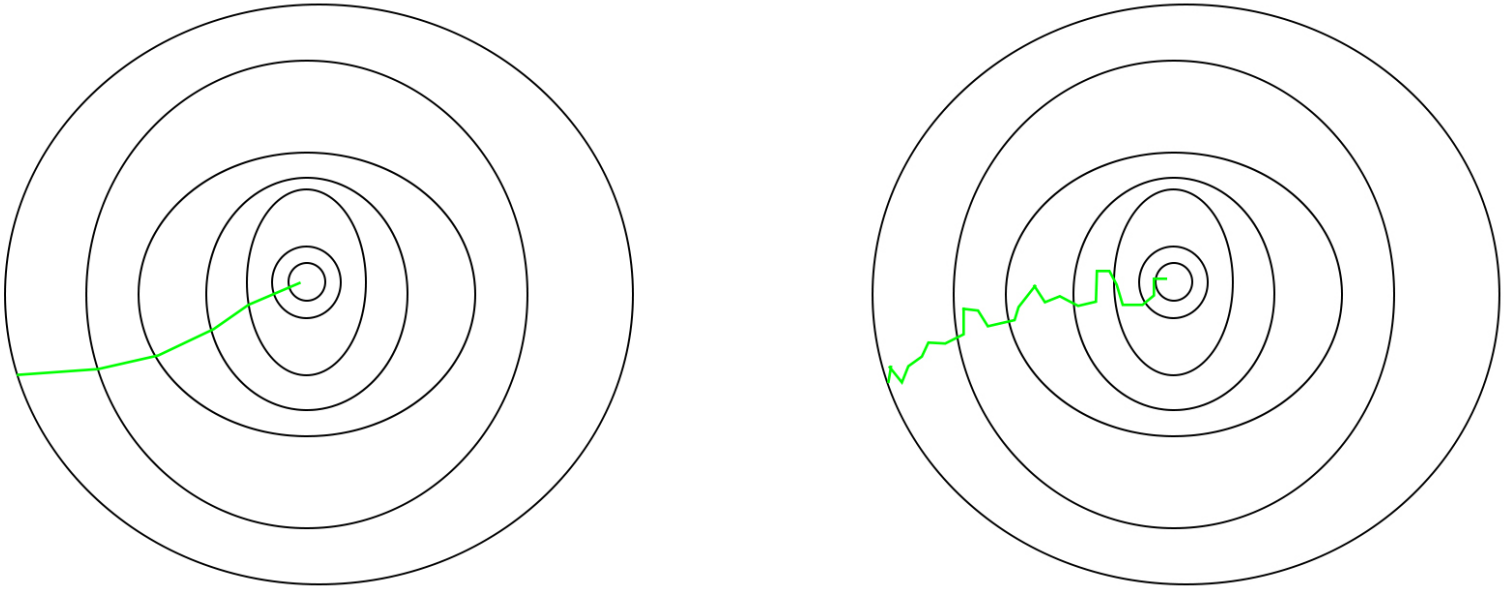
How to choose batch size and learning rate

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

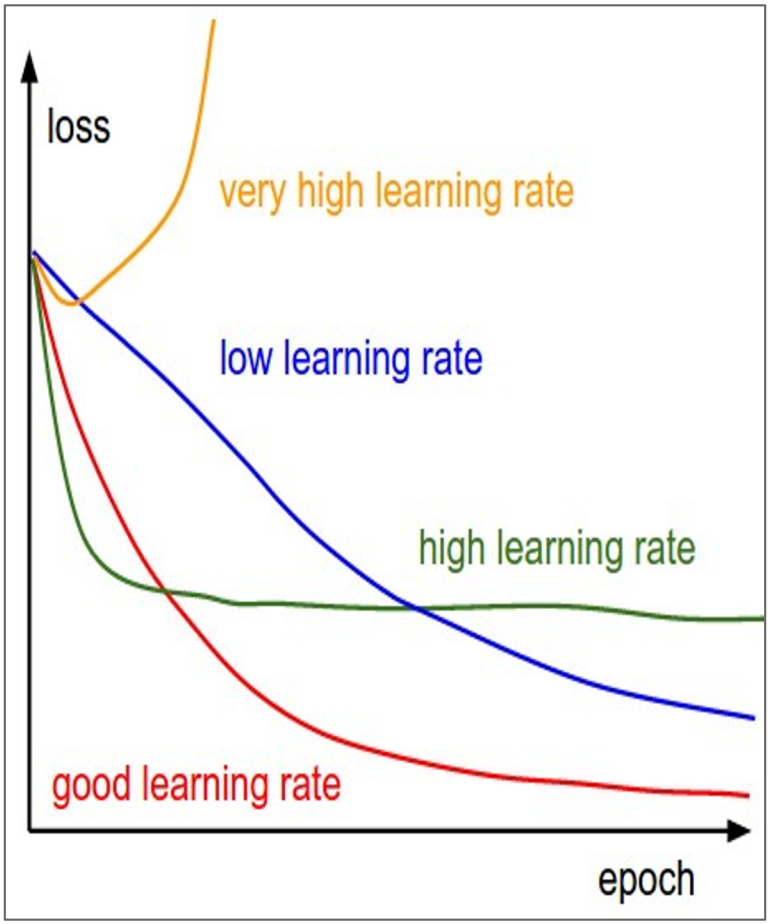
Ajust Learning Rate

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



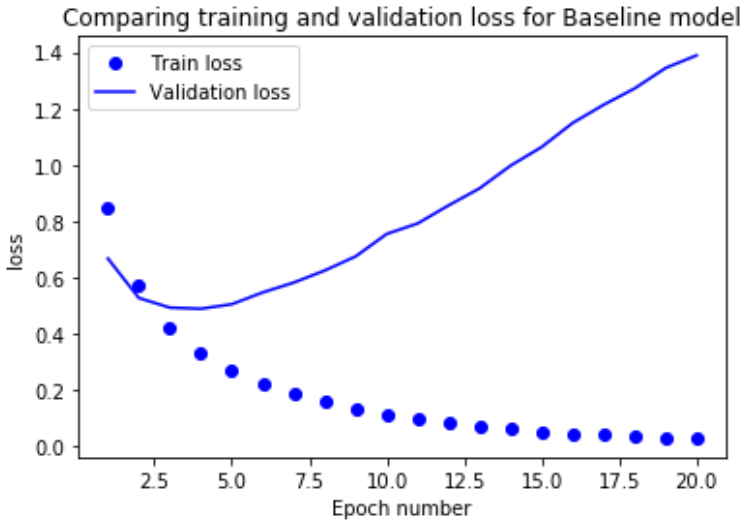
Ajust Learning Rate: Keras

- *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)

How to avoid overfitting

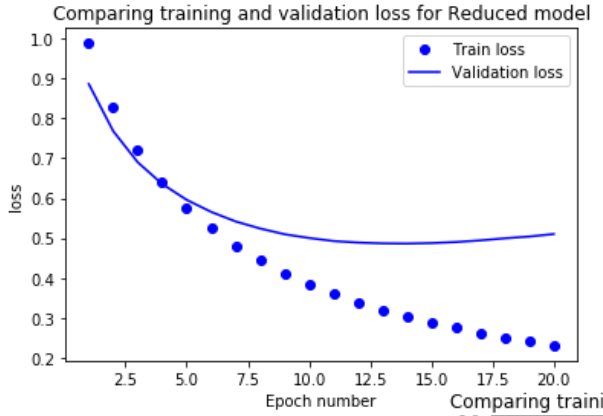
Overfitting in neural networks

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

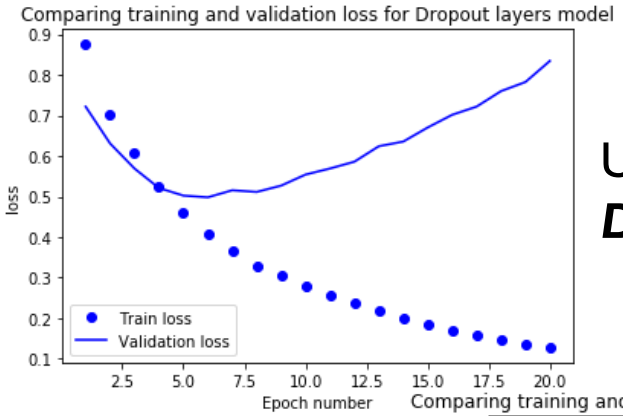


- **Handling overfitting**
  - ***Reduce the network's capacity*** by removing layers or reducing the number of elements in the hidden layers
  - Apply ***regularization***, which comes down to adding a cost to the loss function for large weights
  - Use ***Dropout layers*** *(add noise to data)*, which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

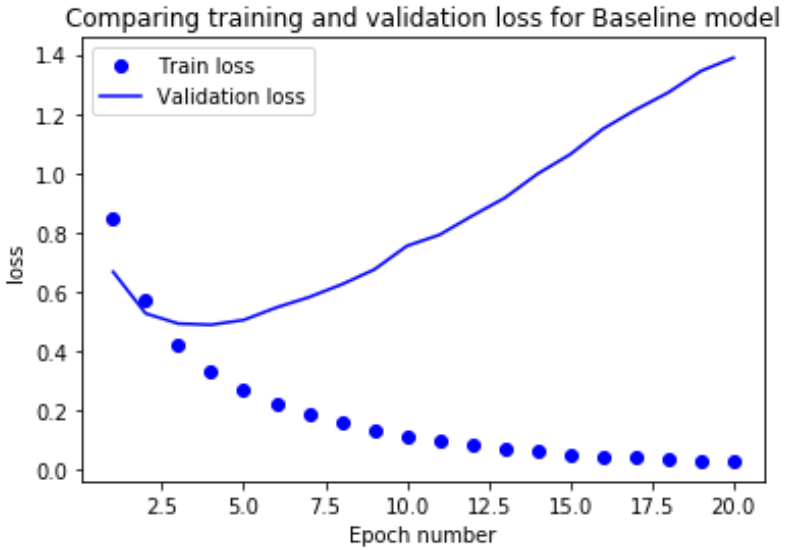
Overfitting in neural networks



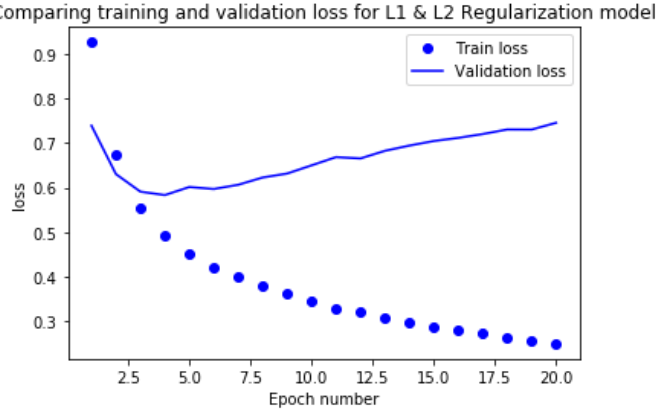
Reduce
network's capacity



Use
Dropout layers



Apply
regularization

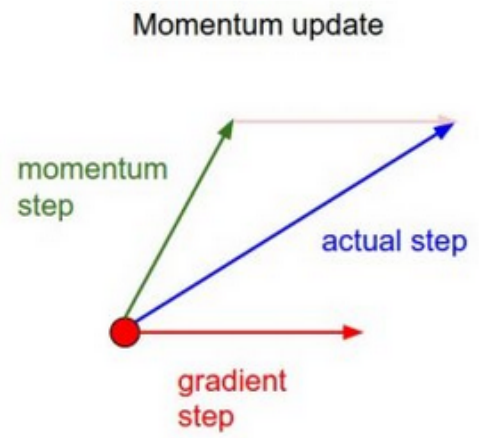


Regularisation

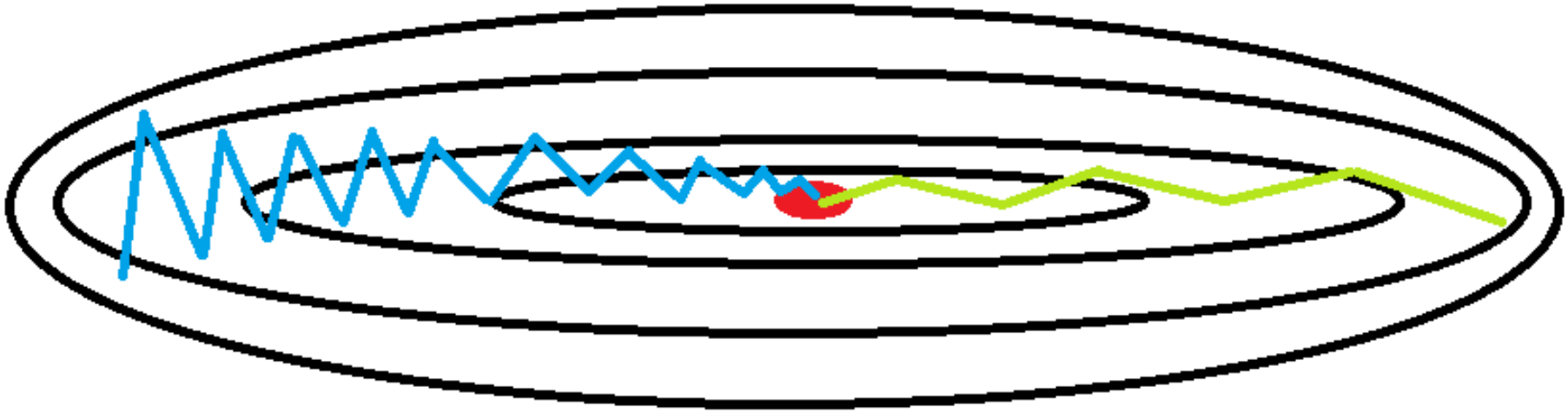
- Regularization, in general, is a mechanism that penalizes model complexity
  - Add a term to the loss function that represents model complexity.
- L2 regularization
  - Penalize the large weights: indicate over fitting to the training data.
  - $J_{new}(X, y) = J(X, y) + \lambda \sum \sum w_{ij}^2$
- 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



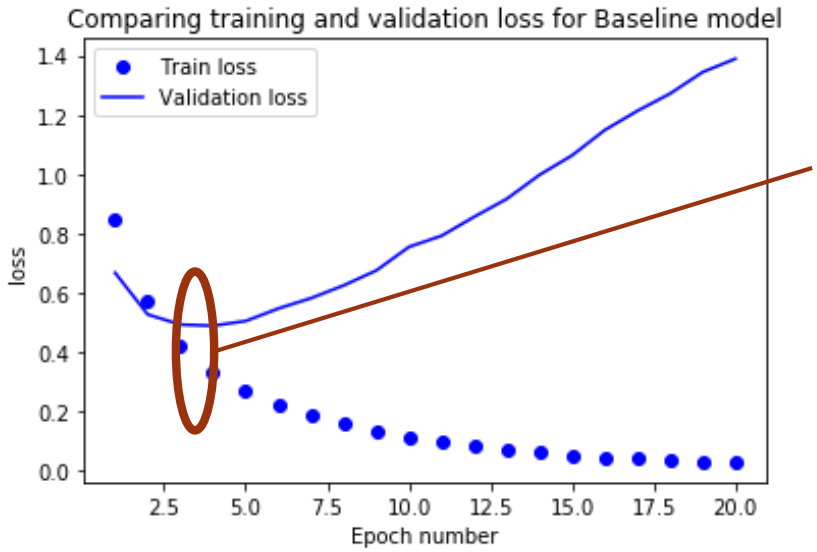
Momentum



Minimum Gradient Descent Momentum

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

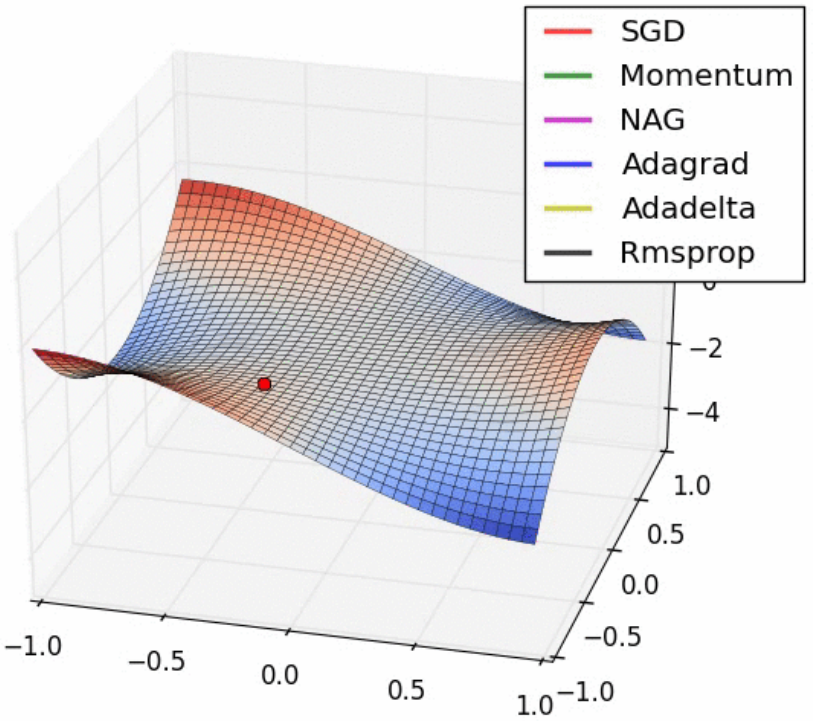
- In Keras use callback
  - es = EarlyStopping(monitor='val_accuracy', mode='max', min_delta=0.001)
    - monitor: Quantity to be monitored.
    - min_delta: Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
    - 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:

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

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

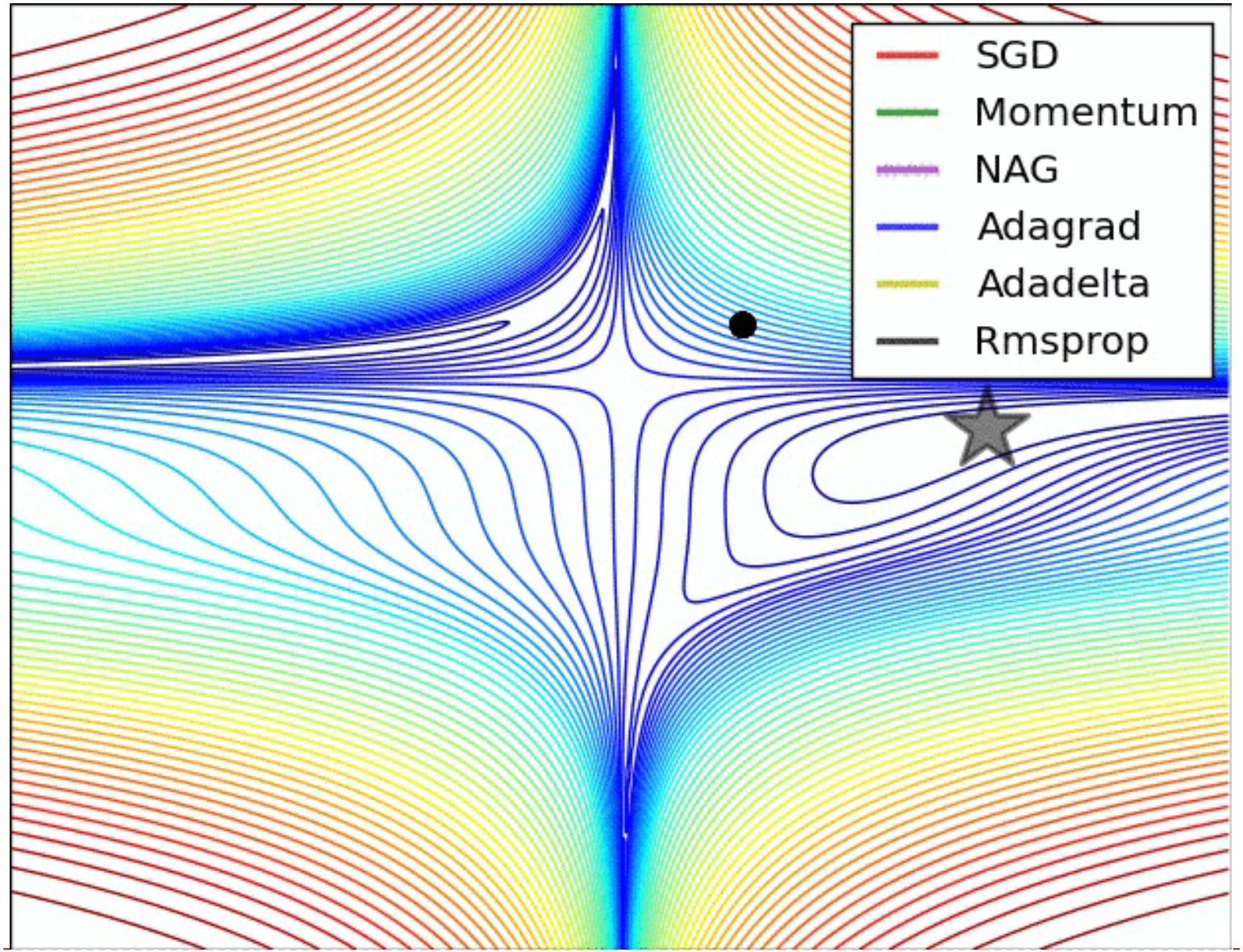
$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2; S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$$

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

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

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

Performance Comparison



How to find the best hyper-parameters

GridSearchCV use for sklearn model

- from sklearn.model_selection import GridSearchCV

- clf = LogisticRegression()
- grid_values = {'penalty': ['l1', 'l2'],
  'C':[0.001,.009,0.01,.09,1,5,10,25]}

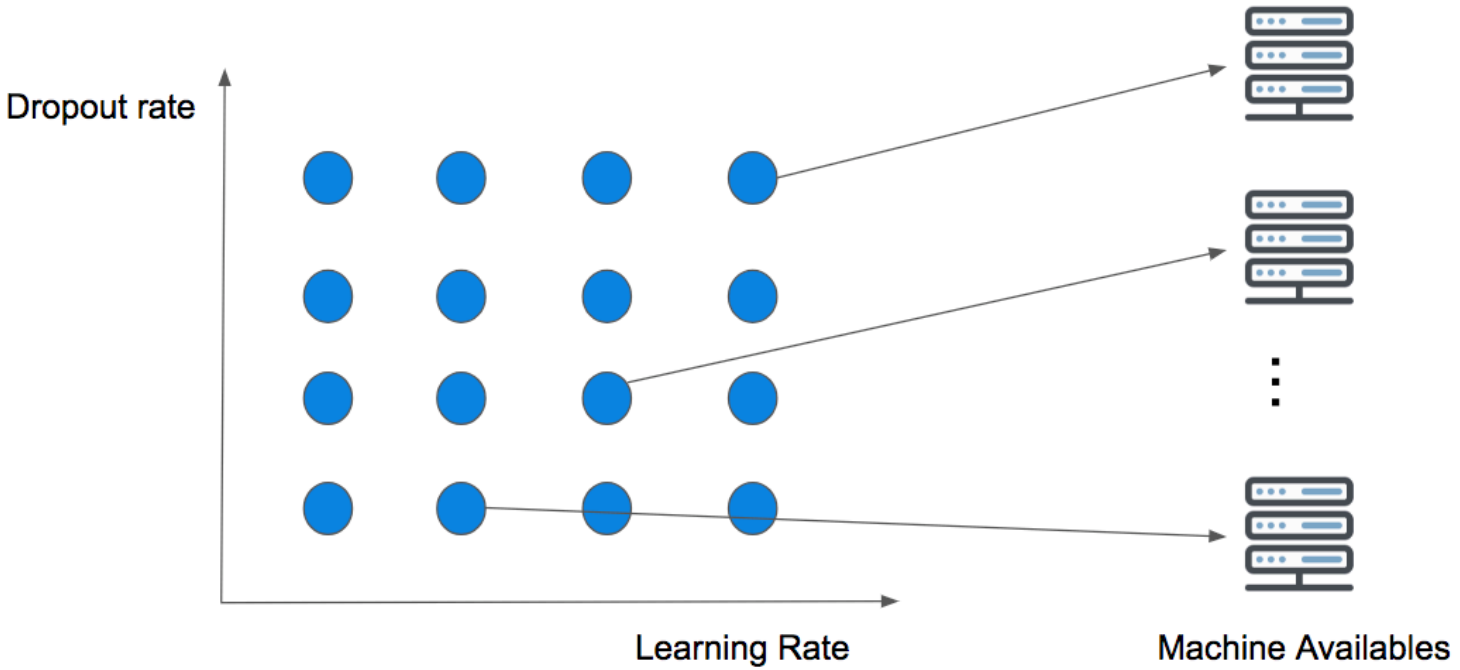
- Grid = GridSearchCV(clf,
  param_grid = grid_values,
  scoring = 'recall')
- grid_result = grid.fit(X_train, y_train)

- # Find best params
- best_params = grid_result.best_params_
- 'penalty = best_params[' 'penalty ']
- C = best_params['C']

- #Predict values based on new parameters
- y_pred = grid_clf_acc.predict(X_test)

GridSearchCV

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!

GridSearchCV use for keras model

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

- from sklearn.model_selection import GridSearchCV
- from keras.wrappers.scikit_learn import KerasClassifier

- def create_model(params):
  - # create and compile your model regarding params
  - return model

- model = KerasClassifier(build_fn=create_model)
- batch_size = [10, 20, 40, 60, 80, 100]
- epochs = [10, 50, 100]
- param_grid = dict(batch_size=batch_size, epochs=epochs)
- grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
- grid_result = grid.fit(X, Y)

- best_score = grid_result.best_score_
- best_result = grid_result.best_params_

RandomSearch

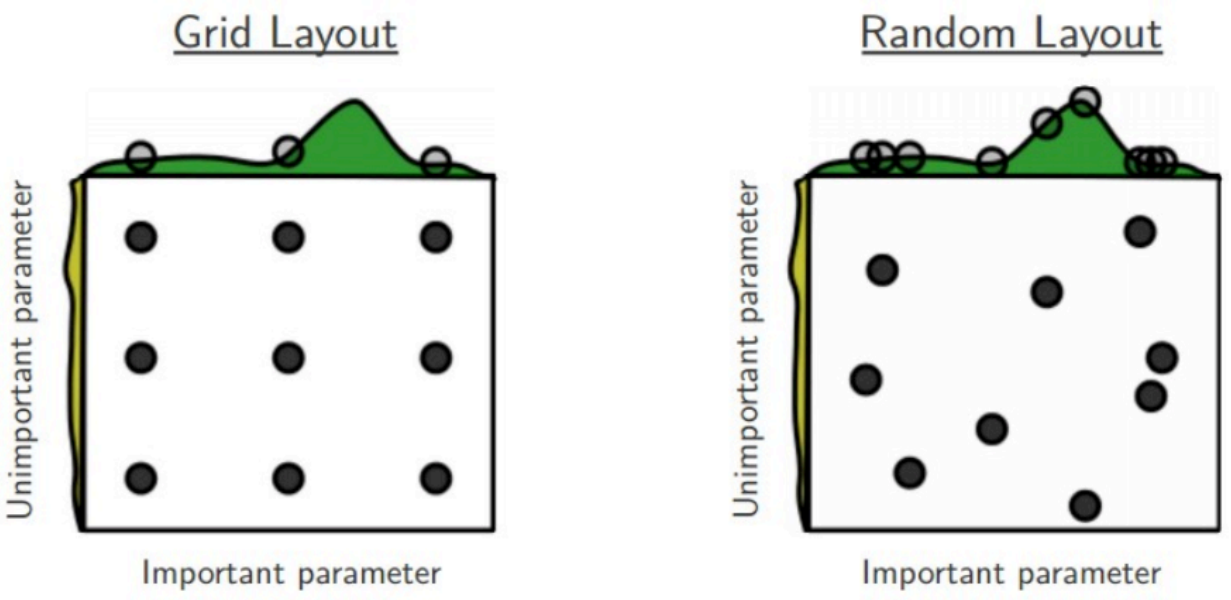


Image from http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf

RandomSearch

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