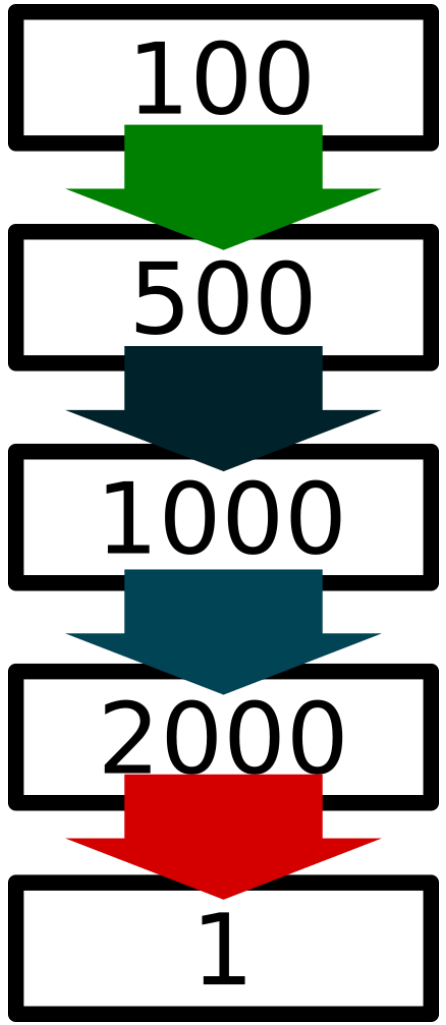
Deep Networks Learning

Álvaro Barbero Jiménez
alvaro.barbero.jimenez@gmail.com

The problem of learning in large networks

- Suppose a deep network with 100 input units, hidden layers with 500, 1000 and 2000 units, and 1 output unit
- This results in 2.5 million weights to optimize!
- Classic neural networks can use optimization methods more advanced than backprogatation: Newton, Gauss–Newton, Levenberg–Marquart, Quasi–Newton
  - But all these require computing and storing in memory the Hessian matrix (or an approximation)
- Hessian matrix is $2.5 \cdot 10\,6 \times 2.5 \cdot 10\,6 \times 8$ bytes = 52 TB !!!
  - Cannot be applied to very large networks
- Can't we do something better than simple backpropagation?



Stochastic Gradient Descent (SGD)

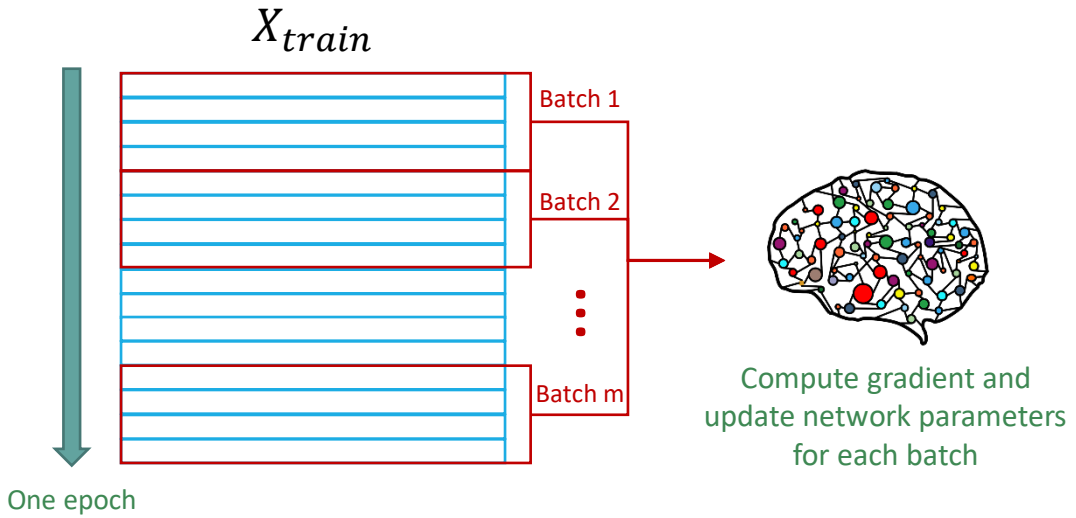
Instead of trying to compute the full gradient over all training data in every epoch, use batches of data to approximate the gradient:

- Process the dataset several times: epochs
- In each epoch, split the dataset into disjoint batches of equal size
- For each batch, run backpropagation to compute gradients $\frac{\partial F(X_j, y_j)}{\partial \theta}$ and update the network parameters θ

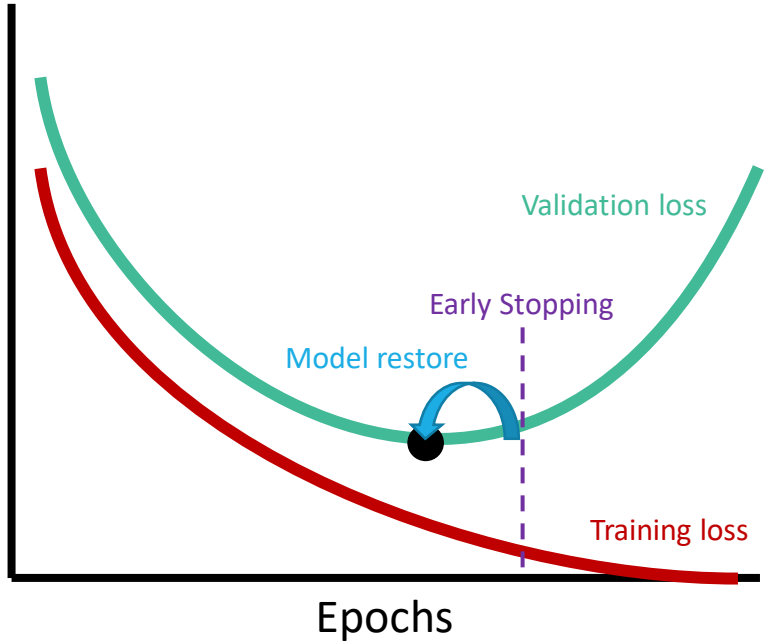
for $i = 1, \dots, n_{epochs}$ do
  $\{(X_1, y_1), \dots, (X_m, y_m)\} = split(X, y)$
  for $j = 1, \dots, m$ do
    $\theta = \theta - \delta_i \frac{\partial F(X_j, y_j)}{\partial \theta}$
  end for
end for

Improvements over standard gradient descent:

- Converges "almost surely" to a local minimum for appropriately decreasing learning rates δ_i
- Can deal with very large datasets that do not fit in memory
- Using batches has computational advantages (more on this later)



Epochs and Early Stopping



If the optimizer is correctly configured, training loss generally decreases after every epoch.

For a large enough network, training loss converges asymptotically to 0.

However, loss on a different validation dataset shows worsening after a certain point → overfitting.

Early Stopping: keep track of validation loss after every training epoch. If validation loss does not improve after p epochs, stop training. p is sometimes referred as $patience$.

It is possible to store a model checkpoint of the best performing epoch so far and restore its weights after stopping.

Early Stopping in Keras - https://keras.io/callbacks/#earlystopping

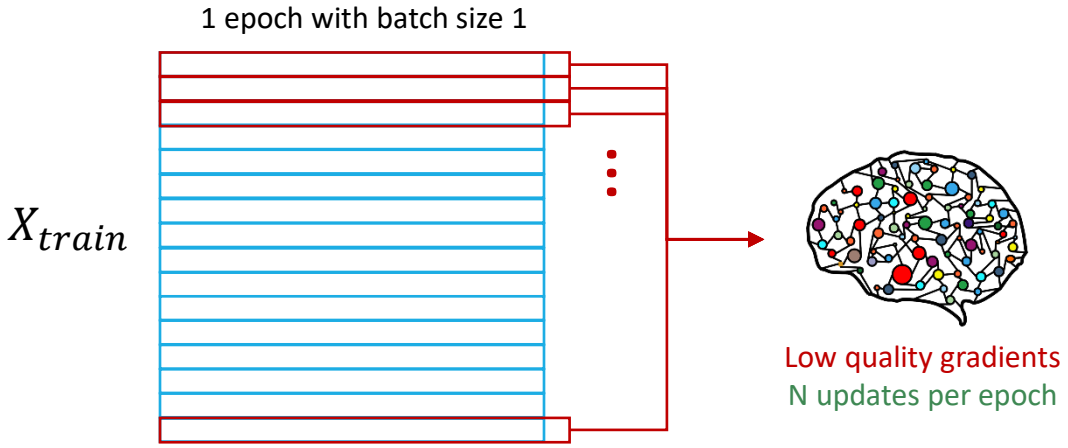
Early Stopping in Keras

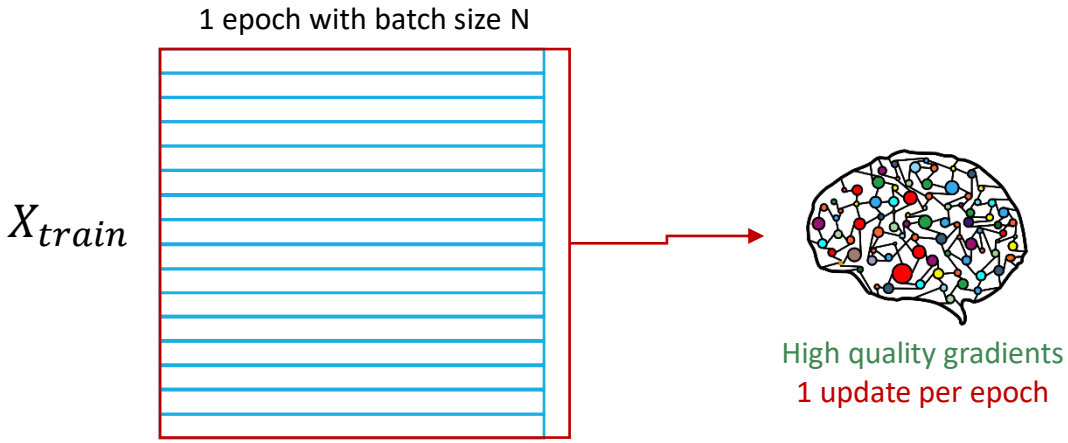
```
from tensorflow.keras.callbacks import EarlyStopping

model.fit( # model is our previously configured neural network
    Xtrain, # Training data
    Ytrain, # Labels of training data
    epochs=100, # Number of epochs to run the optimizer algorithm
    validation_data=(Xval, Yval), # Data to use as validation
    callbacks=[  # List of callback functions to run at each epoch end
        EarlyStopping(patience=10, restore_best_weights=True)
    ]
)
```

Early Stopping in Keras - https://keras.io/callbacks/#earlystopping

Batch size

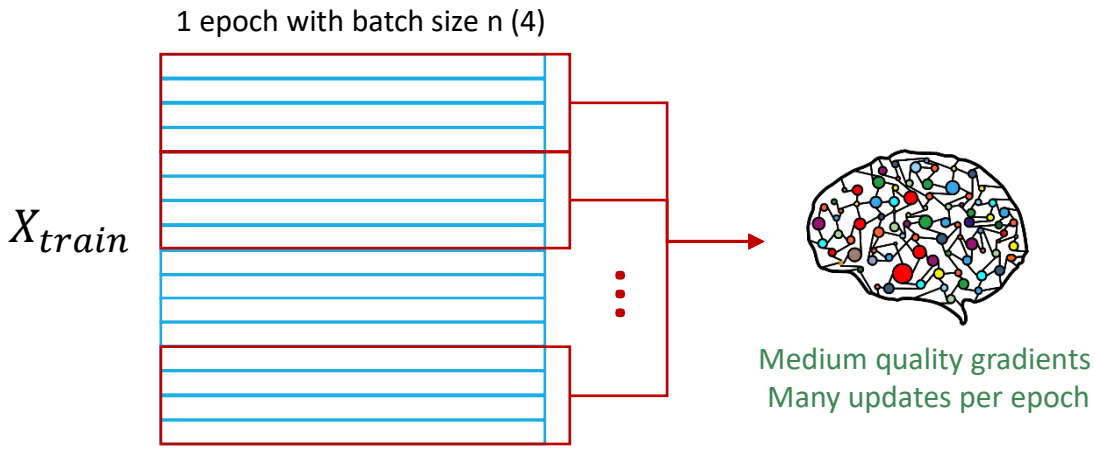




Computationally, large batch size means more efficient operations in CPU/GPU.

However, the batch size can have a significant impact in the learning process.

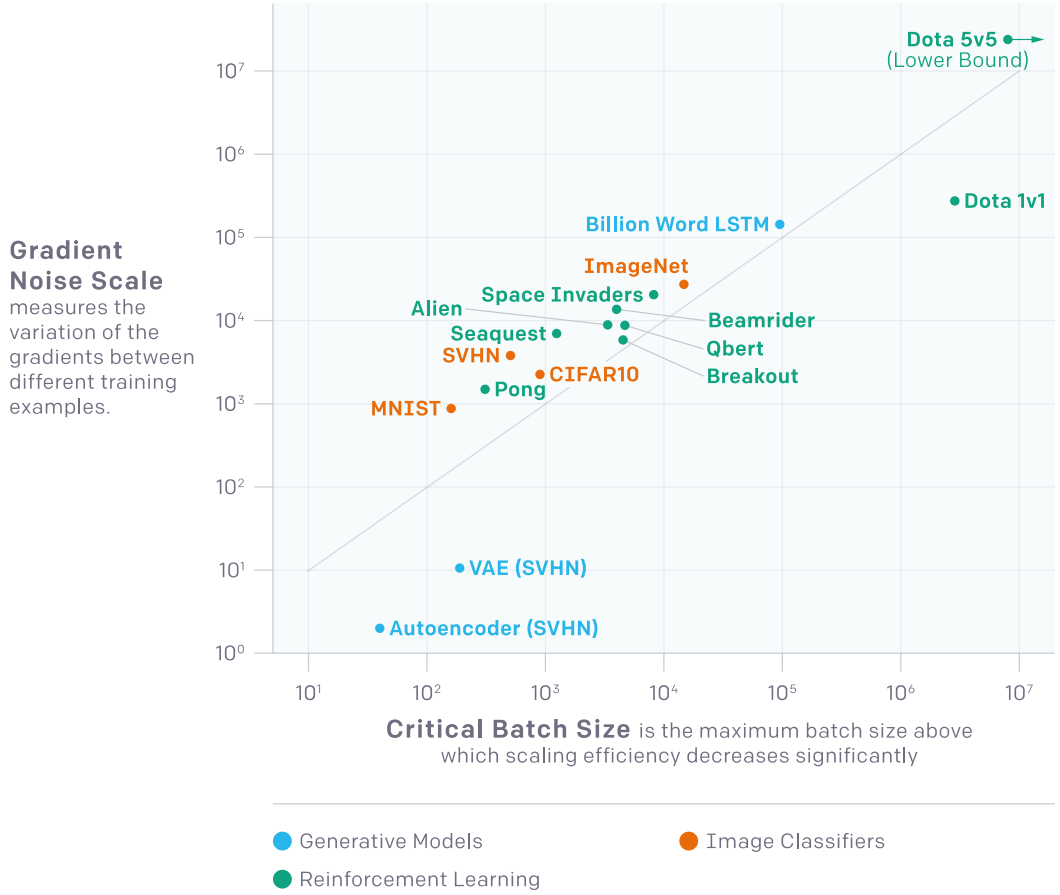
- Too small: many updates of low quality. The model may not learn correctly.
- Too large: few updates of high quality. The model learns correctly but training may take too much time.

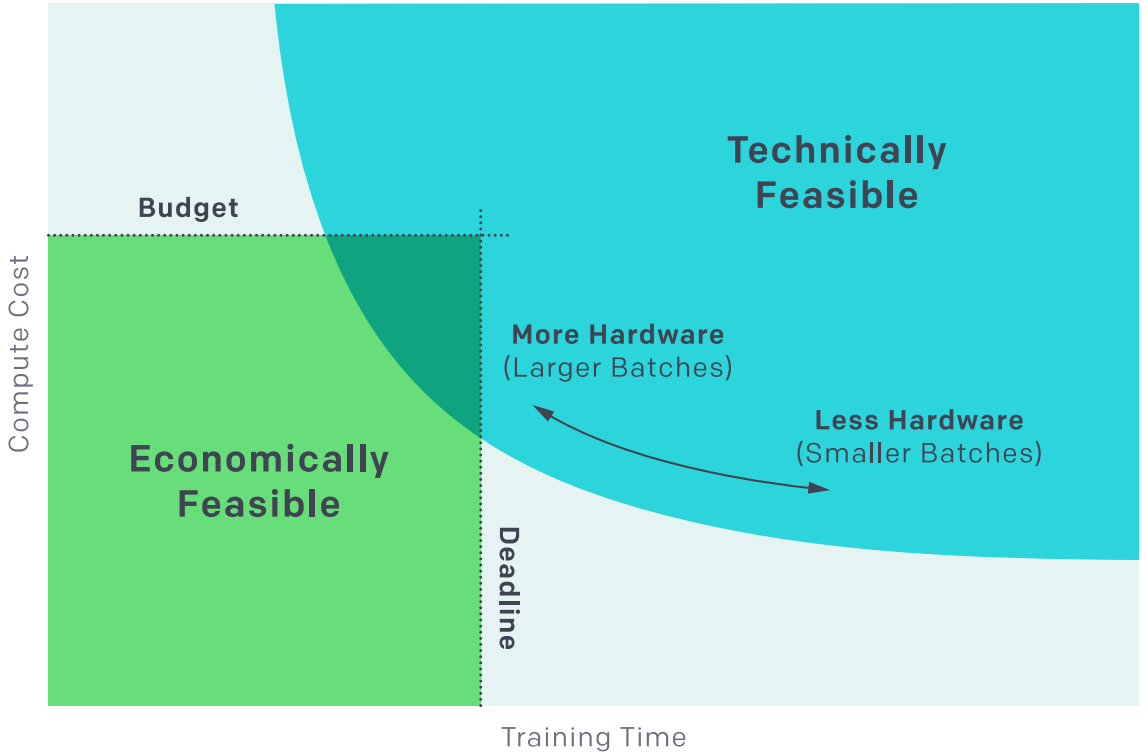


Batch size – optimal batch size

Optimal batch size depends on the complexity of the problem being solved. More complex problems require larger batch size.

Optimal size is difficult to estimate. In practice, use sizes [8, 16, 32, 64, 128]. Or look for similar problems and copy their approach!





Larger batch sizes allow efficient parallelization across many CPUs/GPUs. This means we can exchange training times and hardware costs:

More hardware (more €) → less training time

How AI Training Scales - https://openai.com/blog/science-of-ai/

SGD + momentum (A rolling stone gathers no moss)

- Shallow local minima can stop gradient descent on its tracks
- A rolling ball does not suddenly stop when finding a flat surface: it keeps some inertia
- This can be simulated by adding the gradient to a velocity term v, not directly to the position
- Velocity has an inertia (or momentum) parameter μ

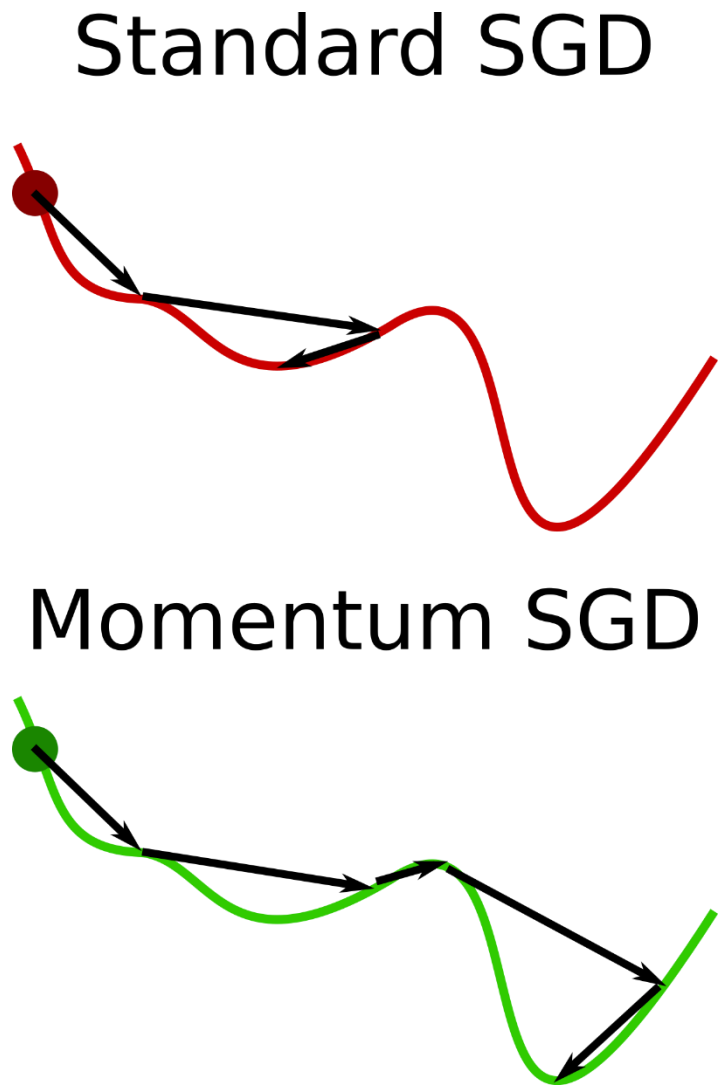
$$v^{t+1} = \mu v^t - \delta \frac{\partial F(\theta^t)}{\partial \theta^t}$$

$$\theta^{t+1} = \theta^t + v^{t+1}$$

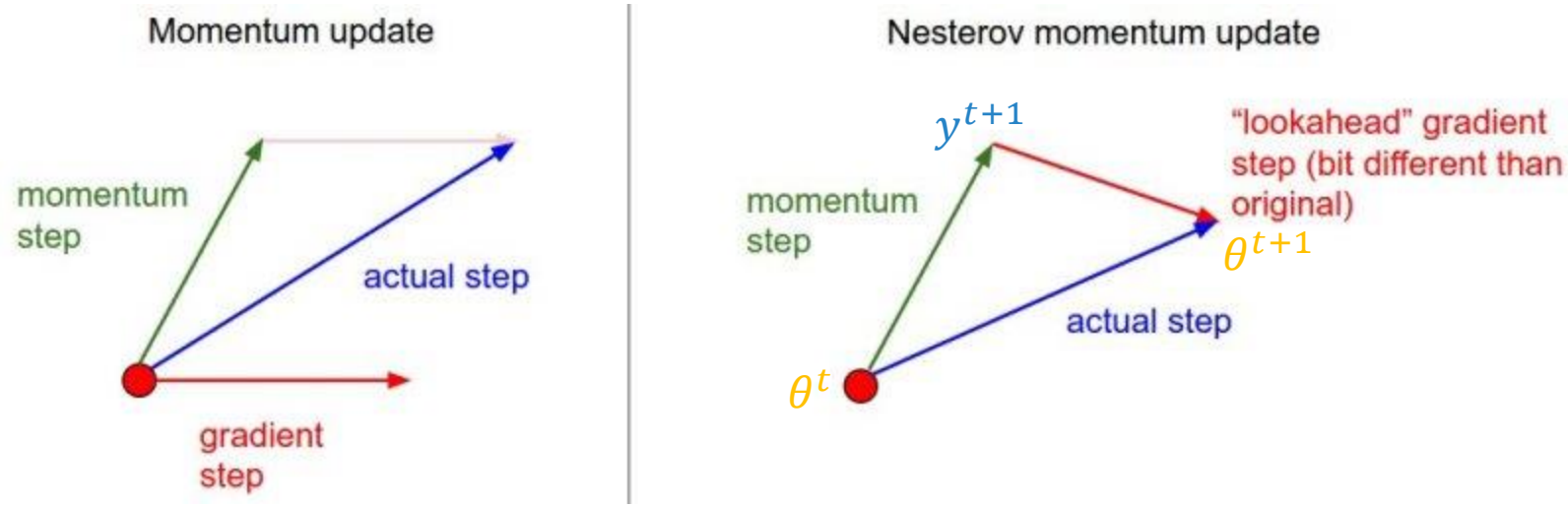
- Or equivalently

$$\theta^{t+1} = \theta^t - \delta \frac{\partial F(\theta^t)}{\partial \theta^t} + \mu\,(\theta^t - \theta^{t-1})$$

- Can be shown to be a crude approximation to Conjugate Gradient



Nesterov Accelerated Gradient (NAG)



- Improvement over standard momentum
- Perform the momentum step to move to a "lookahead" position y, and use the gradient there to perform the SGD step
- Stronger theoretical guarantees for convex functions: $O(1/T^2)$ convergence instead of $O(1/T)$
- Also works better in practice for non-convex functions

$$y^{t+1} = \theta^t + \mu v^t$$

$$v^{t+1} = \mu v^t - \frac{\partial F(y^{t+1})}{\partial y^{t+1}}$$

$$\theta^{t+1} = \theta^t + v^{t+1}$$

CS231n Convolutional Neural Networks for Visual Recognition - Learning: http://cs231n.github.io/neural-networks-3

Adagrad

- In all gradient methods it is mandatory to choose a learning rate
- Bad choices can lead to divergence or slow learning
- Adagrad is a variant of gradient descent with a learning rate that gets adapted automatically

- A cache vector c accumulates the squared values of previous gradients, for each weight in the network
- Learning rate for weights with a history of small updates are augmented
- Parameter ϵ avoids divisions by 0 ($\epsilon \in [10^{-8}, 10^{-4}]$

$$c^{t+1} = c^t + \left(\frac{\partial F(\theta^t)}{\partial \theta^t}\right)^2$$

$$\theta^{t+1} = \theta^t - \frac{\delta}{\sqrt{c^{t+1} + \epsilon}} \frac{\partial F(\theta^t)}{\partial \theta^t}$$

RMSProp

- Adagrad is sometimes too aggressive and produces monotonically decreasing updates
  - This can end up in undesired early stopping
- RMSProp solves this by updating the cache through a weighted average
- A decay rate $\gamma \in \{0.9, 0.99, 0.999\}$ is introduced to slowly forget old historical updates

$$c^{t+1} = \gamma c^t + (1-\gamma)\left(\frac{\partial F(\theta^t)}{\partial \theta^t}\right)^2$$

$$\theta^{t+1} = \theta^t - \frac{\delta}{\sqrt{c^{t+1} + \epsilon}} \frac{\partial F(\theta^t)}{\partial \theta^t}$$

Adadelta

- Adadelta was developed in parallel to RMSProp, attempting to solve the same monotonically decreasing updates problem of Adagrad
- Similar method, except a weighted average of past updates v is computed, and used in replacement of the fixed learning rate δ

$$c^{t+1} = \gamma c^t + (1-\gamma)\left(\frac{\partial F(\theta^t)}{\partial \theta^t}\right)^2$$

$$v^{t+1} = \gamma v^t + (1-\gamma)\,(\theta^t - \theta^{t-1})^2$$

$$\theta^{t+1} = \theta^t - \frac{\sqrt{v^{t+1}+\epsilon}}{\sqrt{c^{t+1}+\epsilon}} \cdot \frac{\partial F(\theta^t)}{\partial \theta^t}$$

Sebastian Ruder – An overview of gradient descent optimization algorithms: http://sebastianruder.com/optimizing-gradient-descent/

Adam (ADAptive Moment estimation)

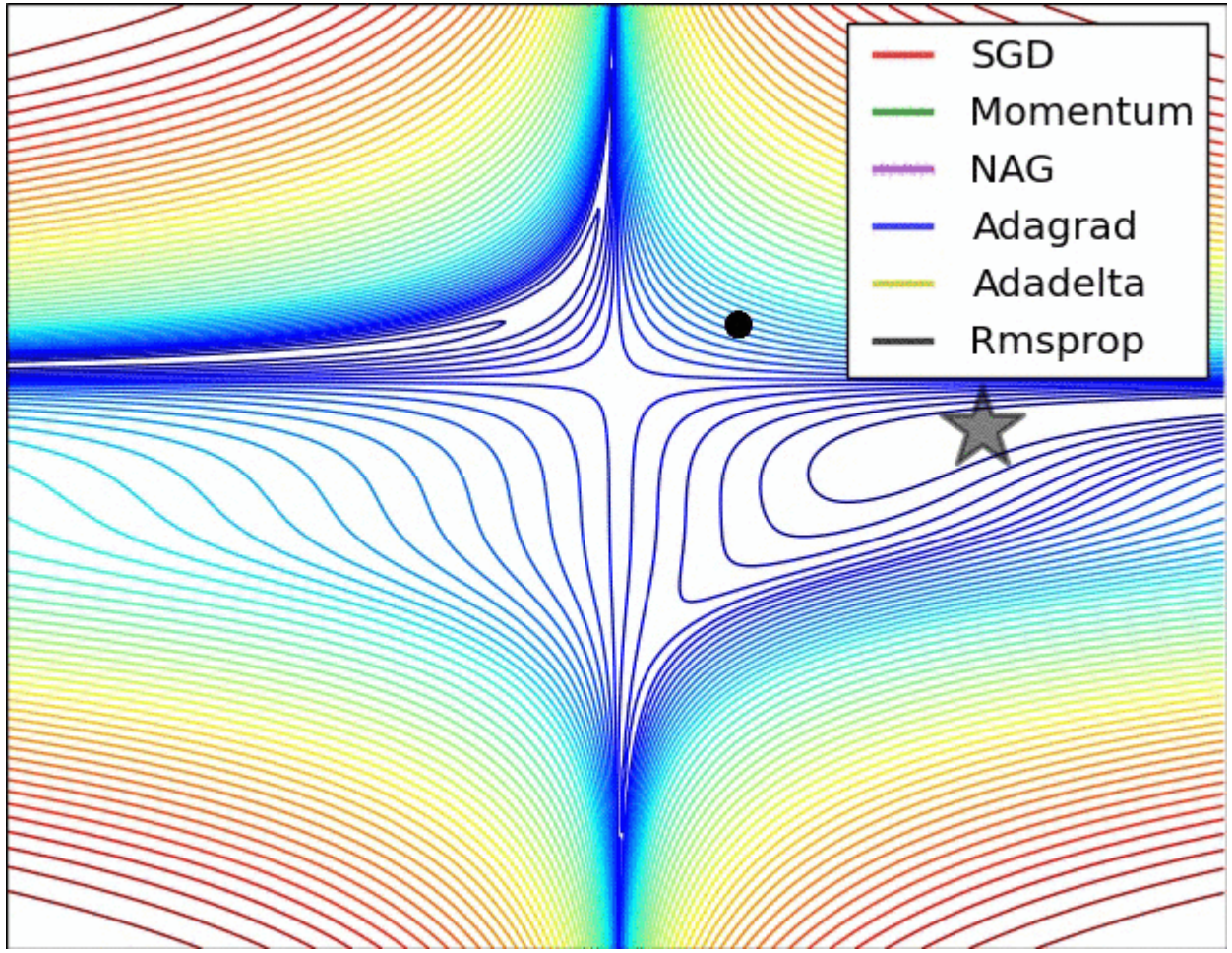
- Improvement over RMSProp where a kind of momentum term m is used
- Instead of the gradient, the smoothed gradient m is used for the update
- Vectors m and v can be seen as estimates of the first and second moments of the gradient, respectively
- Corrected versions $\hat{m}$ and $\hat{v}$ are computed to avoid initialization biases

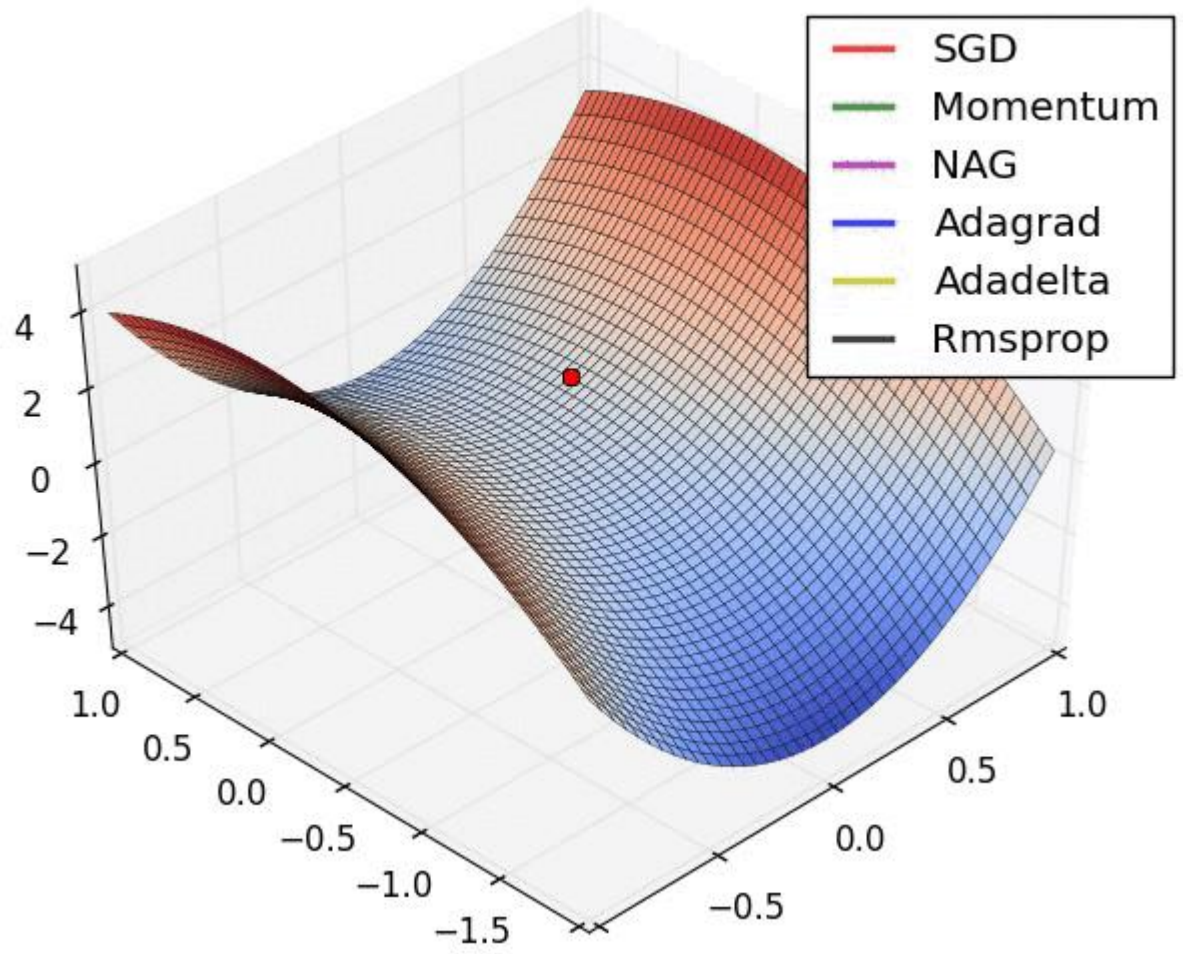
$$m^{t+1} = \beta_1 m^t + (1-\beta_1)\frac{\partial F(\theta^t)}{\partial \theta^t}, \quad \hat{m}^{t+1} = \frac{m^{t+1}}{1-\beta_1}$$

$$v^{t+1} = \beta_2 v^t + (1-\beta_2)\left(\frac{\partial F(\theta^t)}{\partial \theta^t}\right)^2, \quad \hat{v}^{t+1} = \frac{v^{t+1}}{1-\beta_2}$$

$$\theta^{t+1} = \theta^t - \frac{\delta}{\sqrt{\hat{v}^{t+1} + \epsilon}}\hat{m}^{t+1}$$

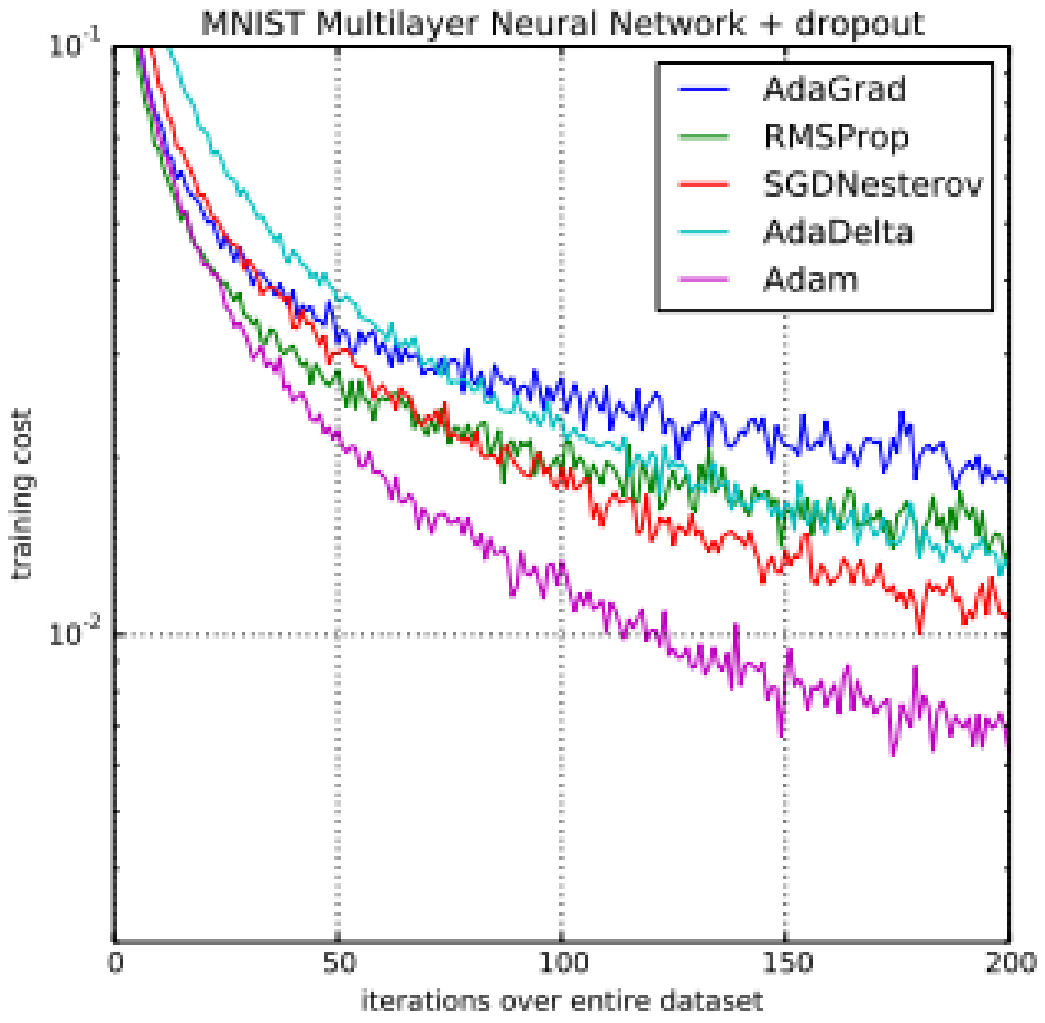
SGD variants comparison – toy problems





CS231n Convolutional Neural Networks for Visual Recognition - Learning: http://cs231n.github.io/neural-networks-3

SGD variants comparison – neural network

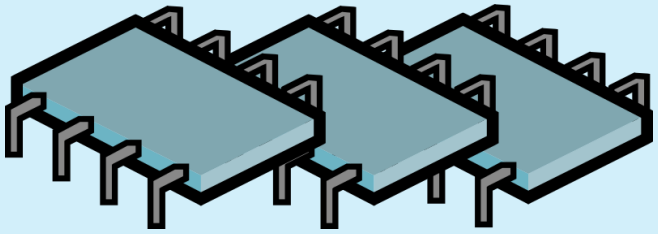


Gentle Introduction to the Adam Optimization Algorithm for Deep Learning: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Parallelization

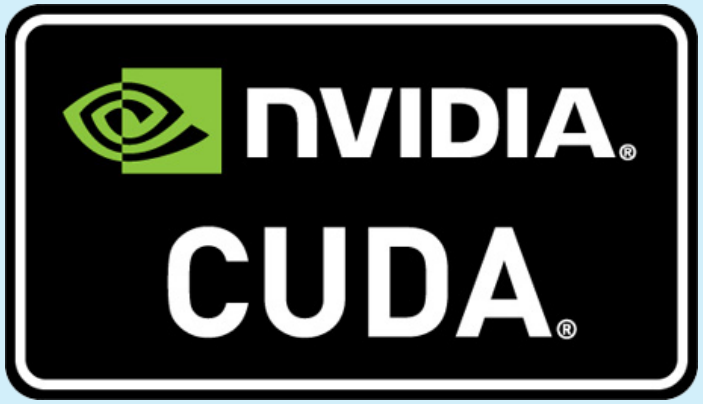
- Using improved optimization methods is helpful, but we can also take advantage of parallel computation resources

Multiple processors



Cloud computing

Coprocessors



- How can this be done?

Pattern separability in training

Loss functions used in deep learning decompose across patterns:

$$loss(\hat{y}, y) = \sum_i loss(\hat{y}_i, y_i)$$

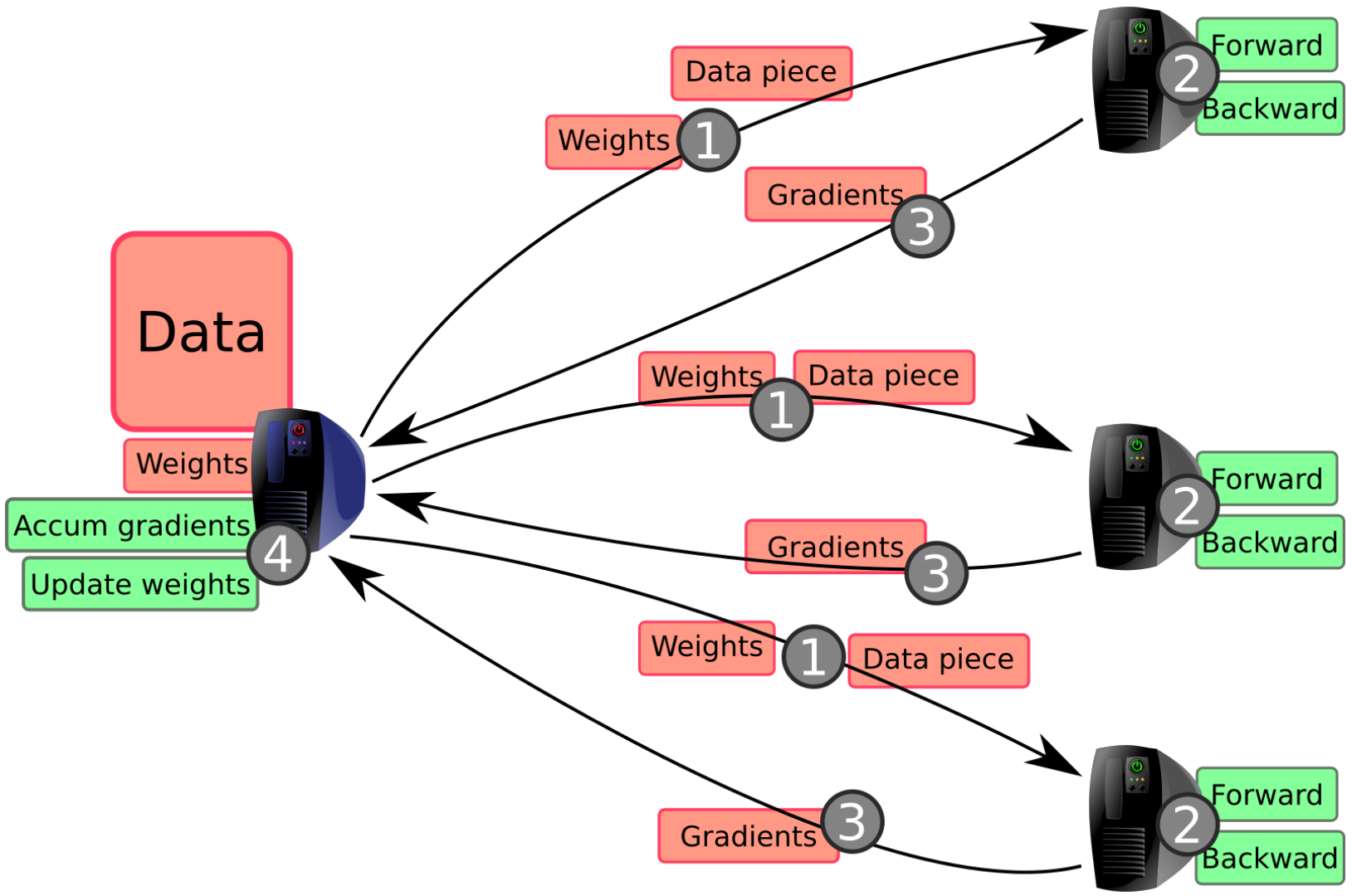
This means we can parallelize error computation (feedforward) and gradients computation (backpropagation)

$$F(X, y, \theta) = \sum_i F(X_i, y_i, \theta)$$

$$\frac{\partial F(X, y, \theta)}{\partial \theta} = \sum_i \frac{\partial F(X_i, y_i, \theta)}{\partial \theta}$$

Map Reduce: the contribution of each pattern can be computed independently, and added up later

Data-parallel training



1. Transfer network weights and a portion of the data to each node
2. Compute forward and backward in each node
3. Transfer gradients back to master
4. Accumulate gradients and update weights

➢ Master must wait for all nodes prior to updating weights

Efficient matrix operations

Most calculations in feedforward and backpropagation can be expressed in terms of matrix or vector calculations

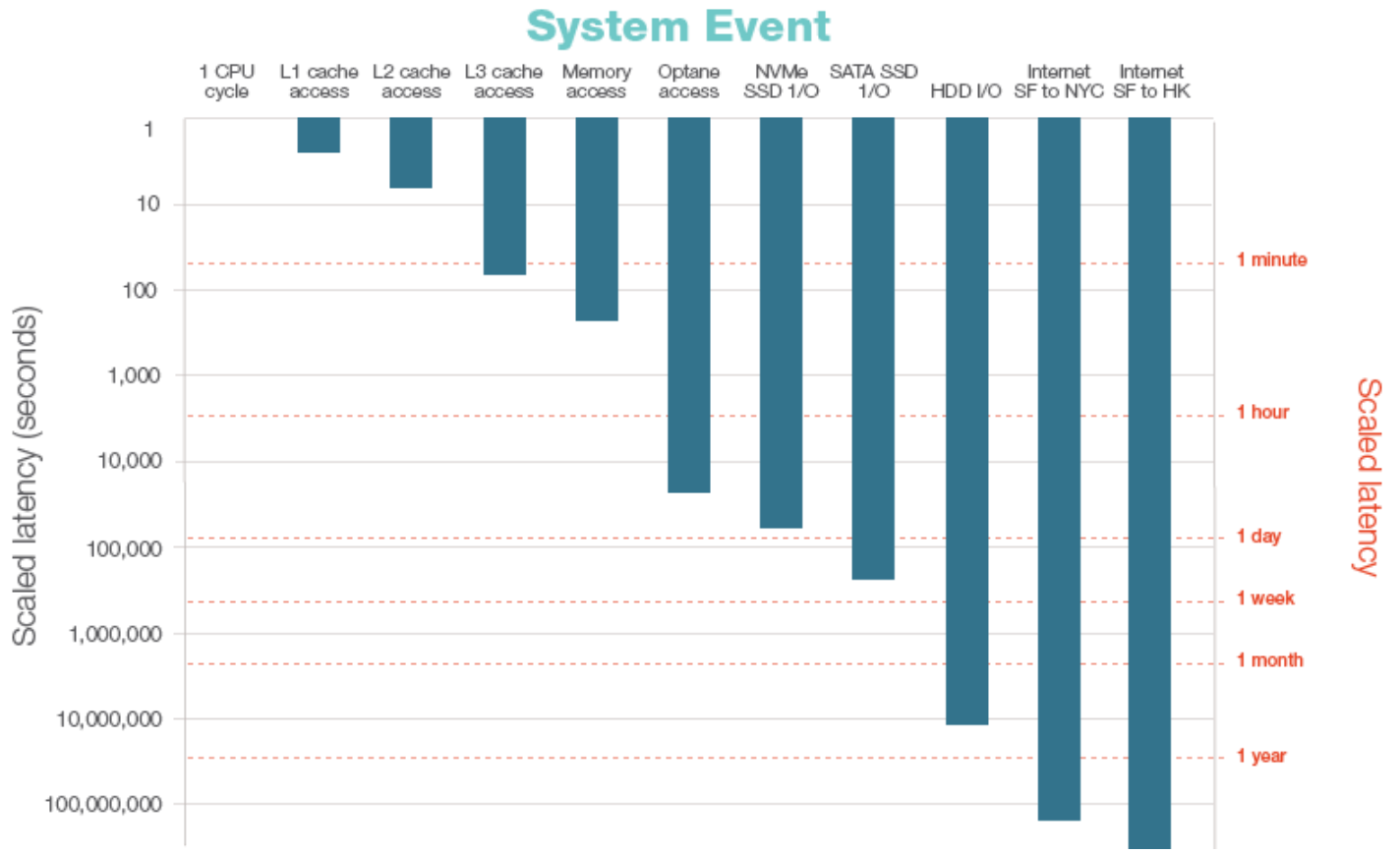
$$a = \sigma(WX + b)$$

Very efficient numerical algebra routines exist for matrix multiplications, matrix-vector products, summation, transpositions…

But… how much more efficient can this be?

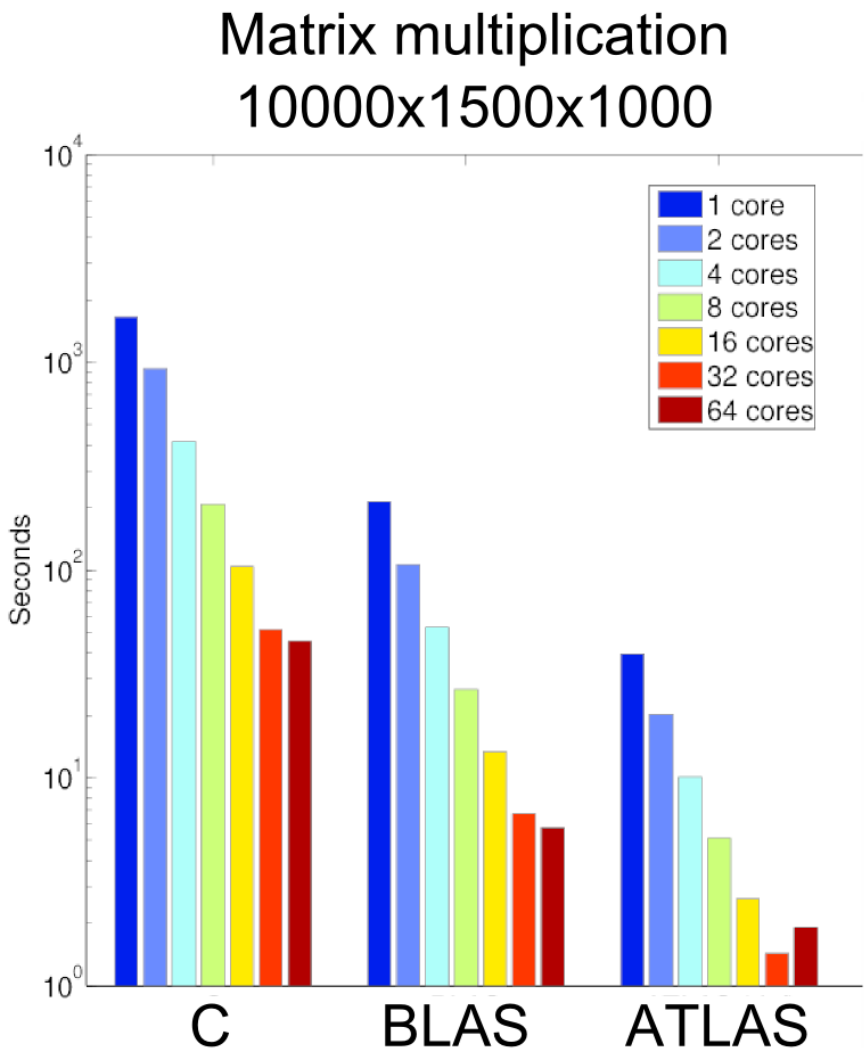
Computer Latency

System Event	Actual Latency
One CPU cycle	0.4 ns
Level 1 cache access	0.9 ns
Level 2 cache access	2.8 ns
Level 3 cache access	28 ns
Main memory access (DDR DIMM)	~100 ns
Intel® Optane™ DC persistent memory access	~350 ns
Intel® Optane™ DC SSD I/O	<10 µs
NVMe SSD I/O	~25 µs
SSD I/O	50–150 µs
Rotational disk I/O	1–10 ms
Internet call: San Francisco to New York City	65 ms[3]
Internet call: San Francisco to Hong Kong	141 ms[3]



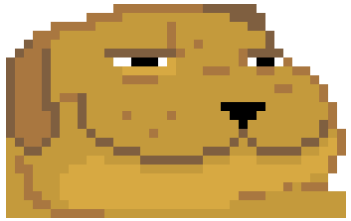
https://www.prowesscorp.com/computer-latency-at-a-human-scale/

Linear algebra libraries



- BLAS: Basic Linear Algebra Subprograms
  - Fortran implementation
  - In development since 1979!
- GotoBLAS / OpenBLAS
  - Specific implementations for different architectures
  - In assembler!
- ATLAS
  - Compiles and tests several implementations of each function, chooses the best for your computer
- Intel Math Kernel Library (MKL)
  - Already included in many Python distributions, used by numpy.

Fact: you will (probably) never do better than these libraries!



http://www.netlib.org/blas/ http://www.openblas.net/ http://math-atlas.sourceforge.net/ https://software.intel.com/en-us/mkl

GPU computation

GPUs are specialized hardware for image processing and matrix operations.

Libraries for low level neural network operations also available (cuDNN)

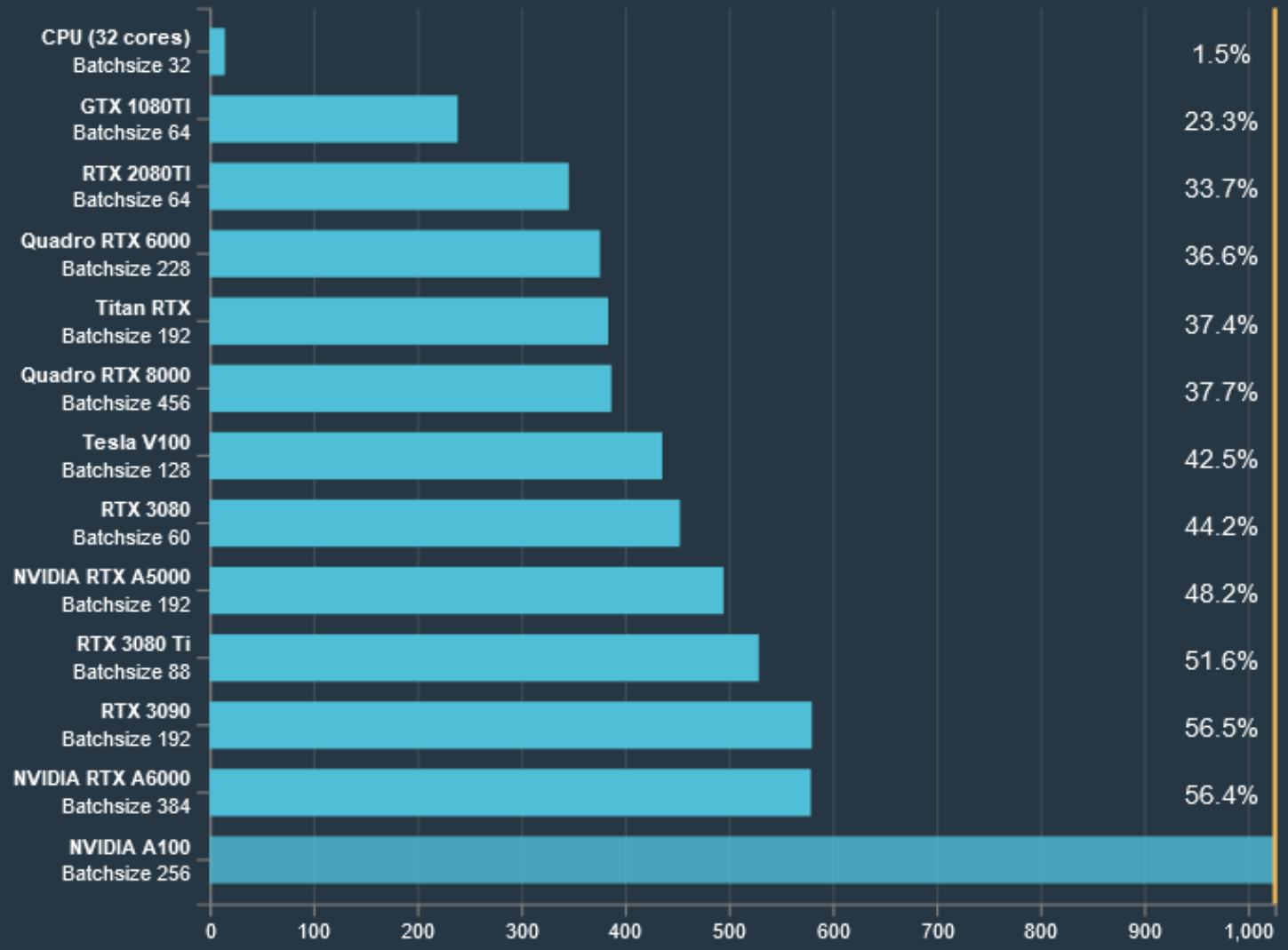
They can be integrated into a laptop or server, and also rented in cloud services.

amazon web services™

Google Cloud

Microsoft Azure

AIME A4000 Tensorflow 1.x Benchmark 2021 - Single GPU - float 32bit



ResNET50 Training Benchmark: Processed Images per Second

AIME Deep Learning GPU Benchmarks 2021

https://www.aime.info/blog/deep-learning-gpu-benchmarks-2021/

Afi Escuela