IntroML
ML. 7.2. Neural nets
Deep learnign. General concepts

DataLab CSIC

Objectives and schedule

Introduce key concepts about deep neural networks, deep learning. Architecture, SDG and variants, Automatic differentiation

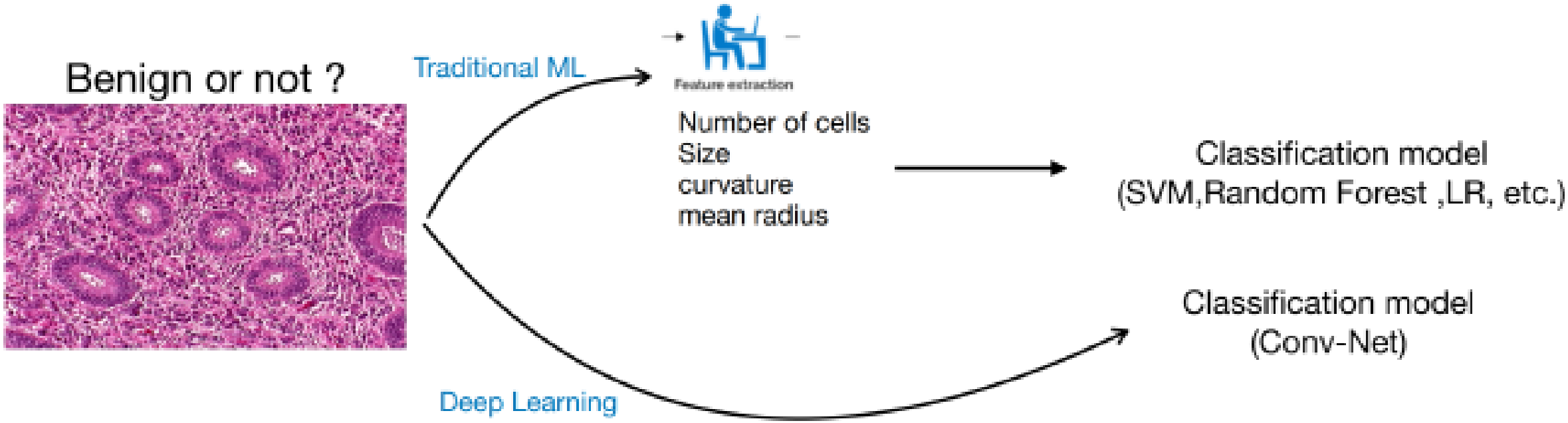
CASI 18, Goodfellow et al, Chollet and Allaire

Deep NNs. Motivation

Motivation

- (Shallow) NNs rocketed in the late 90's but plummeted
  - Vanishing gradient
  - Emergence of GB and SVMs
- Tremendous comeback in 2010's until today
  - Faster computers
  - Some math discoveries and rediscoveries
  - Massive annotated datasets
  - Success in computer vision competitions
  - Success in other traditionally complex domains (NLP, ADS,…)

New paradigm



But…

- Backprop from late 80’s
- Stochastic gradient descent from mid 50’s
- CNNs from late 80’s
- LSTM from late 90’s

Hardware

Recall training NNs entails many, many, many matrix (tensor) multiplications

- From 1990 until 2010 CPUs speed increases by factor 5000
- NNs trainable on standard laptops (still insufficient for CNNs, RNNs,…)
- GPUs https://en.wikipedia.org/wiki/Graphics_processing_unit
  - Simpler processors
  - Useful for processing large data blocks in parallel. Parallelisation
- Nvidia Titan X (app 1000€) 350 times more powerful than modern laptop

  6,6 trillion operations in floating comma per second
- TPUs… https://en.wikipedia.org/wiki/Tensor_Processing_Unit

Data

- Exponential development in storage technology
- Internet explodes. New data generated, collected and distributed
  - Wikipedia (text)
  - YouTube (video)
  - Flickr (images)
  - Twitter (opinions)
  - Smartphone (geo)
  - …
- Benchmark competitions: Imagenet, Kaggle,…

Algos

- Until 2010 no reliable training algos with NNs (beyond 1 or 2 hidden layers)
  - Vanishing gradient
- Advances
  - New activation functions beyond sigmoidals
  - SGD recalled and important variants: ADAM et al
- ‘Easy’ to train networks of 10 or more layers
- Feasible now even in the 100’s layers…

Software democratisation

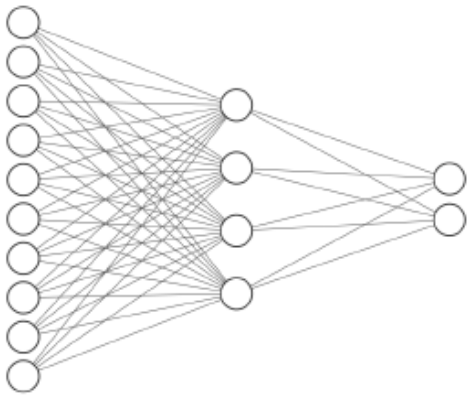
- Before: programming GPUs was sophisticated (CUDA, C++) https://en.wikipedia.org/wiki/CUDA
- From 2010 libraries like
  - Torch https://en.wikipedia.org/wiki/Torch_(machine_learning) replaced by PyTorch
  - Theano https://en.wikipedia.org/wiki/Theano_(software) now forked as Aesara
  - Caffe https://en.wikipedia.org/wiki/Caffe_(software)
  - TensorFlow https://en.wikipedia.org/wiki/TensorFlow

Facilitate automatic differentiation, matrix and tensor operations, take advantage of GPUs (and TPUs)

- On top of them Keras (interface to TensorFlow), PyTorch (Torch) facilitating their use

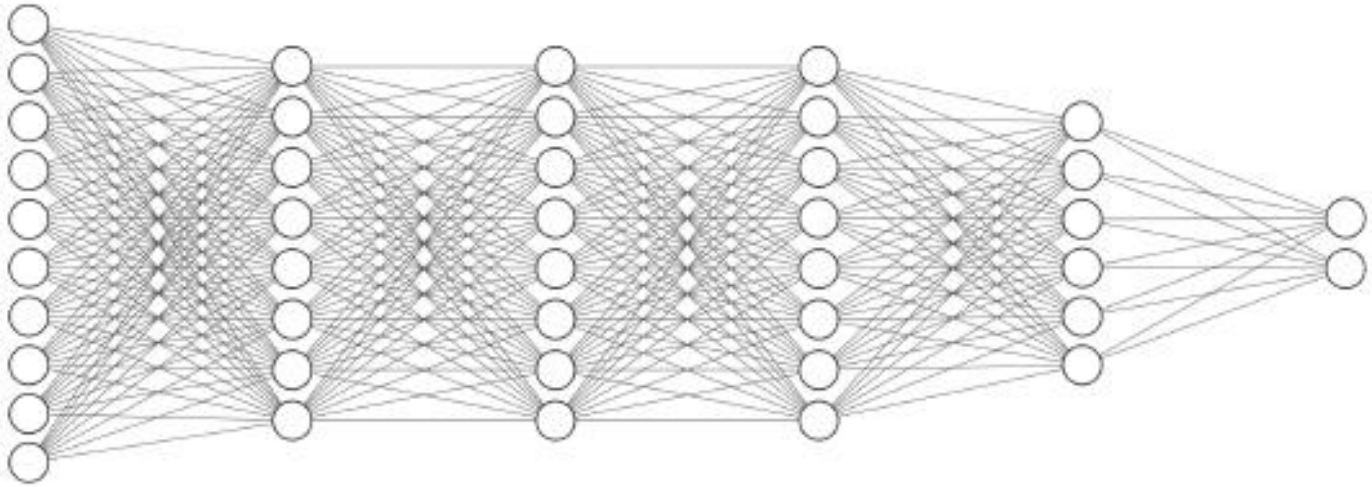
DNN architecture and training

Concept



$$y = \sum_{j=1}^{m} \beta_j \psi(x'\gamma_j) + \epsilon$$
$$\epsilon \sim N(0, \sigma^2),$$
$$\psi(\eta) = \exp(\eta)/(1+\exp(\eta))$$

(Shallow) Neural nets



$$\{f_0, f_1, ..., f_{L-1}\}$$

$$z_{l+1} = f_l(z_l, \gamma_l).$$

$$y = \sum_{j=1}^{m_L} \beta_j z_{L,j} + \epsilon$$
$$\epsilon \sim N(0, \sigma^2),$$

Deep neural nets

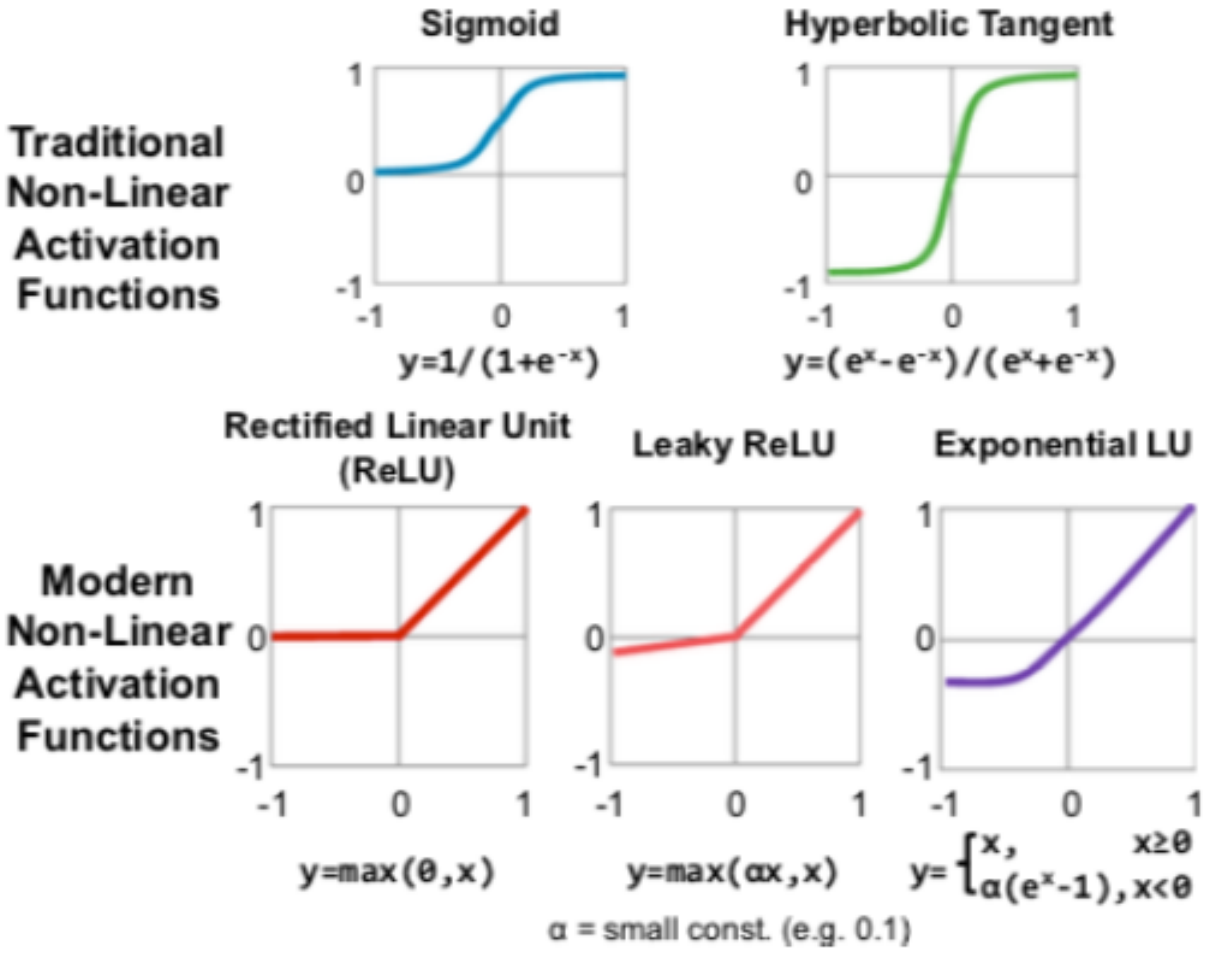
Motivation. New universal approximation theorems

Arbitrary-width. Remember Cybenko's (and others)

Arbitrary-depth.

Any (Lebesgue) integrable function can be arbitrarily approximated by RELU fully-connected neural network by as its depth goes to infinity (and its width is bounded by max(n+1,m) n dimension of origin, m dimension of image)

The evolution in activation functions



Training. Same idea!!!

Given training data, minimise -- log-likelihood + regulariser

$$\min g(\beta, \gamma) = f(\beta, \gamma) + h(\beta, \gamma),$$

Gradient descent

Backpropagation to estimate gradient

Problems

Evaluating the objective function. Depends on n

$$\sum_{i=1}^{n} f_i(\beta, \gamma)$$

Evaluating the gradient. Depends on n

$$\sum_{i=1}^{n} \nabla f_i(\beta, \gamma)$$

Each gradient sub-term $\nabla f_i(\beta, \gamma)$ over a large number of parameters and over a long backwards recurrence

Complexity was O(w) and w is getting pretty big in Deep networks

Problems

Algorithm 18.1 BACKPROPAGATION

1 Given a pair x, y, perform a "feedforward pass," computing the activations $a_\ell^{(k)}$ at each of the layers $L_2, L_3, \ldots, L_K$; i.e. compute $f(x; \mathcal{W})$ at x using the current $\mathcal{W}$, saving each of the intermediary quantities along the way.

2 For each output unit ℓ in layer L_K, compute

$$\begin{aligned} \delta_\ell^{(K)} &= \frac{\partial L[y, f(x, \mathcal{W})]}{\partial z_\ell^{(K)}} \\ &= \frac{\partial L[y, f(x; \mathcal{W})]}{\partial a_\ell^{(K)}} \dot{g}^{(K)}(z_\ell^{(K)}), \end{aligned} \tag{18.10}$$

where $\dot{g}$ denotes the derivative of $g(z)$ wrt z. For example for $L(y, f) = \frac{1}{2}\|y - f\|_2^2$, (18.10) becomes $-(y_\ell - f_\ell) \cdot \dot{g}^{(K)}(z_\ell^{(K)})$.

3 For layers $k = K-1, K-2, \ldots, 2$, and for each node ℓ in layer k, set

$$\delta_\ell^{(k)} = \left(\sum_{j=1}^{p_{k+1}} w_{j\ell}^{(k)} \delta_j^{(k+1)} \right) \dot{g}^{(k)}(z_\ell^{(k)}). \tag{18.11}$$

4 The partial derivatives are given by

$$\frac{\partial L[y, f(x; \mathcal{W})]}{\partial w_{\ell j}^{(k)}} = a_j^{(k)} \delta_\ell^{(k+1)}. \tag{18.12}$$

Stochastic Gradient Descent and variants

From gradient descent…

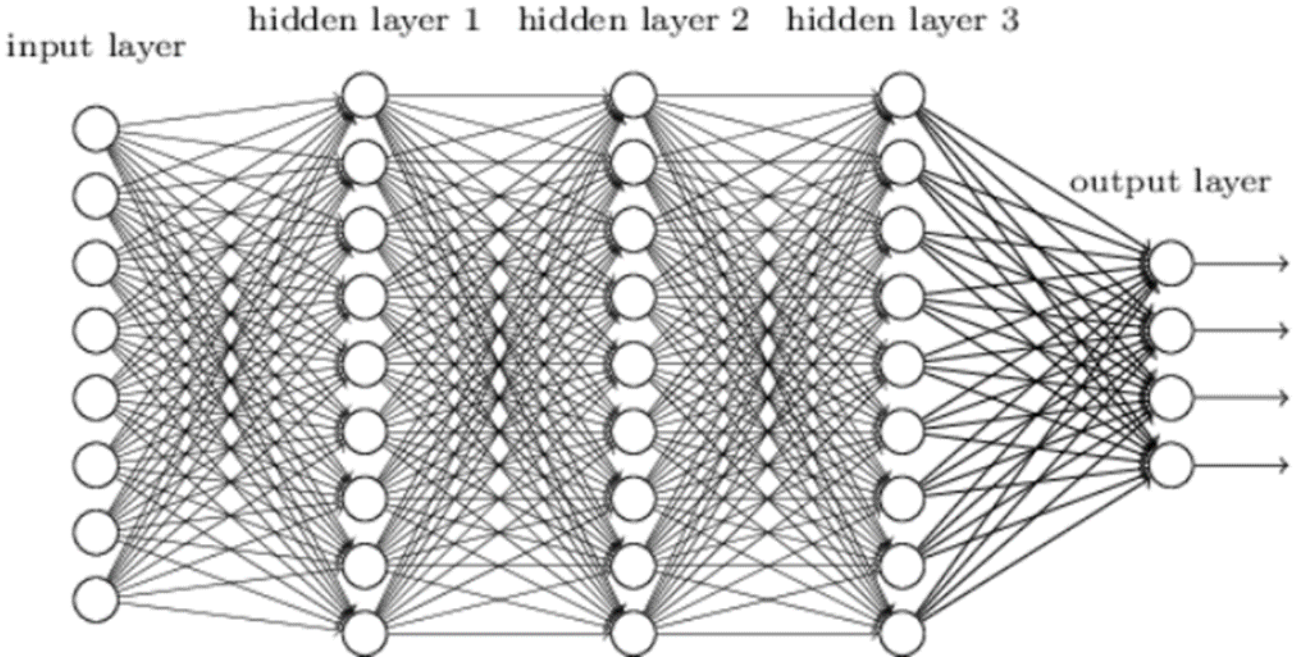
Training goes through minimising

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

$$L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$$

Requires gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$



Might not even fit in memory, very slow anyway

… to stochastic gradient descent

(Randomly) sample a minibatch of size m'.

Approximate gradient

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

Update via gradient descent

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$$

Stochastic gradient descent

Require: Learning rate ϵ_k.

Require: Initial parameter $\boldsymbol{\theta}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$

end while

Use backprop at this stage

SGD. Robbins Monro conditions (1954!!!!)

If the learning parameters are chosen so that $\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$

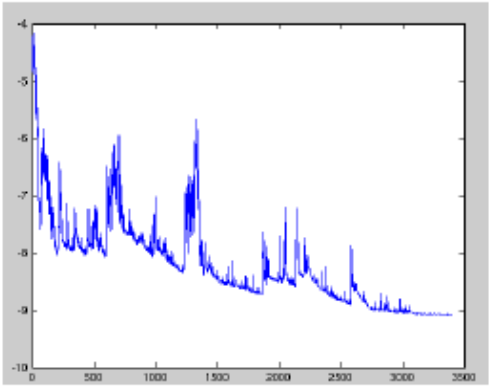
and the gradient estimator is unbiased

then SDG converges a.s. (to a local optimum)

NB: The minibatch size can even be 1 (some authors reserve SDG name to size 1 batches) , ie randomly sample just one instance and proceed

NB2: The batch now fits in memory!!!

NB3: May aid in escaping local minima!!!



Including regularizers

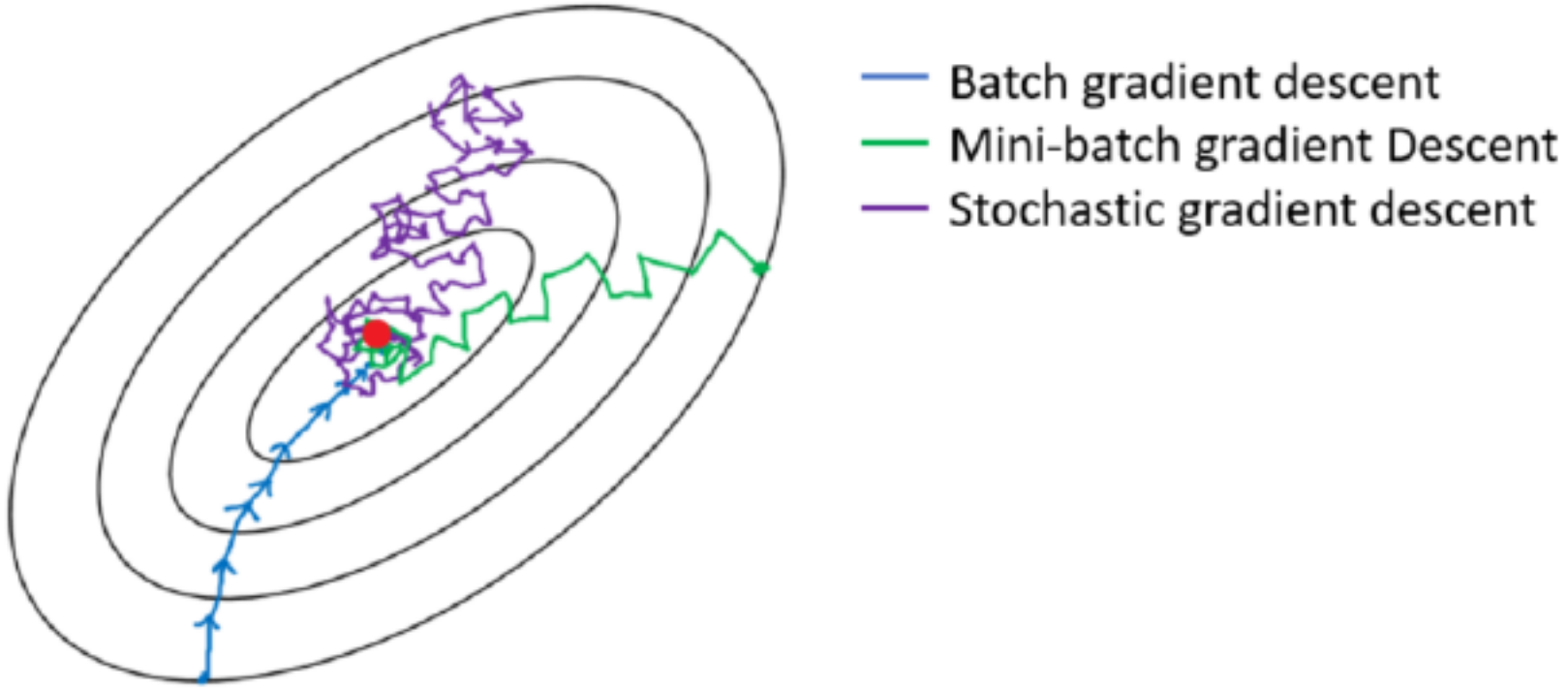
$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \quad +h(\theta)$$

$$L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \quad +\nabla_{\theta} h(\theta)$$

So same strategy works

The effects of stochasticity



The whole data set
Several random instances
1 random instance

SGD variants

SGD variants

Check Ruder's paper https://arxiv.org/abs/1609.04747

https://keras.io/api/optimizers/

Overview on a few variants (but Adam tends to be the favourite)

Trascends neural nets: very large scale optimization $\min J(\theta)$

The basic algos. GD, 1-SGD, SGD

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

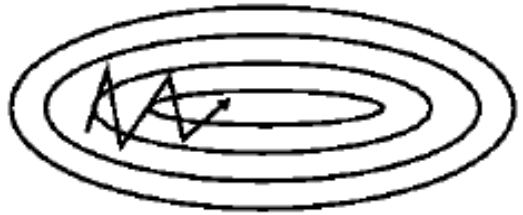
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

with $\gamma = 0.9$, typical choice

Without momentum



With momentum

NAG Nesterov accelerated gradient

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

with $\gamma = 0.9$, a typical choice

Slow down before hill slopes up

Gradient with respect to future position

Adam (Adaptive moment estimation)

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m, estimate first moment of gradient exp. decay
v, estimate second moment of gradient exp. Decay
Initialised as 0 and bets's close to 1

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Bias corrected estimates

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
  $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
  $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while

Autodiff

Differentiation modes

- By hand. Error prone, does not scale
- Numerical differentiation. Numerical errors accumulate, higher computational cost

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon} + O(\epsilon^2)$$

Differentiation modes

- Symbolic differentation (like Mathematica, Maple,…) exact handling of expressions (through tables of derivatives) leads to explosión of terms (expression swell)

n	l_n	$\frac{d}{dx} l_n$
1	x	1
2	$4x(1-x)$	$4(1-x)-4x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

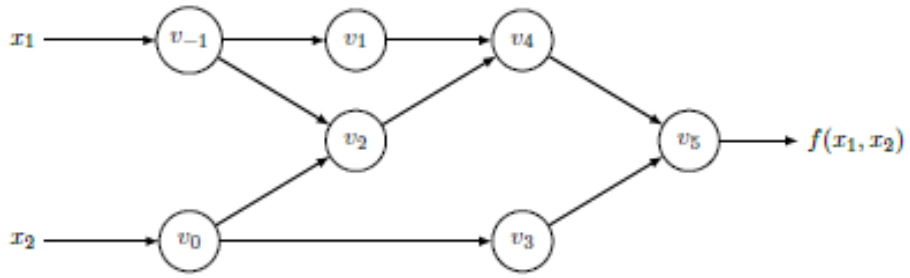
Differentiation modes

- Automatic (or algorithmic) differentiation. Symbolic differentiation to simple expressions, uses chain rule, when composing updates results.

  Generalises backprop (and trascends neural nets)

$$f(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$$

$$(x_1, x_2) = (2, 5)$$



Forward Primal Trace			Reverse Adjoint (Derivative) Trace		
$v_{-1} = x_1$		$= 2$	$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$v_0 = x_2$		$= 5$	$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$v_1 = \ln v_{-1}$		$= \ln 2$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$	$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
			$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$v_3 = \sin v_0$		$= \sin 5$	$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$v_4 = v_1 + v_2$		$= 0.693 + 10$	$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
			$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$	$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
			$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$y = v_5$		$= 11.652$	$\bar{v}_5 = \bar{y}$	$= 1$	

Autodiff

- Implemented in TensorFlow (thus Keras), Theano, ….

- If curious check review by Baydin et al https://www.jmlr.org/papers/volume18/17-468/17-468.pdf

Final comments

All conceptual elements to perform (ML+Reg) deep learning in place

Additional computational elements (Keras as an example) in Lab

We have seen fully connected NNs

Next, specialised architectures suitable in important applications (CNNs, RNNs et al)

Bayesian approaches in DL lagging….

Transfer learning

Adversarial machine learning

Explainability/Interpretability

See you next week

introml@icmat.es

Stuff at

https://datalab-icmat.github.io/courses_stats.html