A crash course in machine learning

Florent Krzakala & Antoine Baker

https://sphinxteam.github.io/mlcourse_2019/

florent.krzakala@gmail.com **antoine.baker@gmail.com**


TAKE
A
DEEP
BREATH.

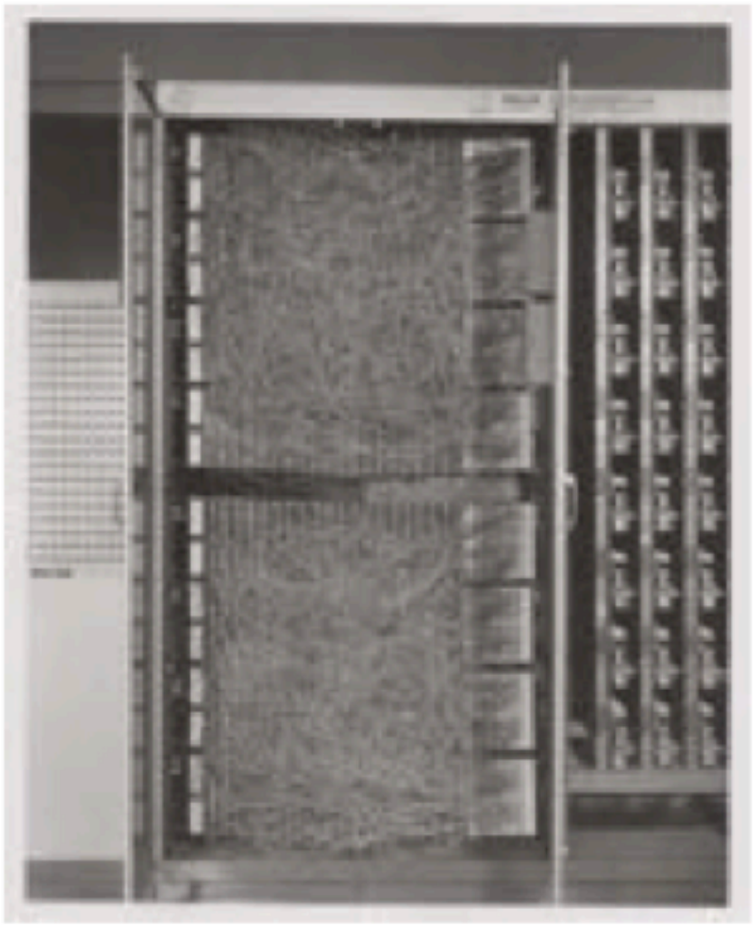
Mark I Perceptron

- first implementation of the perceptron algorithm
- the machine was connected to a camera that used 20x20 cadmium sulfide photocells to produce a 400-pixel image
- it recognized letter of the alphabet

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

update rule:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i}$$

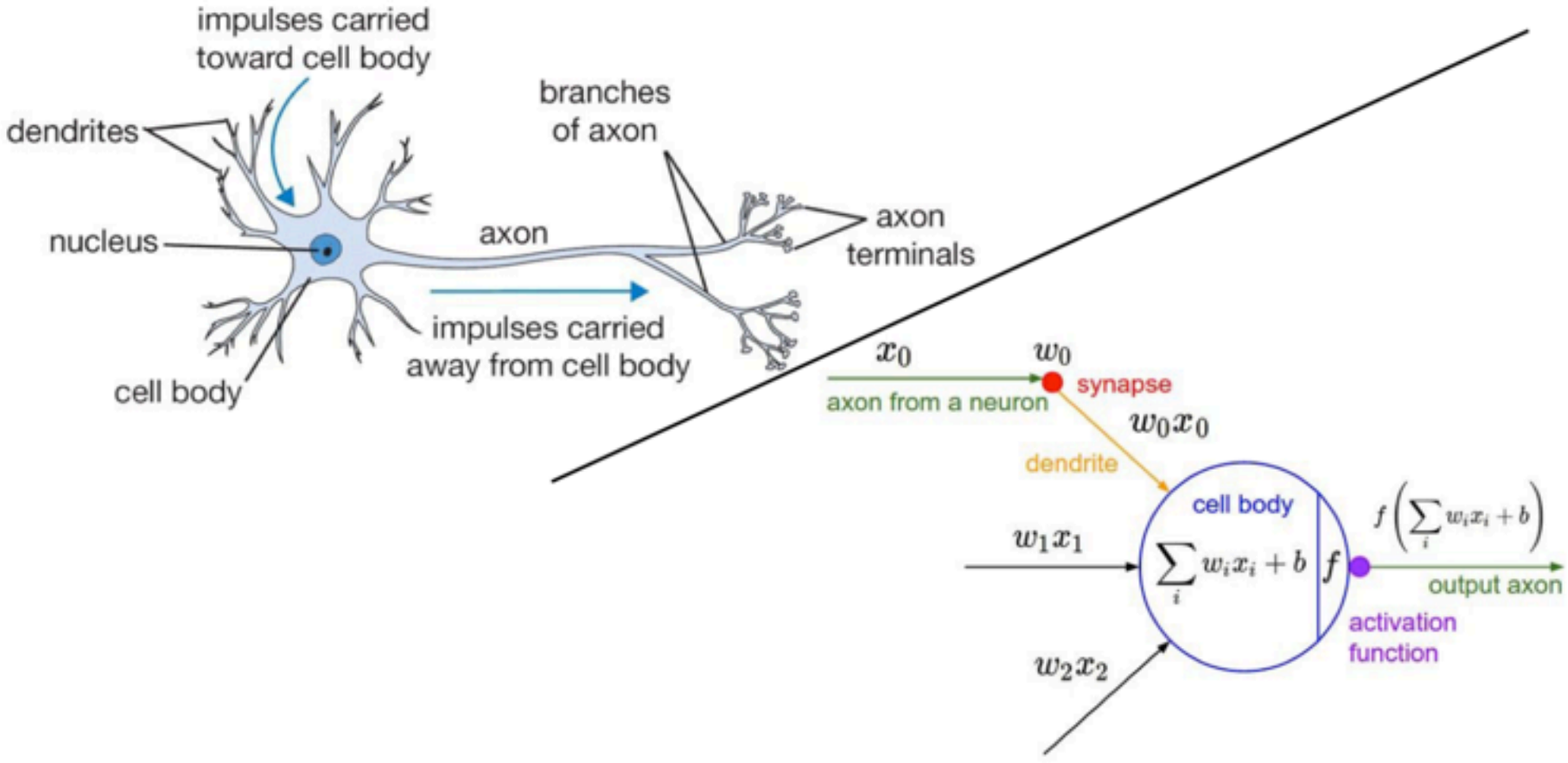


Rosenblatt, 1957

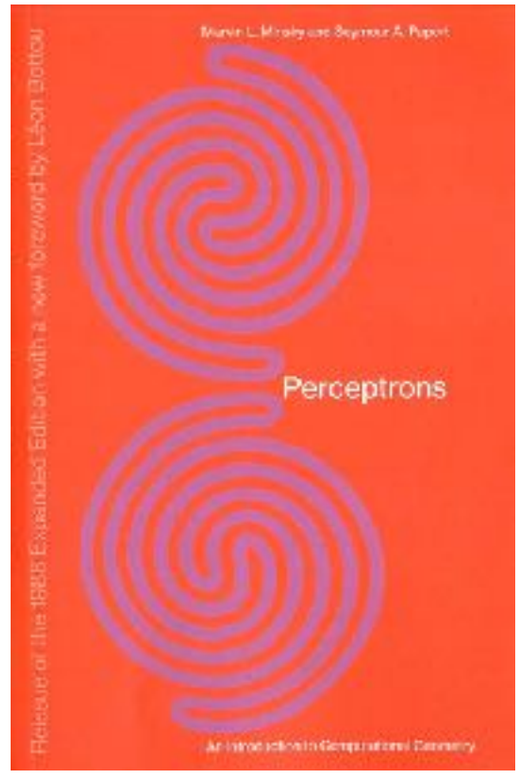
Neural Network for classification

The neuron

Inspired by neuroscience and human brain, but resemblances do not go too far



"Perceptrons have been widely publicized as ‘pattern recognition' or 'learning machines' and as such have been discussed in a large number of books, journal articles, and voluminous 'reports'. Most of this writing ... is without scientific value .."



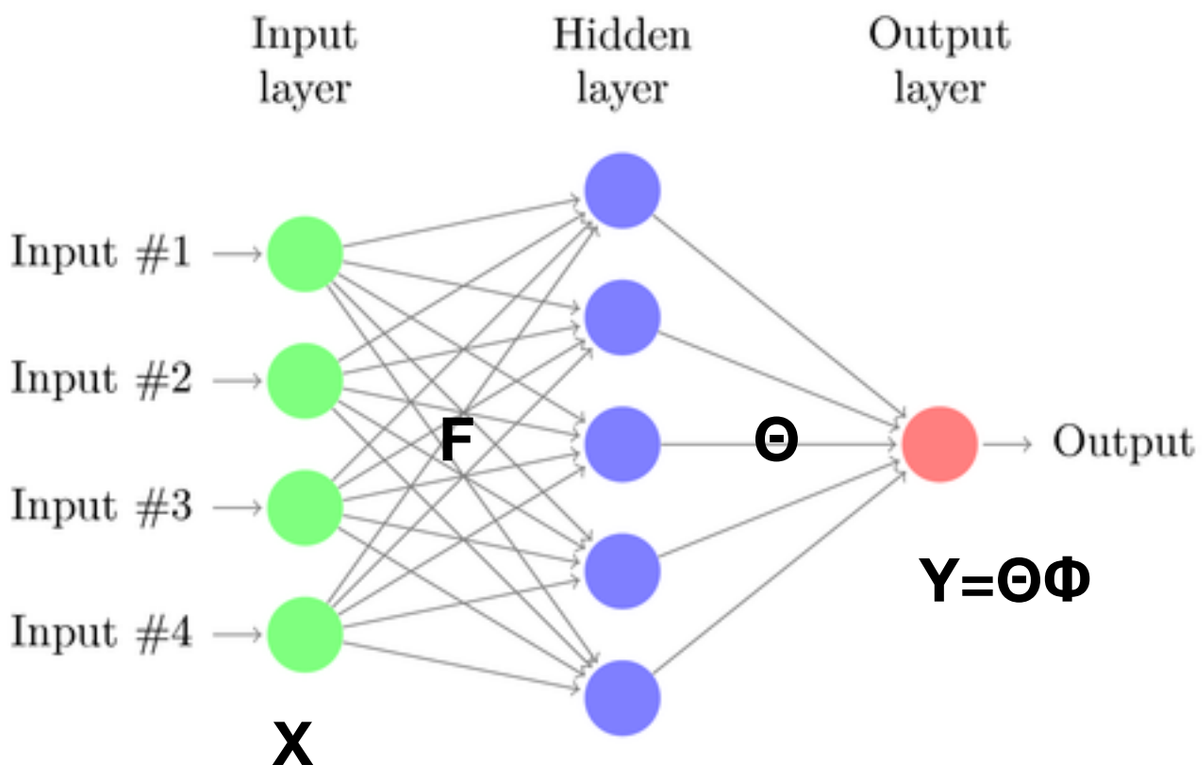
The perceptron ... has many features that attract attention: its linearity, its intriguing learning theorem.

There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it an important research problem to elucidate (or reject) our intuitive judgment that the extension is sterile.

Marvin Minsky Seymour Papert, (1969).

2-layered networks

This is our network from lesson 2
We know that it approximate kernel very well with random projections



$$\Phi = g(Fx)$$

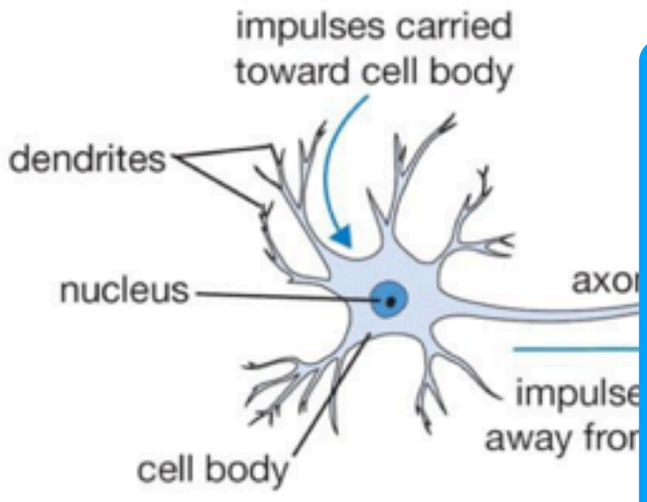
A multi-layer perception corresponds to learning the coefficients F instead of use using random ones

Non-convex, hard optimisation problem!

Neural Network for classification

The neuron

Inspired by neuroscience and human brain, but resemblances do not go too far

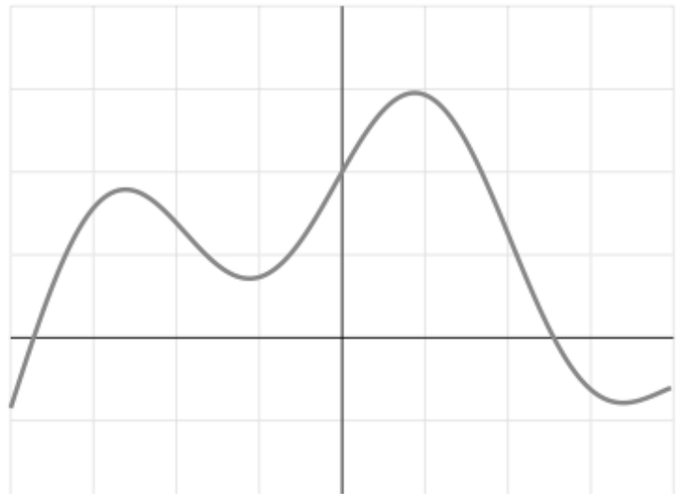


"Neural networks copy the human brain." I cringe every time I read something like this the press. It is wrong in multiple ways.
First, neural nets are loosely *inspired* by some aspects of the brain, just as airplanes are loosely inspired by birds.
Second the Inspiration doesn't come from the human brain. It comes from *any* animal brain: monkey, cat, rat, mouse, bird, fish, fruit fly, aplysia sea slug, all the way down to caenorhabditis elegans, the 1mm-long roundworm whose brain has exactly 302 neurons.

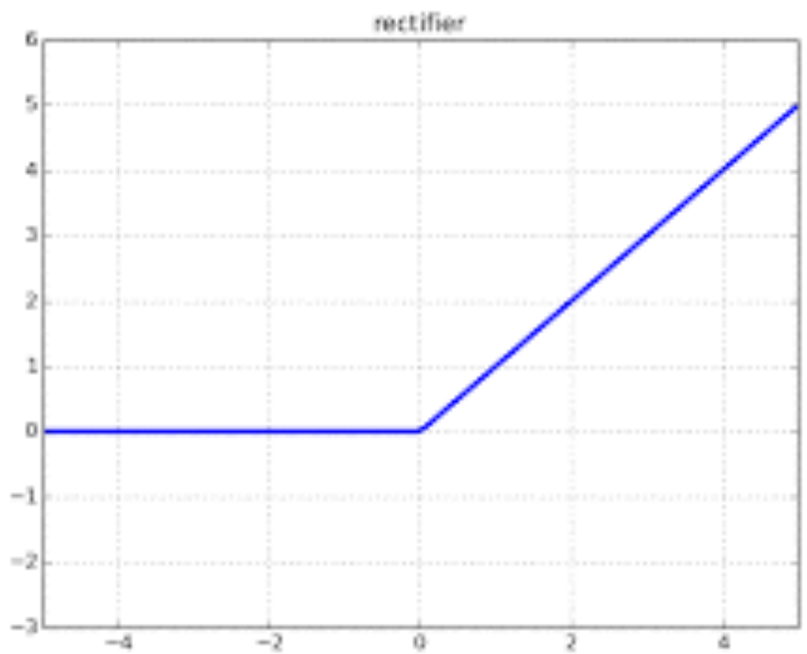
Yann LeCun, Facebook IA

Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions

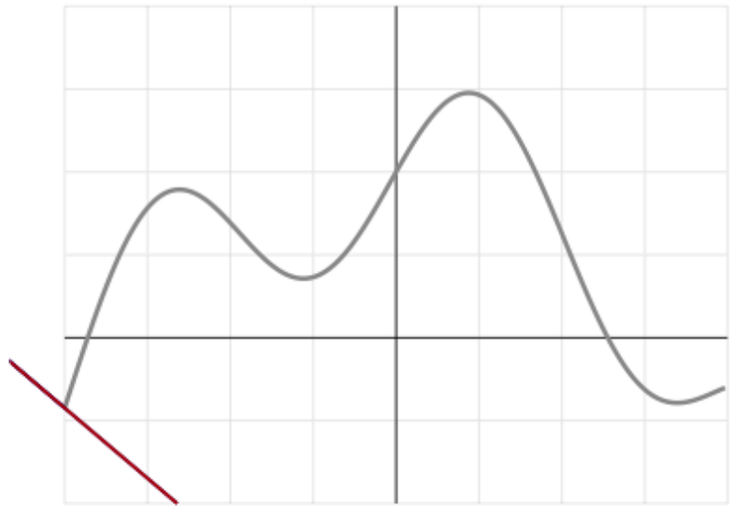


relu(x) = x if x>0 & 0 otherwise



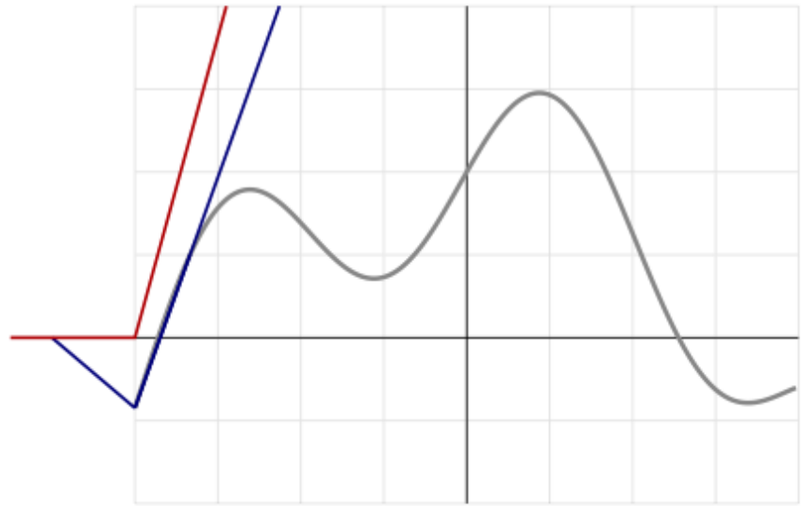
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



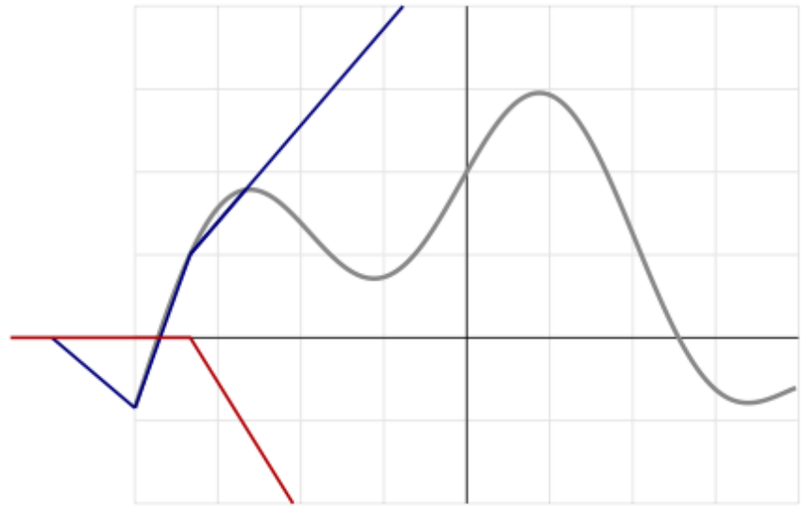
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



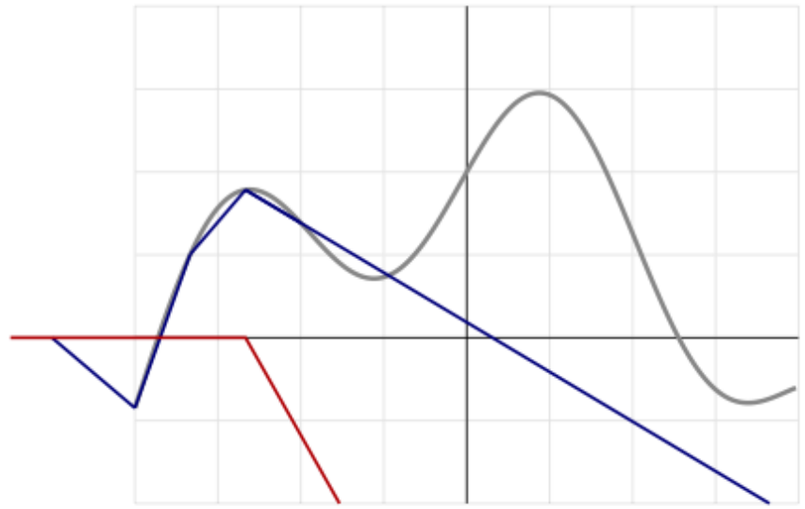
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



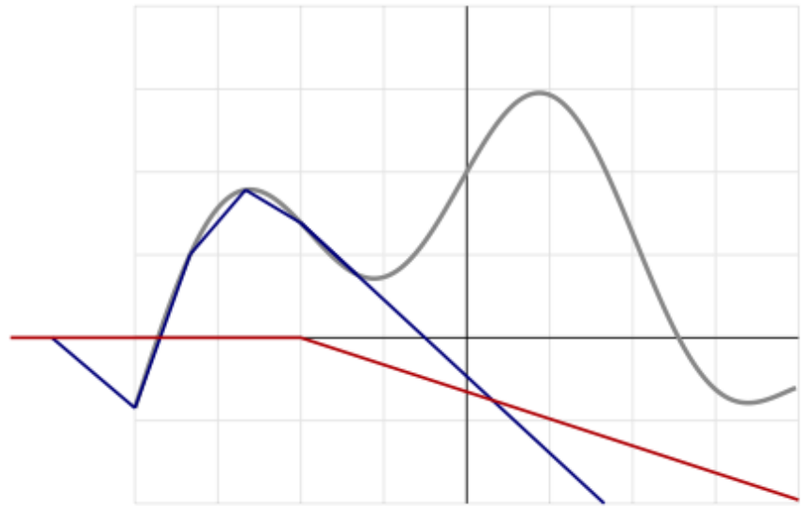
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



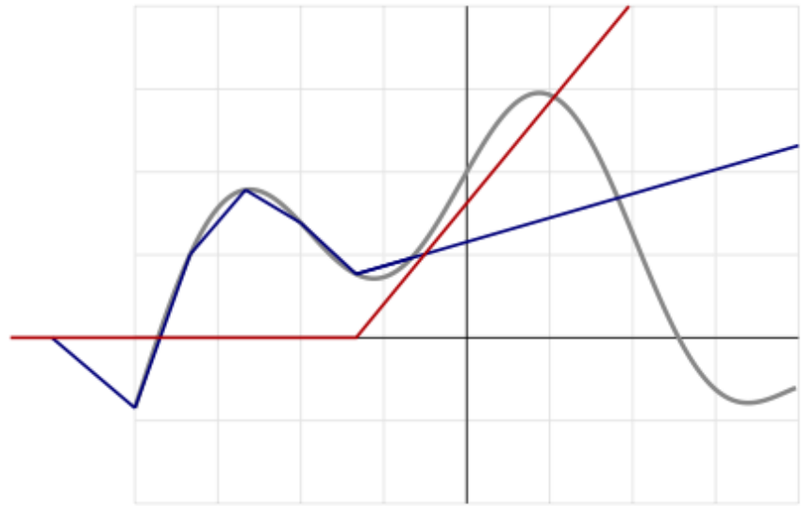
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



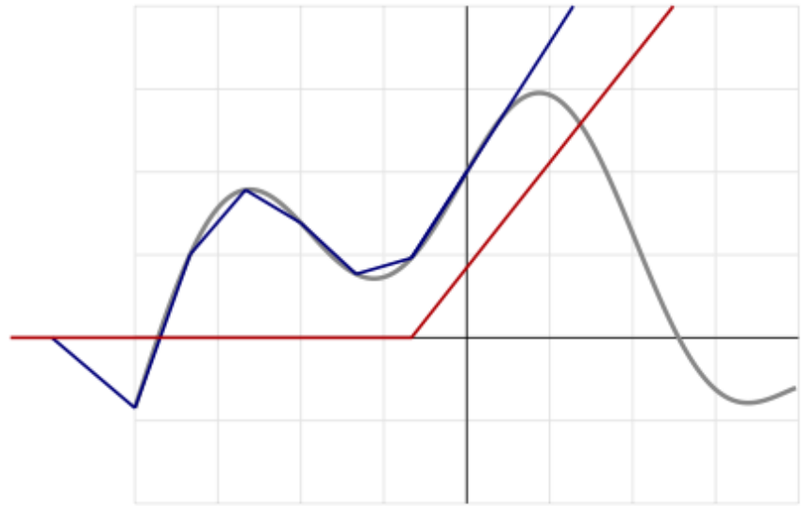
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



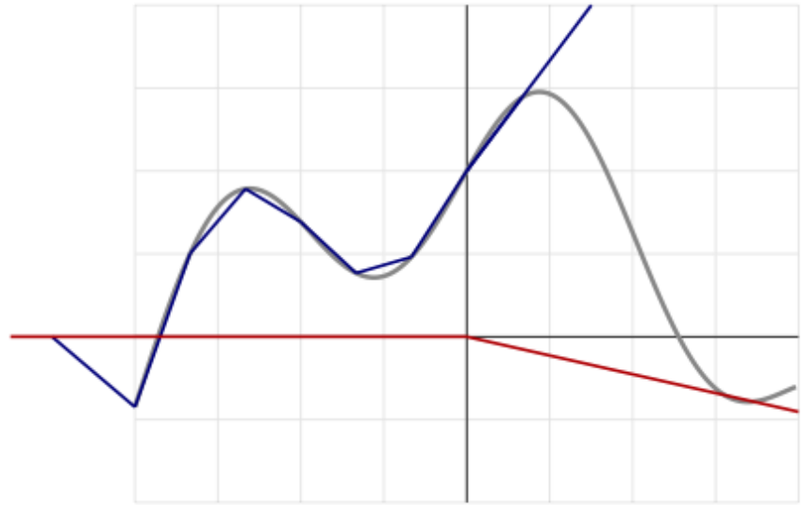
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



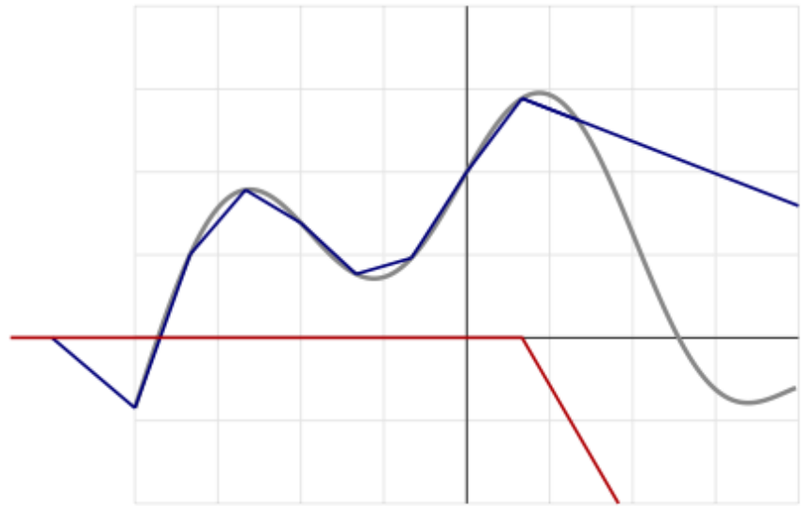
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



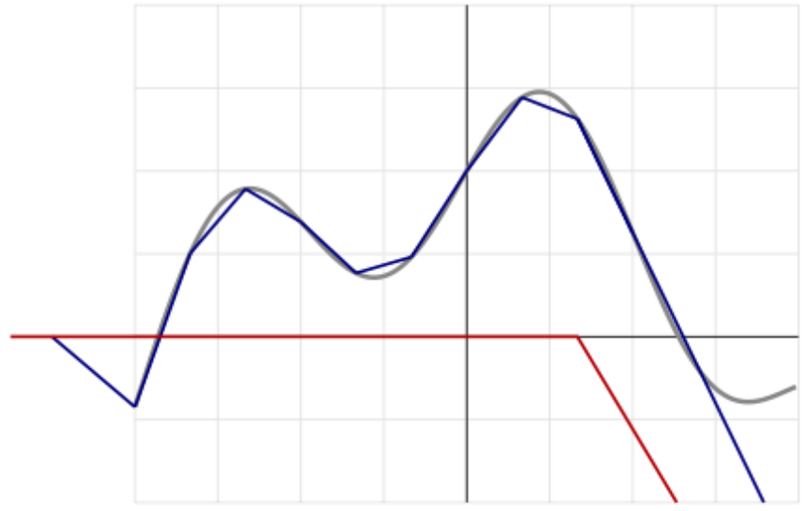
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



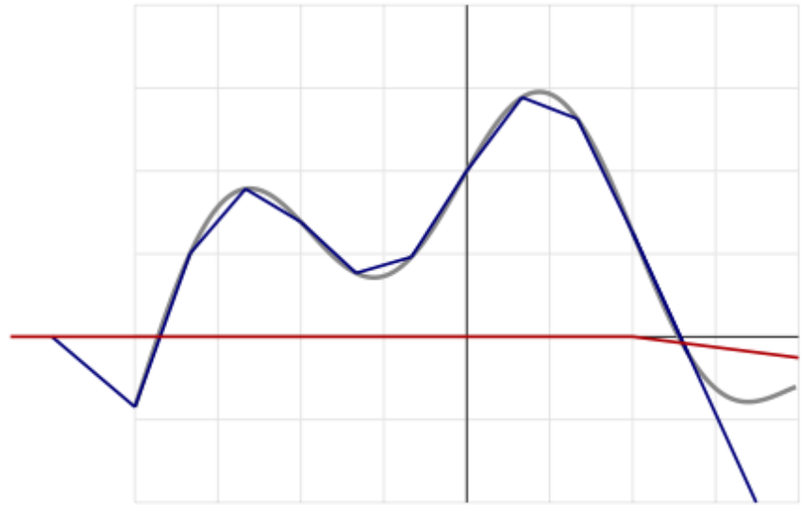
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



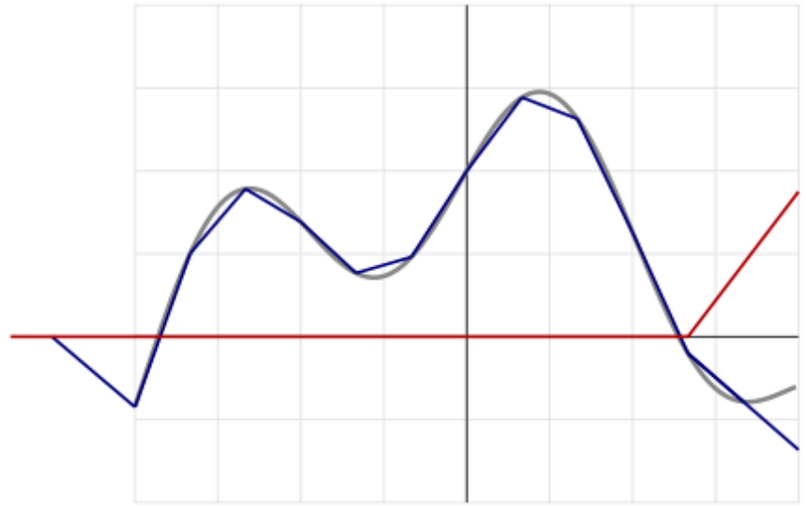
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



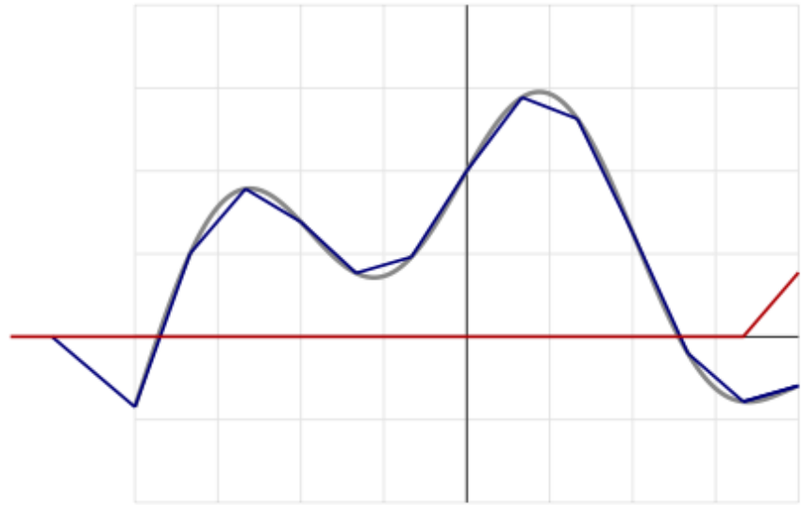
Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



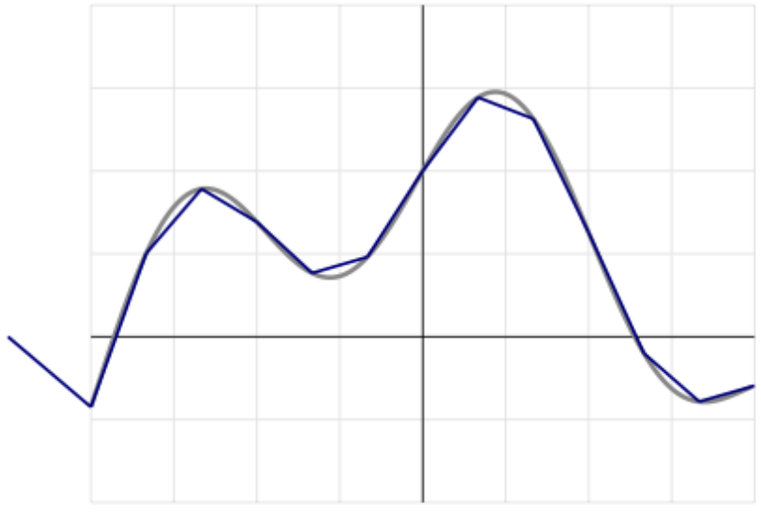
Universal approximation

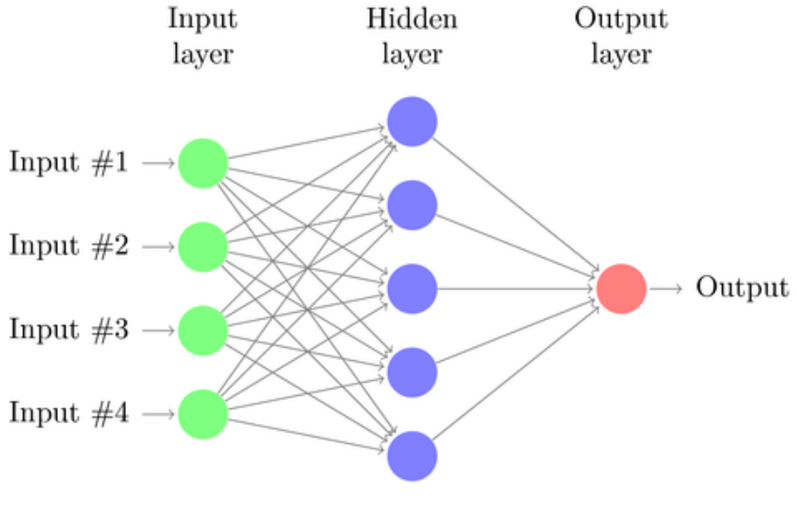
We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions



Universal approximation

We can approximate any $f \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions

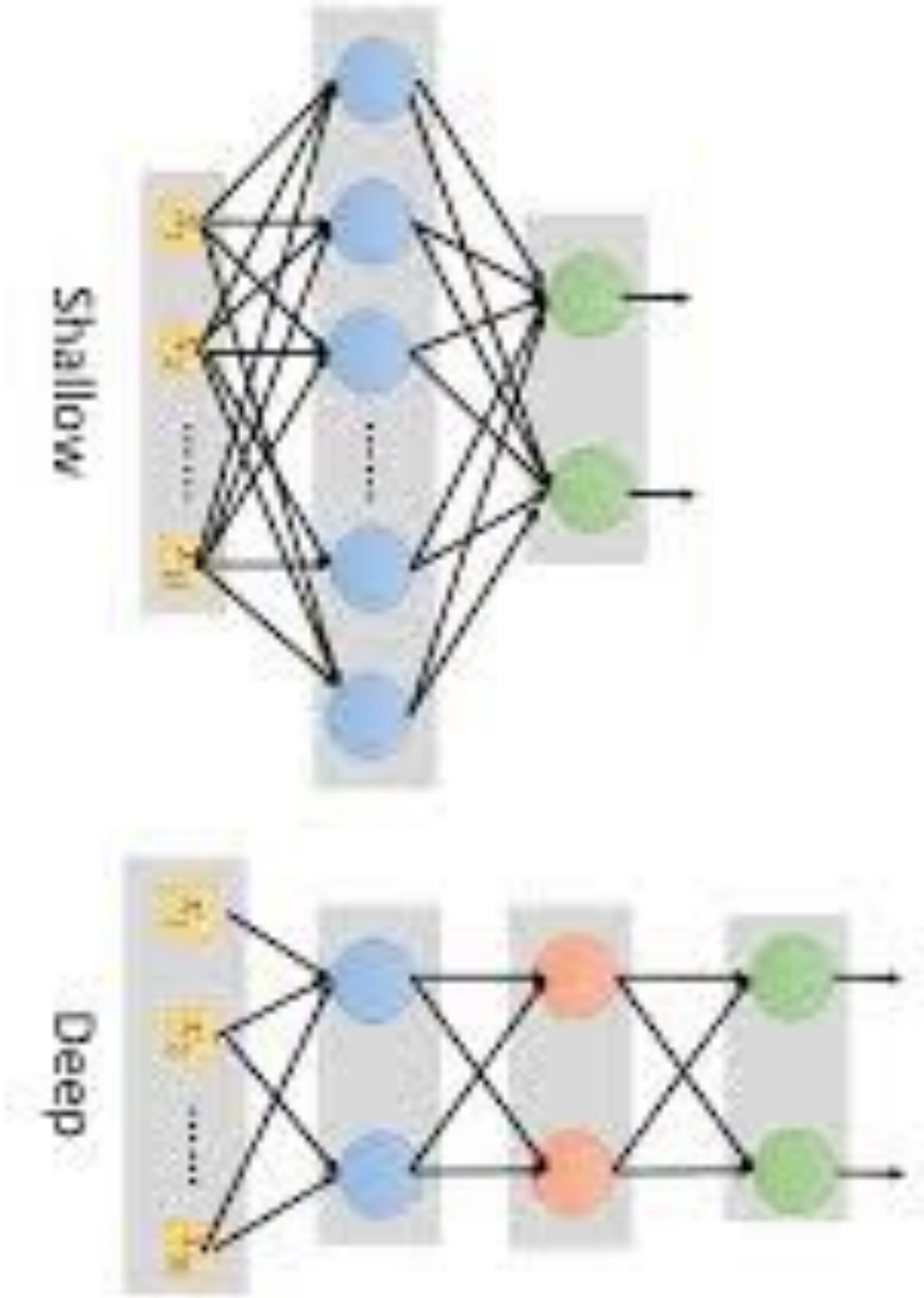




$$Y = \sum_i \alpha_i \mathrm{Relu}(\mathrm{a_i} * \mathrm{x} + \mathrm{b_i})$$

Why deep learning?

Deep vs Shallow



Why deep?

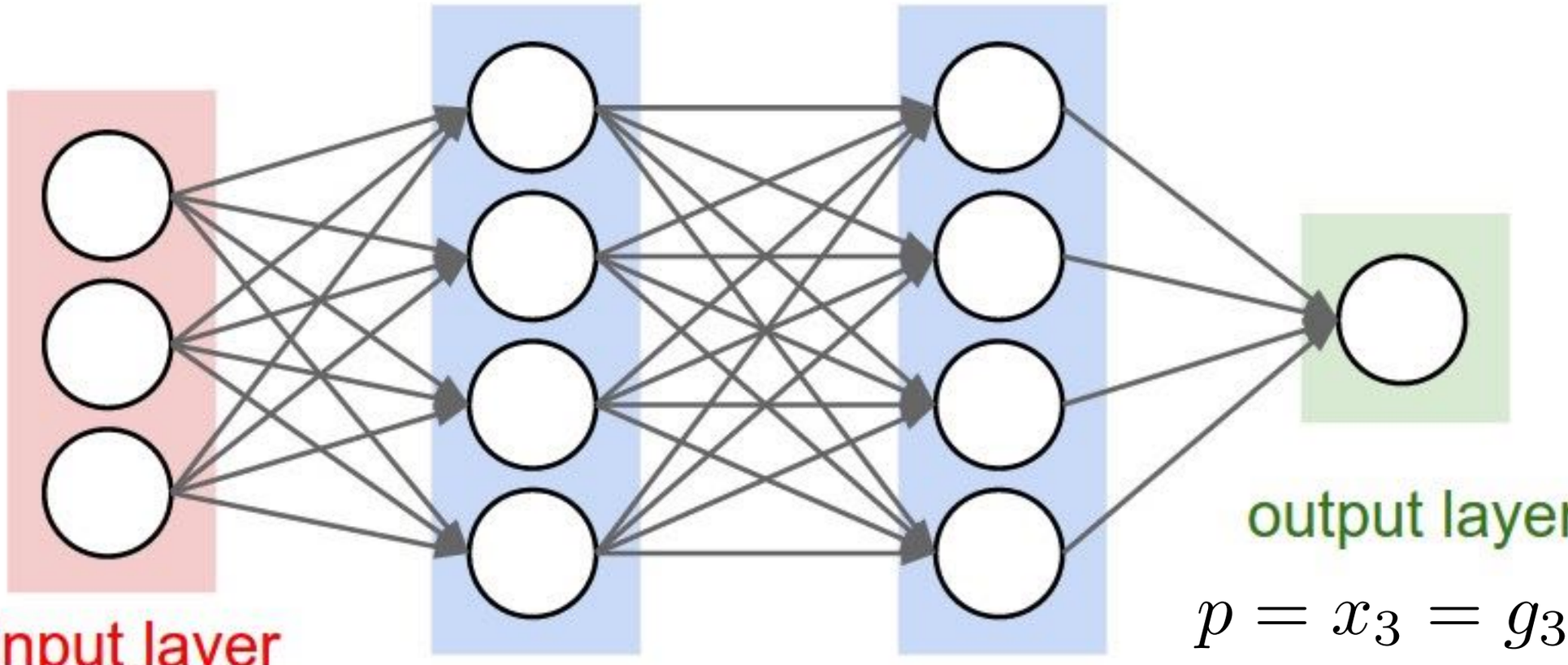
(i) With the same number of nodes, one can represent more complex function with deep networks

(ii) Many data are actually hierarchical…
…just think of the classification of animal species!

(iii) inspiration from the visual cortex (convnet, see next lecture)

(iv) seems to do some weird voodoo magic that somehow limit overfit

Feed-forward Neural networks



$$p = x_3 = g_3(W_3\vec{x}_2)$$

4x1 matrix

$\vec{x}_0$

$$\vec{x}_1 = g_1(W_1\vec{x}_0) \quad \vec{x}_2 = g_2(W_2\vec{x}_1)$$

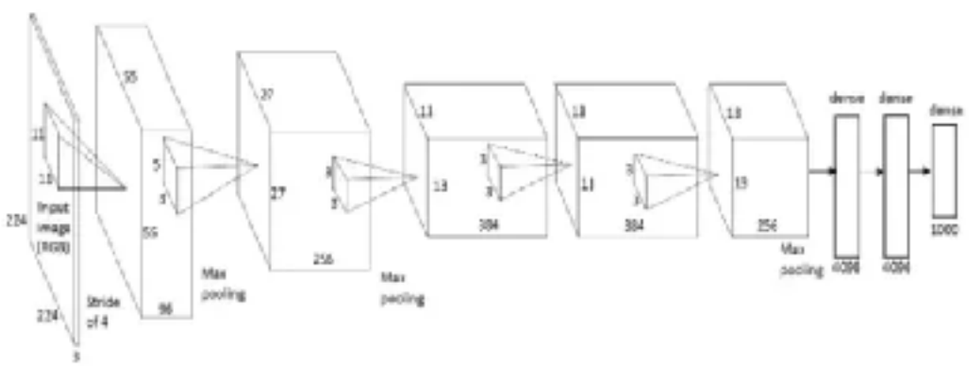
4x3 matrix **4x4 matrix**

$$p = f(\vec{x}_0) = g_3(W_3\ g_2(W_2\ g_1(W_1\vec{x}_0)))$$

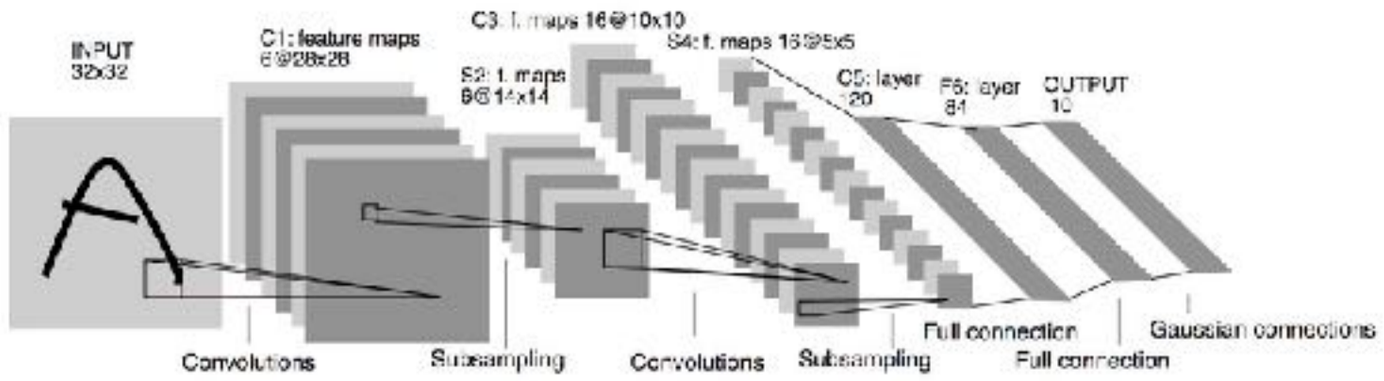
W matrices are called the « *weights* »
The functions g_n ( ) are called « *activation functions* »

Modern neural nets

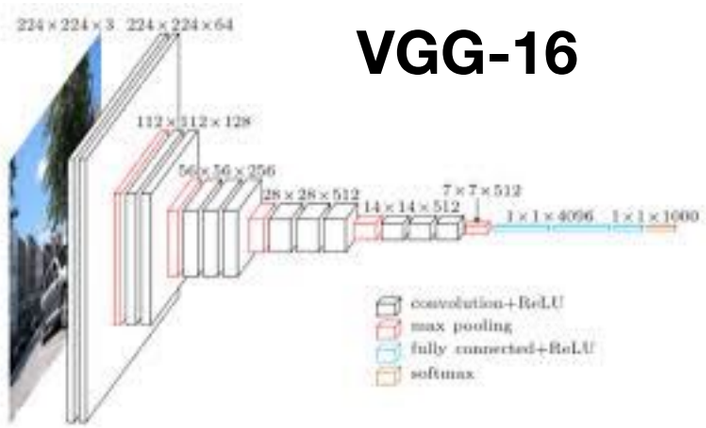
Alexnet



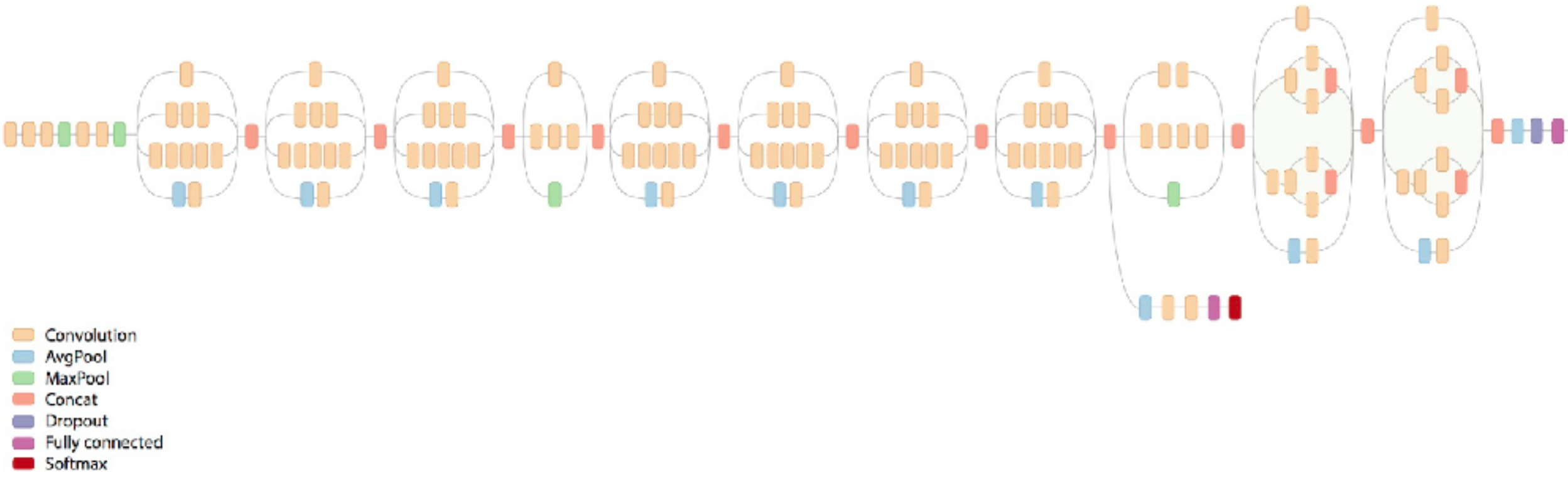
Lenet



VGG-16



Modern neural nets



inception res-net

How do we minimise the empirical risk for the neural network?

Optimization

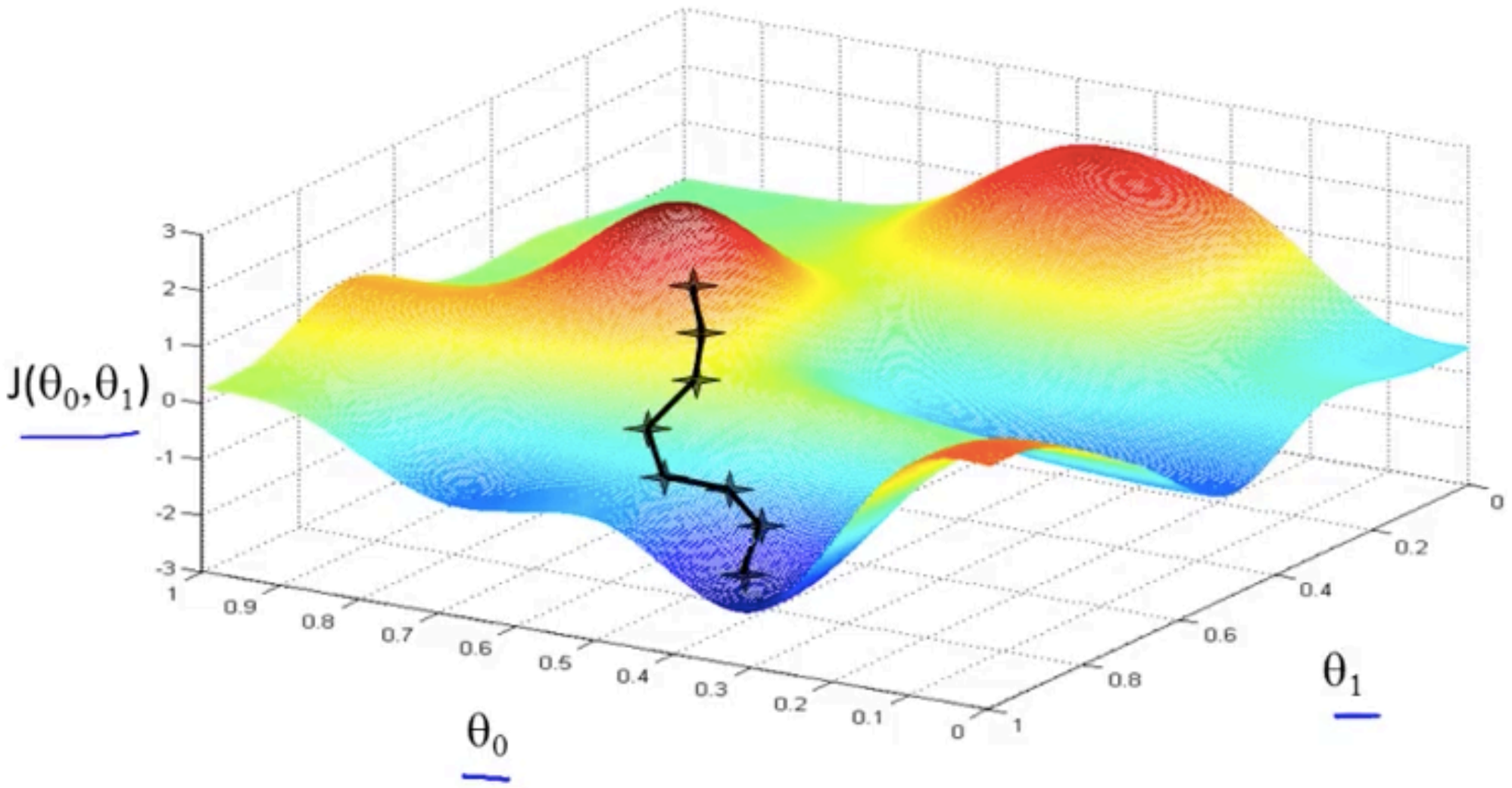
Optimization

Follow the slope!

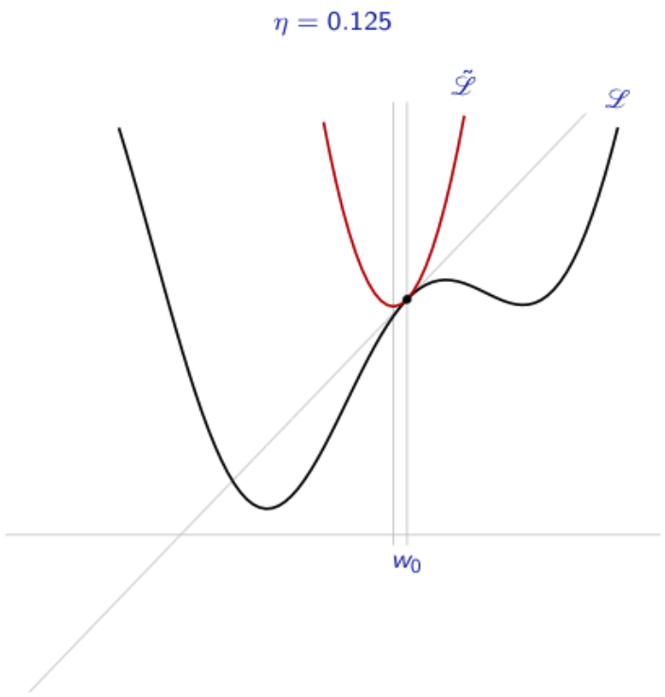
Minimising the cost function by gradient descent

$$\vec{\theta}^{t+1} = \vec{\theta}^{t} - \gamma \nabla R(\vec{\theta}^{t})$$

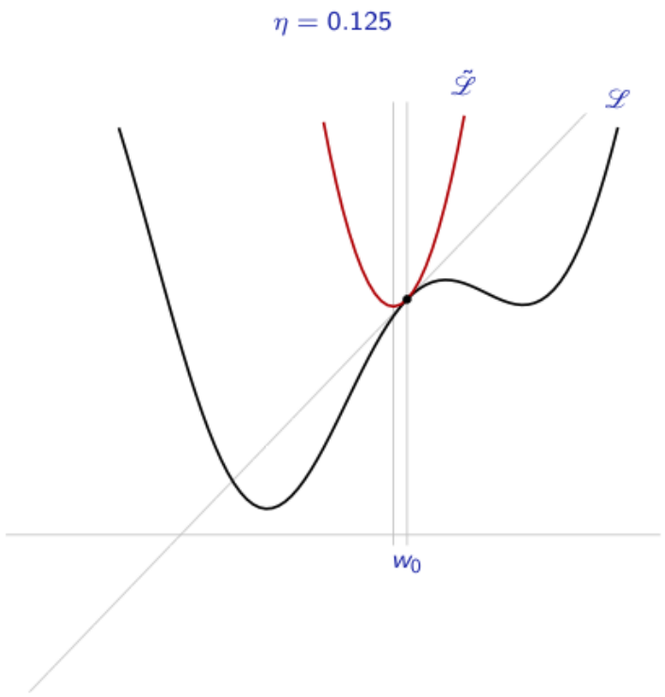
If γ small enough, should converge to a (possible local) minima



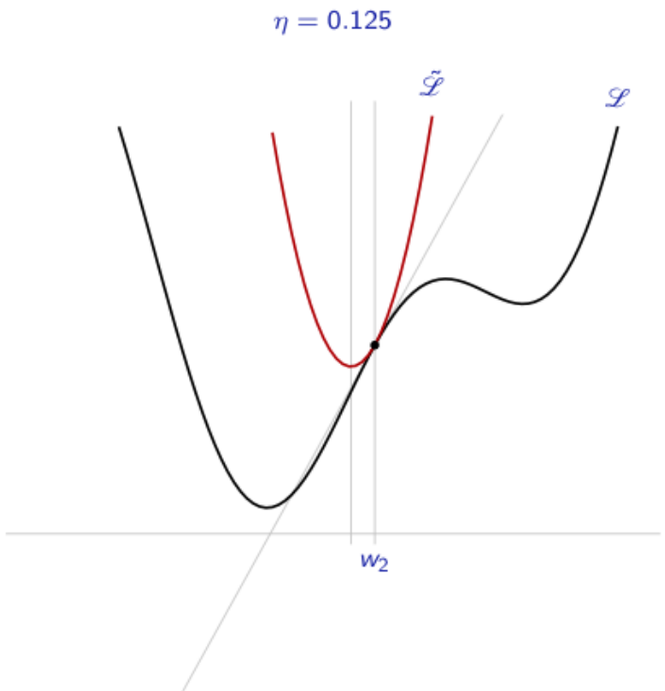
Gradient descent



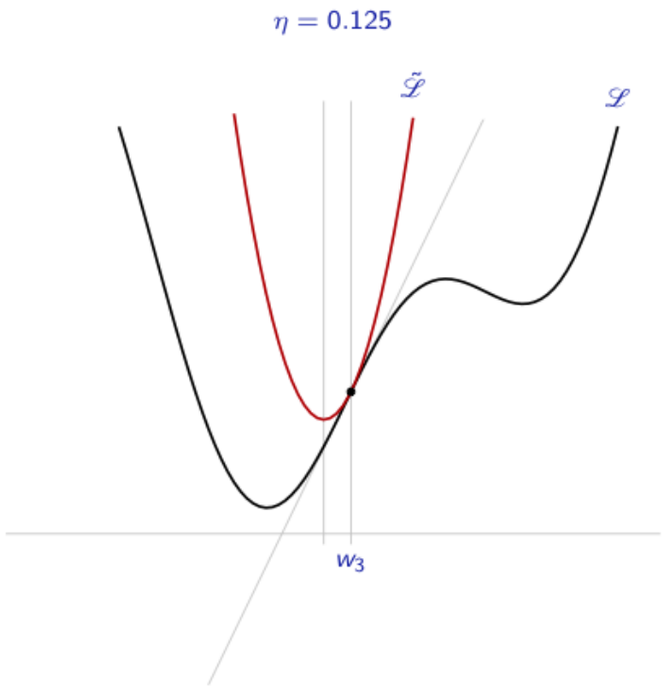
Gradient descent



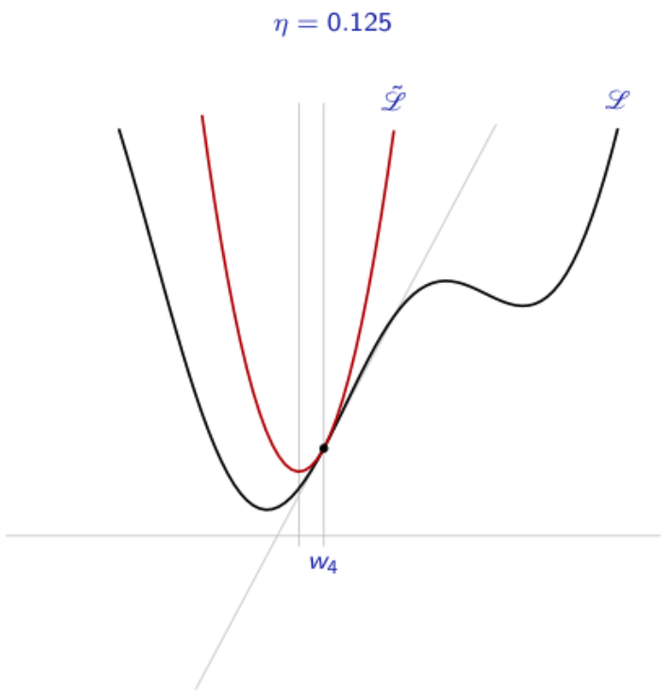
Gradient descent



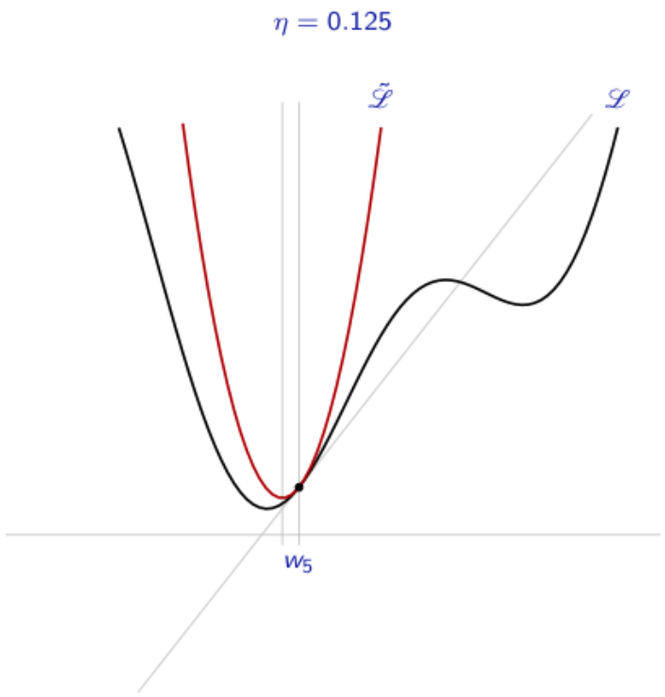
Gradient descent



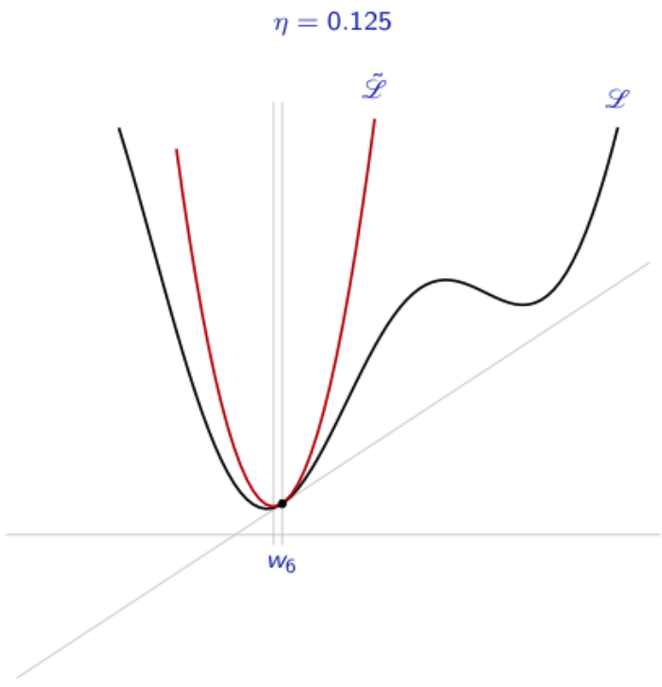
Gradient descent



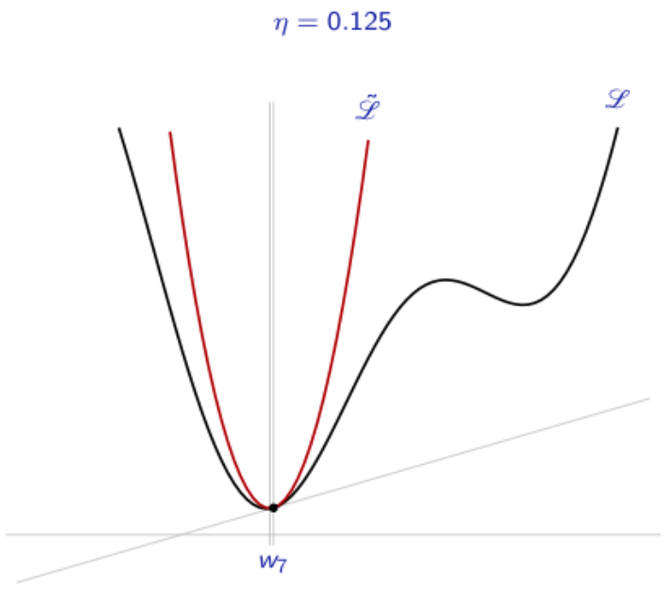
Gradient descent



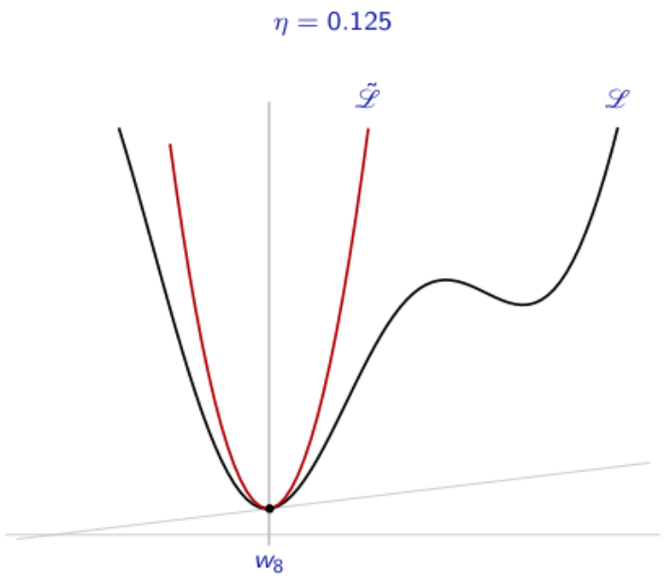
Gradient descent



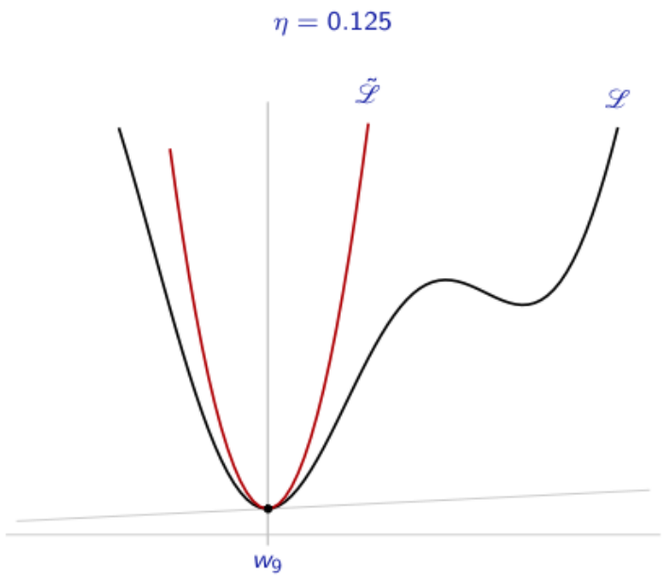
Gradient descent



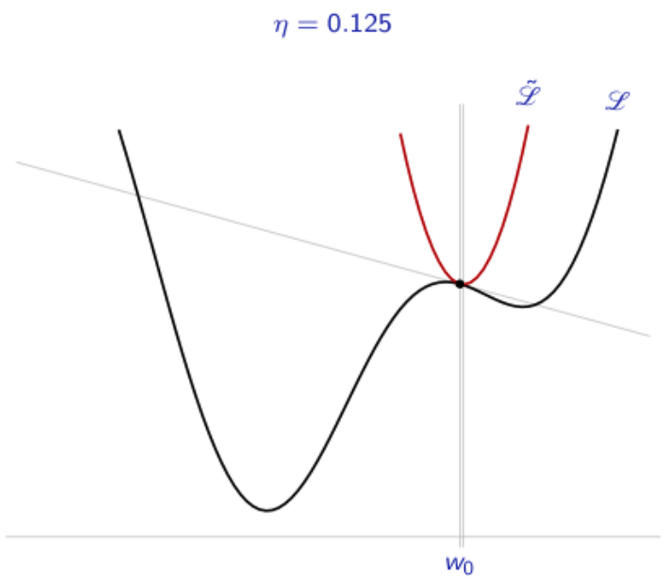
Gradient descent



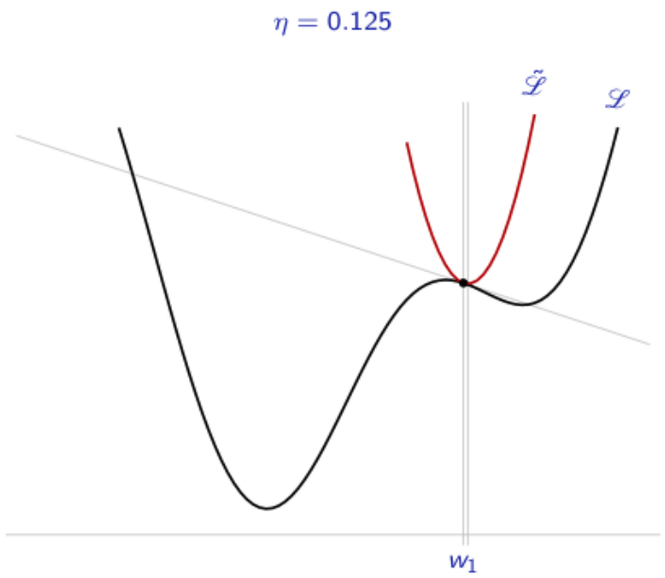
Gradient descent



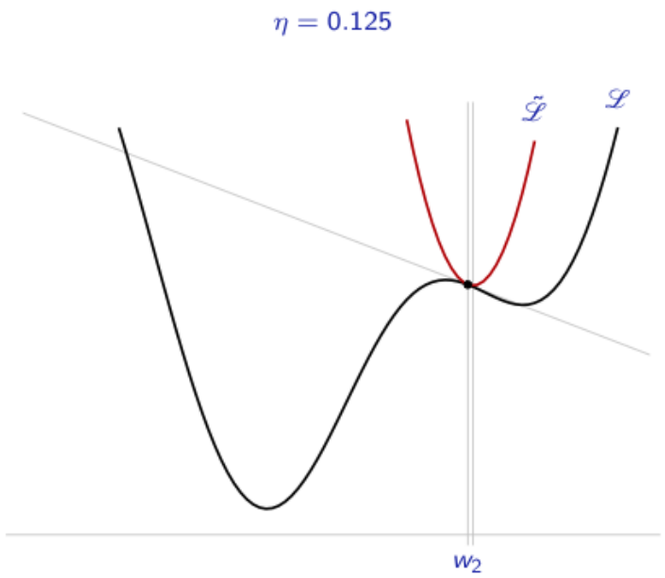
Gradient descent



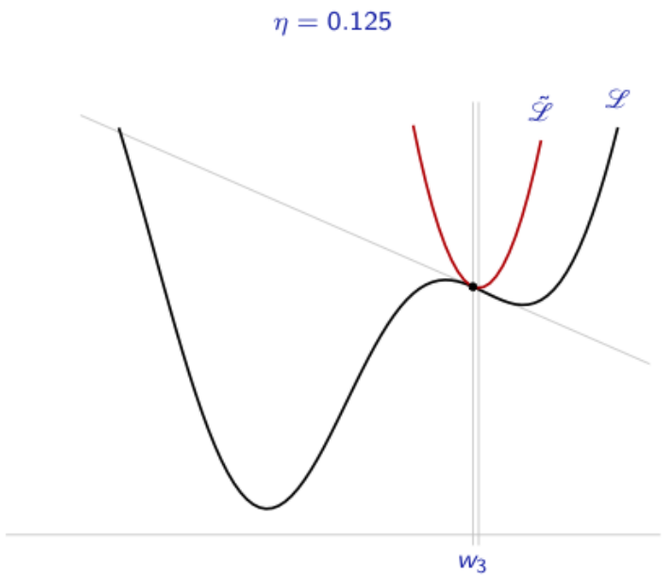
Gradient descent



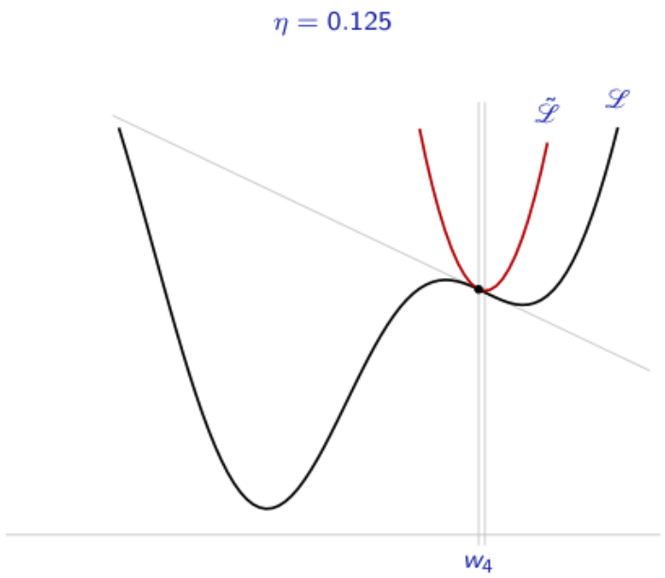
Gradient descent



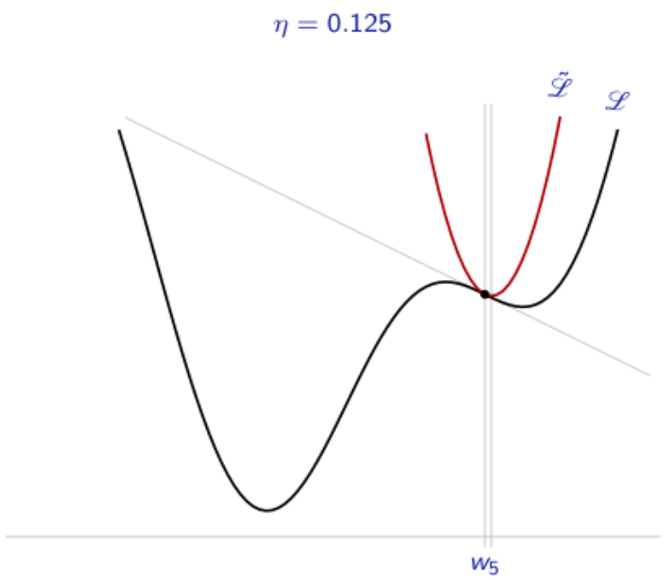
Gradient descent



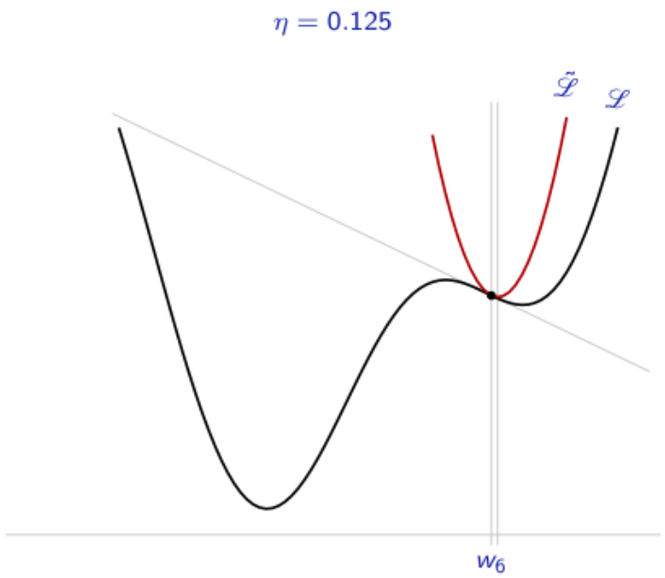
Gradient descent



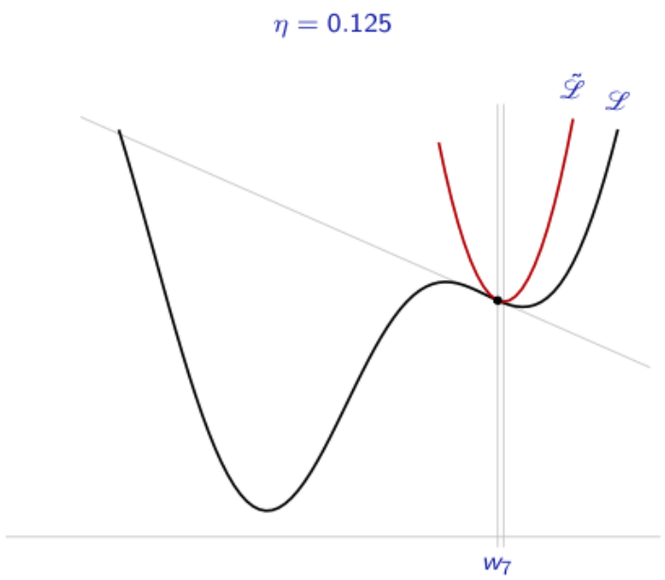
Gradient descent



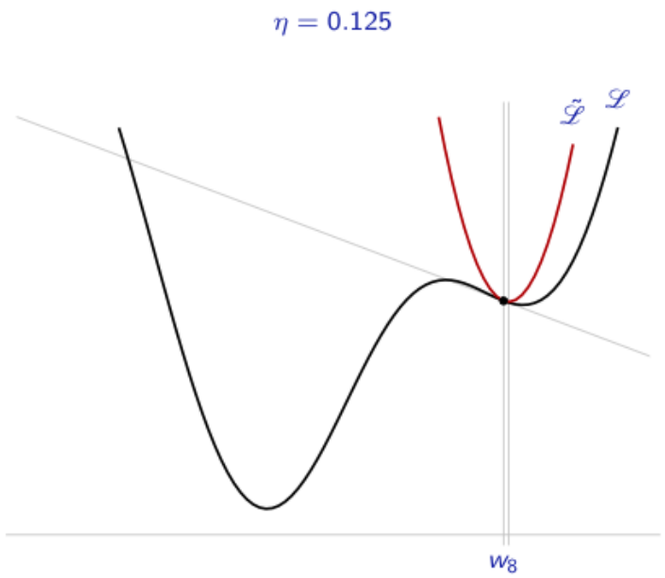
Gradient descent



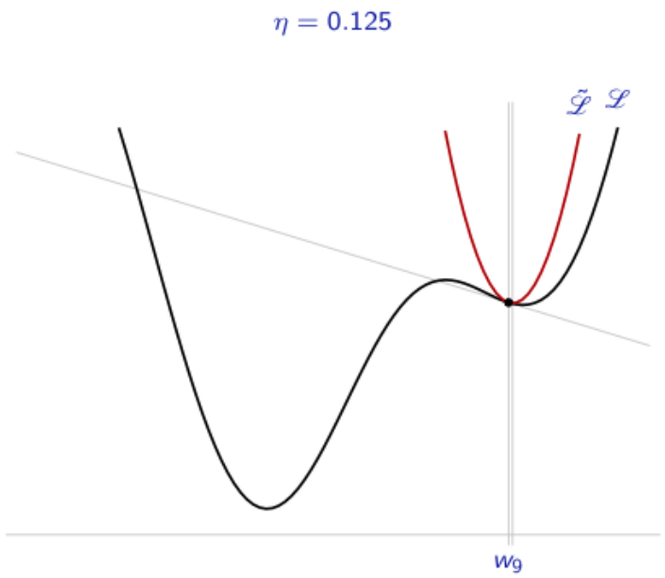
Gradient descent



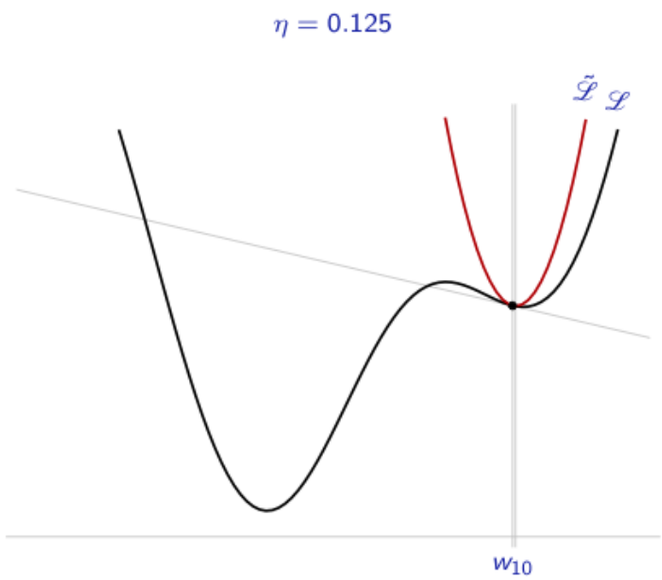
Gradient descent



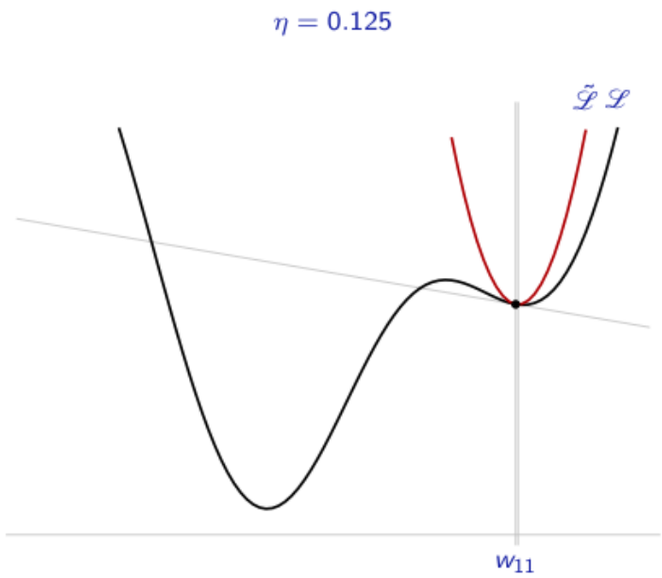
Gradient descent



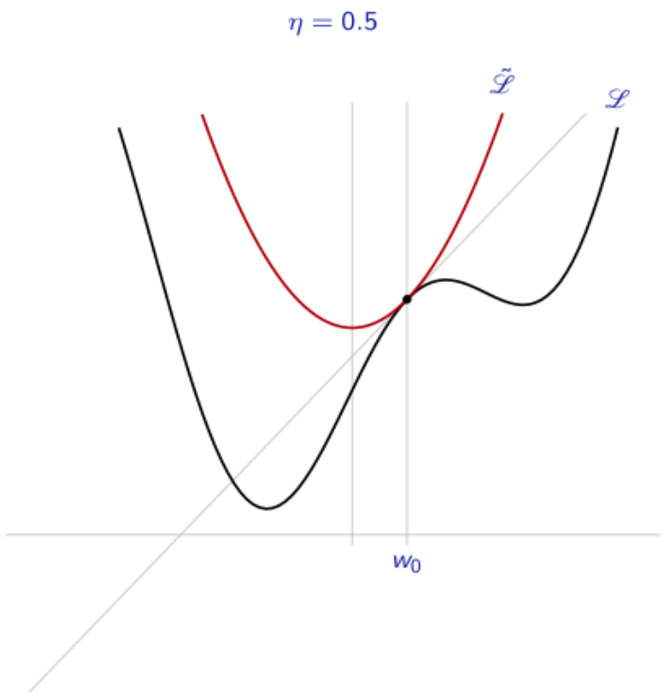
Gradient descent



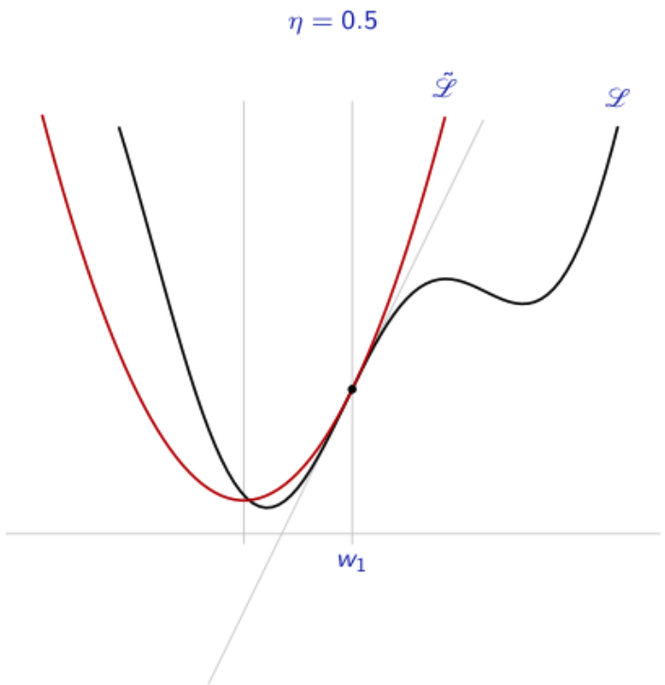
Gradient descent



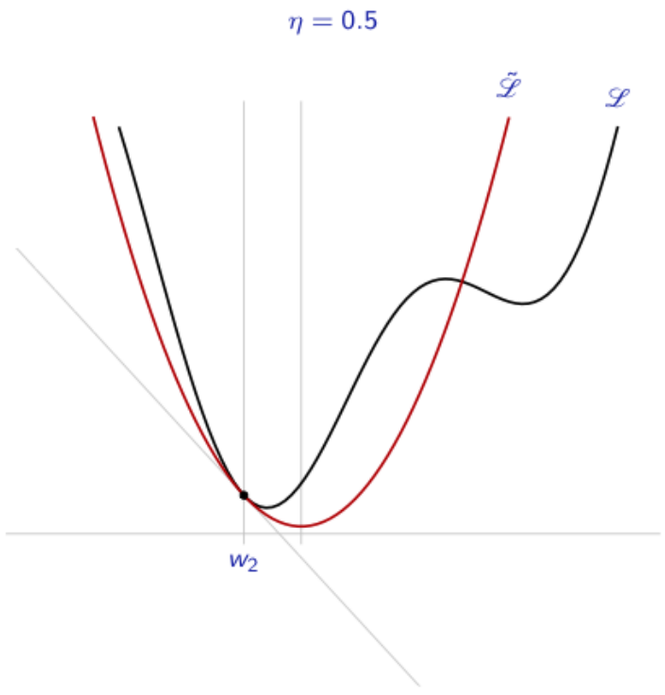
Gradient descent



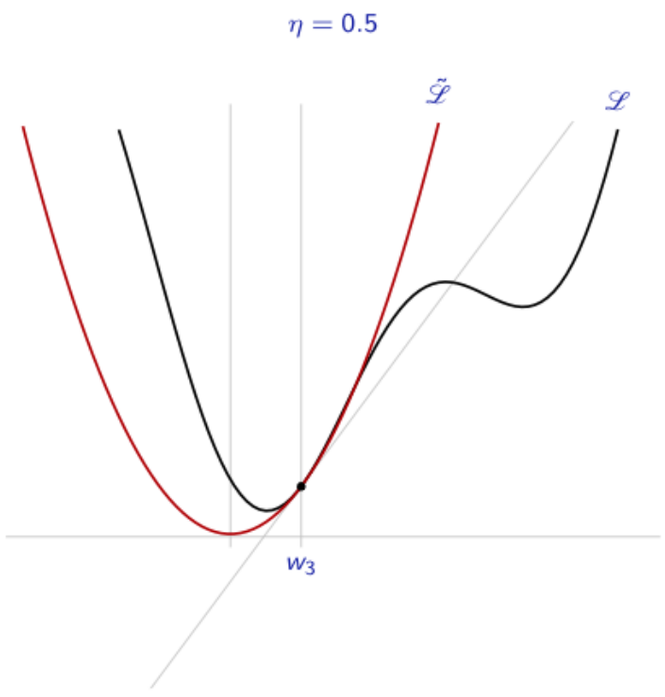
Gradient descent



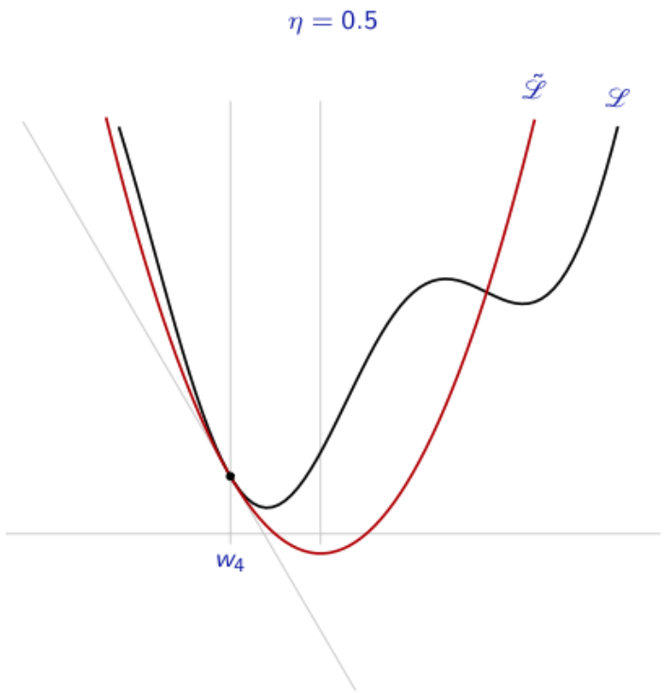
Gradient descent



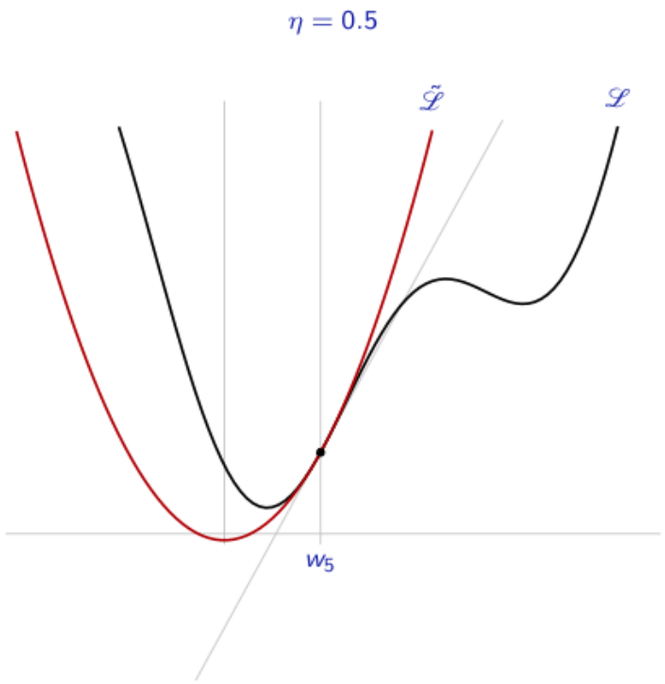
Gradient descent



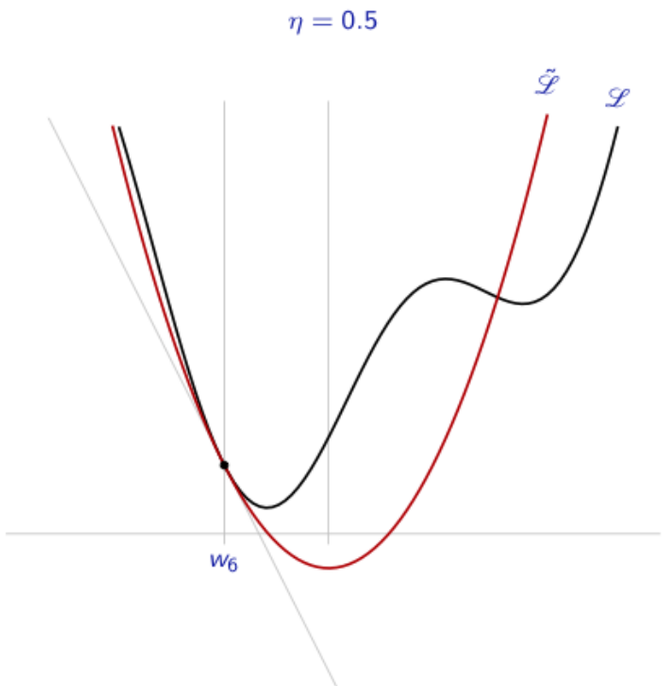
Gradient descent



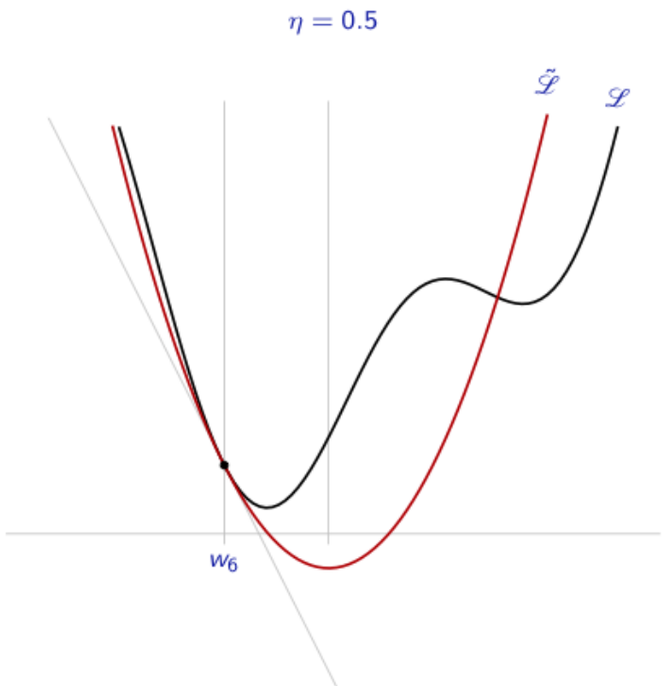
Gradient descent



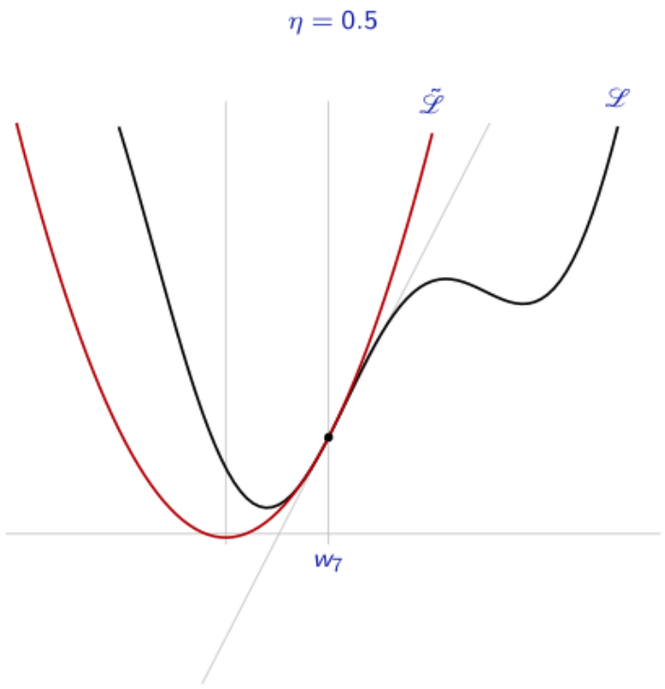
Gradient descent



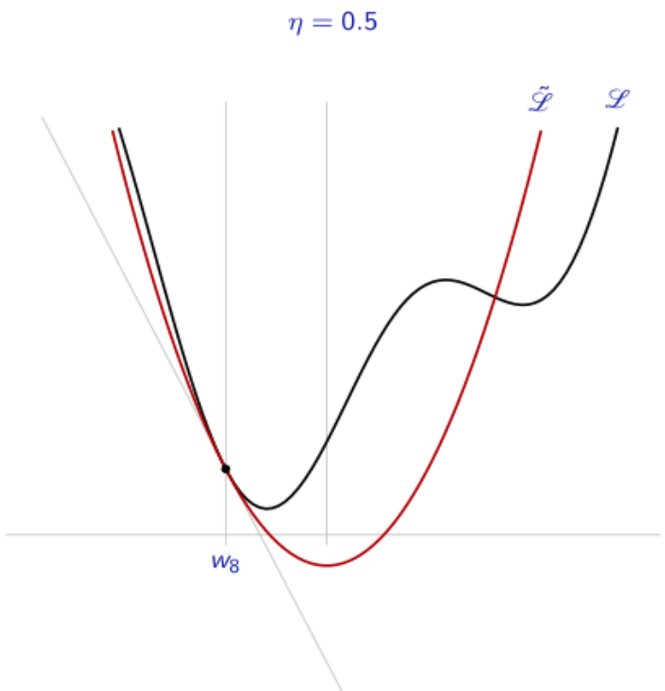
Gradient descent



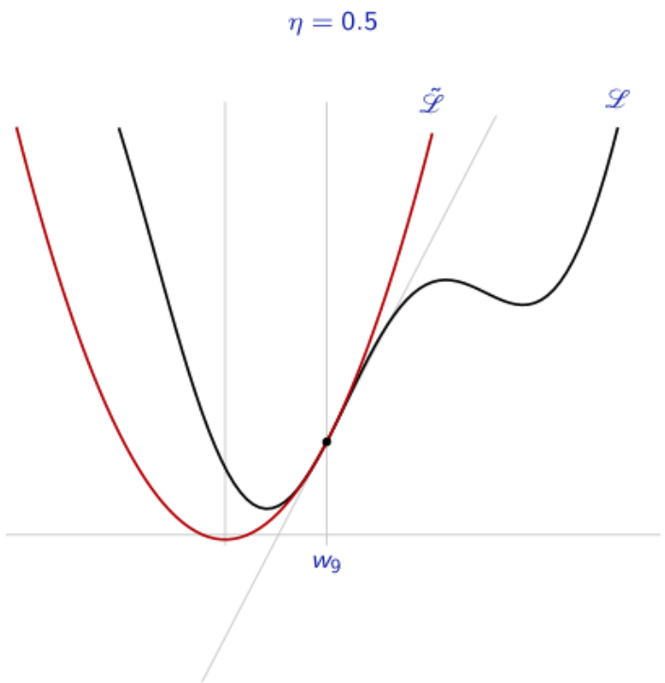
Gradient descent



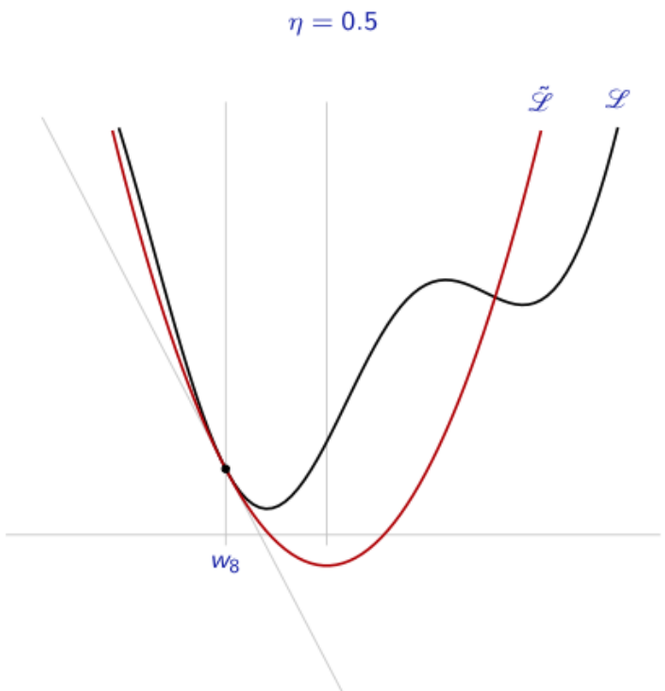
Gradient descent



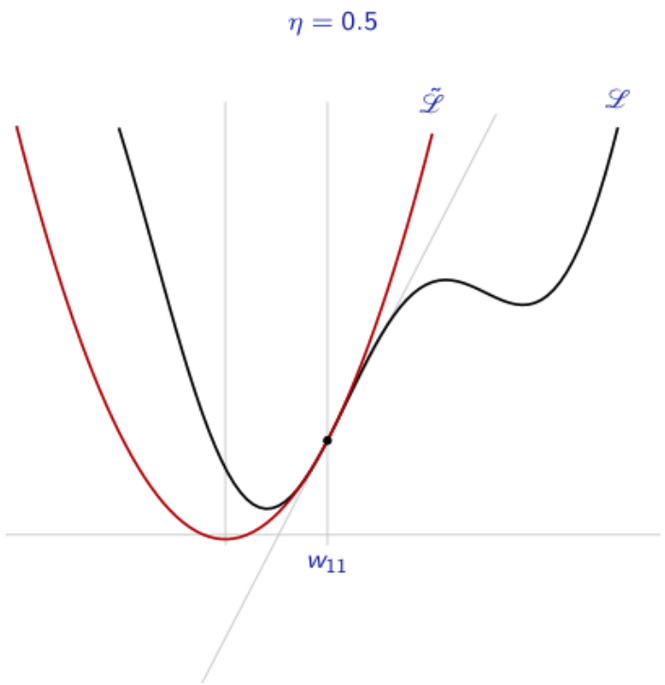
Gradient descent



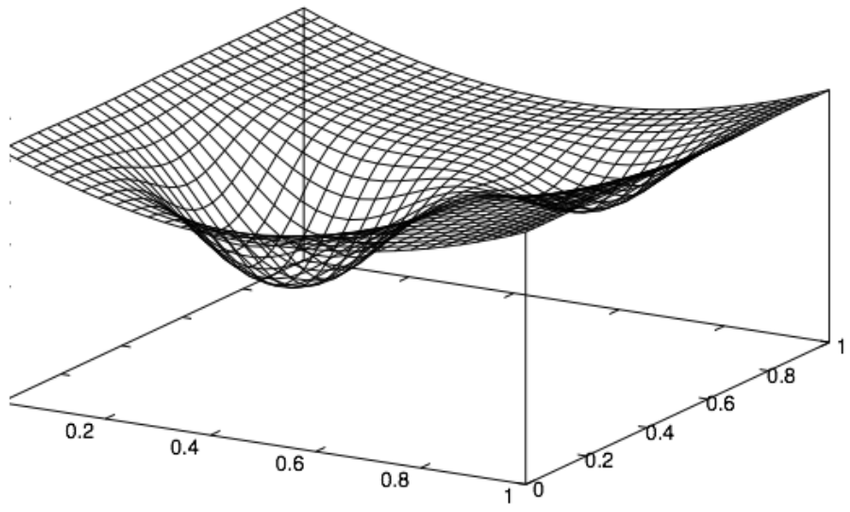
Gradient descent



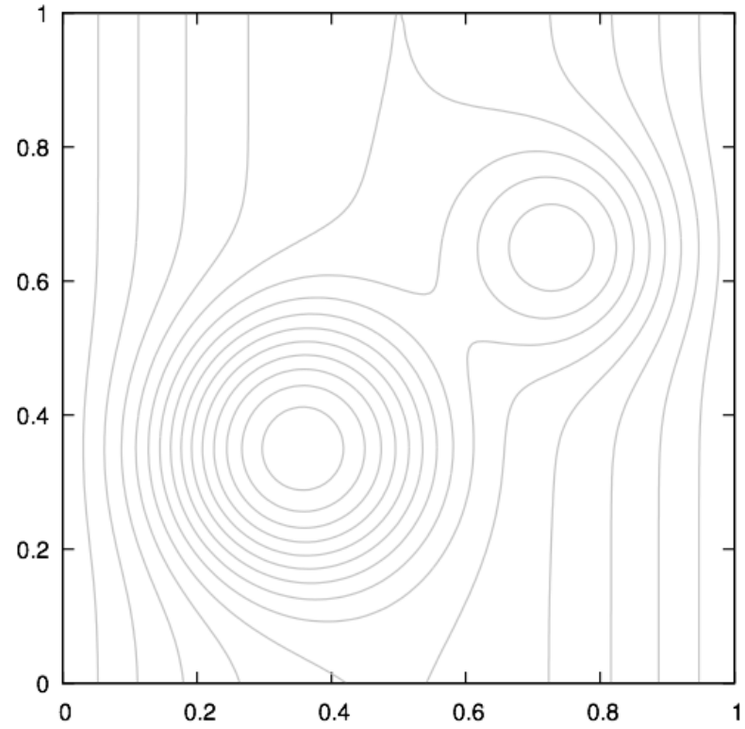
Gradient descent



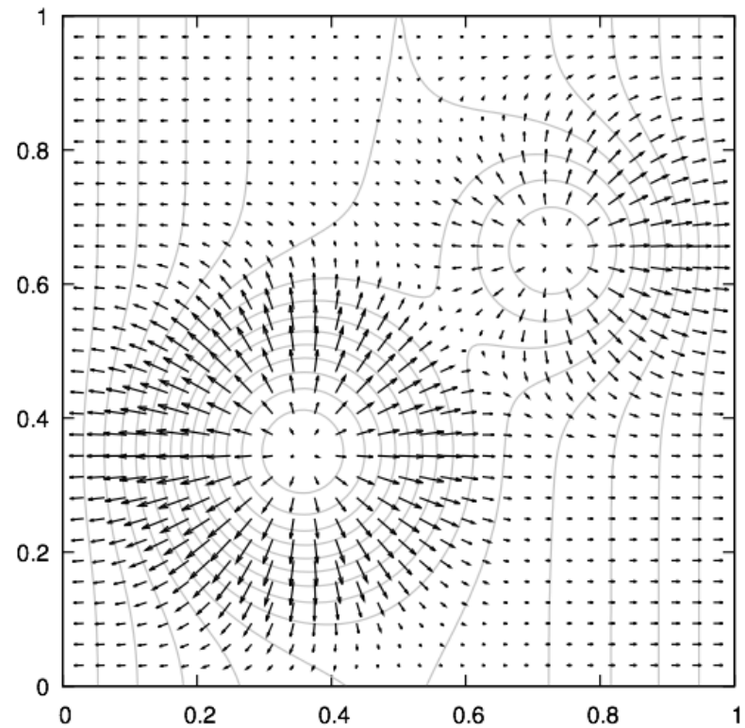
Gradient descent



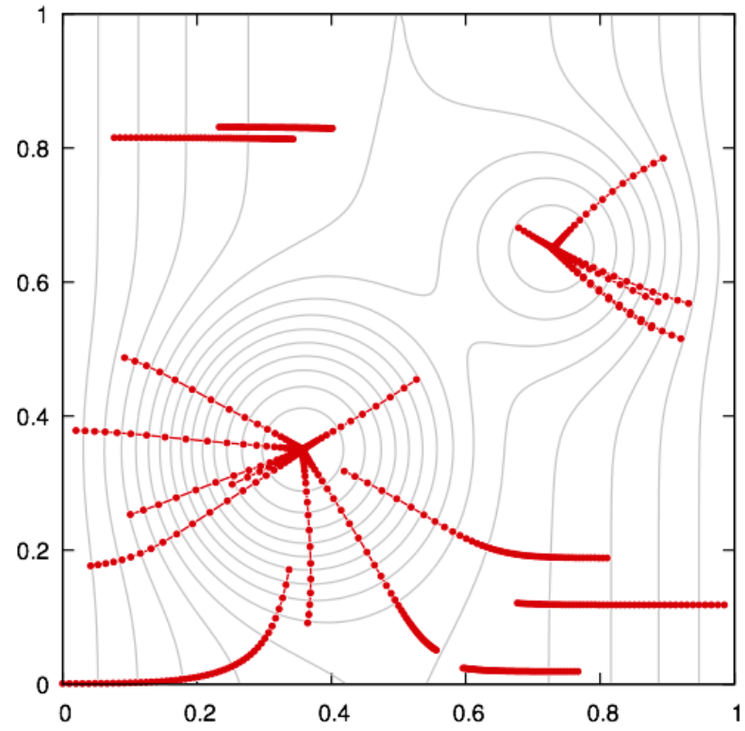
Gradient descent



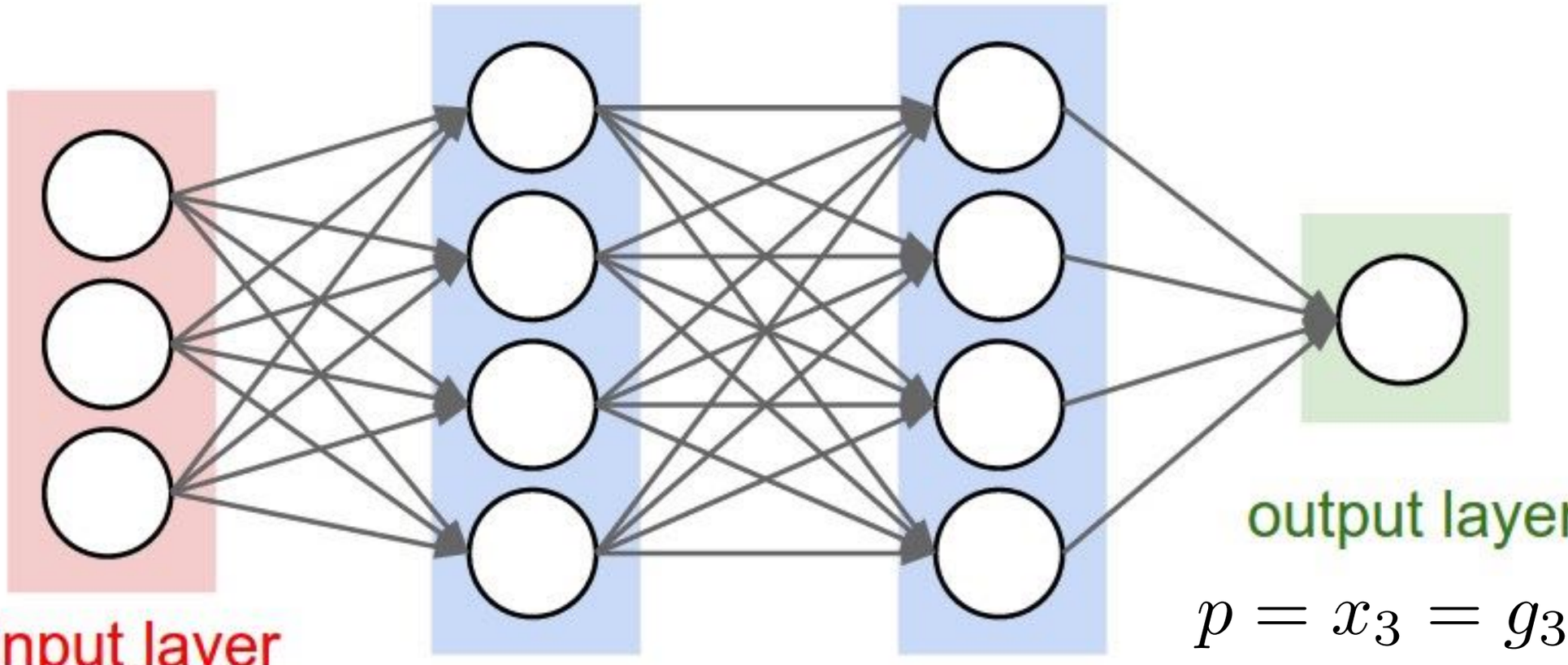
Gradient descent



Gradient descent



Feed-forward Neural networks



$$p = x_3 = g_3(W_3\vec{x}_2)$$

4x1 matrix

$\vec{x}_0$

$$\vec{x}_1 = g_1(W_1\vec{x}_0) \quad \vec{x}_2 = g_2(W_2\vec{x}_1)$$

4x3 matrix **4x4 matrix**

$$p = f(\vec{x}_0) = g_3(W_3\ g_2(W_2\ g_1(W_1\vec{x}_0)))$$

W matrices are called the « *weights* »
The functions g_n ( ) are called « *activation functions* »

How to compute the gradient efficiently?

$$\vec{x}_0 \quad \vec{x}_1 = g_1(\overbrace{W_1\vec{x}_0}^{\vec{h}_1}) \quad \dots \quad \vec{x}_n = g_n(\overbrace{W_n\vec{x}_{n-1}}^{\vec{h}_n}) \quad \dots \quad p = g_L(\overbrace{W_L\vec{x}_{L-1}}^{\vec{h}_L})$$

Feed-forward

Compute the loss $L = \frac{(y-p)^2}{2}$

Back-propagation of errors

$$e_j^1 = g_1{}'(h_j^1)\sum_i W_{ij}^2 e_i^2 \quad \dots \quad e_j^n = g_n{}'(h_j^n)\sum_i W_{ij}^{n+1} e_i^{n+1} \quad \dots \quad e^L = g_L{}'(h^L)(p-y)$$

Once this is done, gradients are given by $\frac{\partial L}{\partial W_{ab}^l} = x_b^{l-1} e_a^l$

Demonstration by the chain rule of derivatives

$$L = \frac{(y-p)^2}{2} \qquad \frac{\partial L}{\partial w_{ab}^{(l)}} = \textbf{?}$$

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \overbrace{(p-y)g'^{(L)}(h^{(L)})}^{e^L = {g_L}'(h^L)(p-y)} \sum_k w_k^{(L)} \frac{\partial x_k^{(L-1)}}{\partial w_{ab}^{(l)}} \quad \Rightarrow \quad \frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k w_k^{(L)} \frac{\partial x_k^{(L-1)}}{w_{ab}^{(l)}} e^L$$

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k w_k^{(L)} \left(\frac{\partial}{\partial w_{ab}^{(l)}} g^{(L-1)} \left[\sum_{k'} w_{kk'}^{(L-1)} x_{k'}^{(L-2)} \right] \right) e^L$$

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_{k'} \frac{\partial x_{k'}^{(L-2)}}{\partial w_{ab}^{(l)}} \sum_k w_{kk'}^{(L-1)} \underbrace{w_k^{(L)} \left(g^{(L-1)'}[h_k^{L-1}] \right) e^L}_{e_k^{L-1}} = \sum_{k'} \frac{\partial x_{k'}^{(L-2)}}{\partial w_{ab}^{(l)}} \sum_k w_{kk'}^{(L-1)} e_k^{L-1}$$

...

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k \frac{\partial x_k^{(n-2)}}{w_{ab}^{(l)}} \sum_i w_{ik}^{(n-1)} e_i^{(n-1)}$$

...

$$\frac{\partial L}{\partial w_{ab}^{(l)}} = \sum_k \frac{\partial x_k^{(l)}}{w_{ab}^{(l)}} \sum_i w_{ik}^{(l+1)} e_i^{(l+1)} = x_b^{(l-1)} e_a^{(l)}$$

How to compute the gradient efficiently?

$$\vec{x}_0 \quad \vec{x}_1 = g_1(\overbrace{W_1\vec{x}_0}^{\vec{h}_1}) \quad \dots \quad \vec{x}_n = g_n(\overbrace{W_n\vec{x}_{n-1}}^{\vec{h}_n}) \quad \dots \quad p = g_L(\overbrace{W_L\vec{x}_{L-1}}^{\vec{h}_L})$$

Feed-forward

Compute the loss $L = \frac{(y-p)^2}{2}$

Back-propagation of errors

$$e_j^1 = g_1{}'(h_j^1)\sum_i W_{ij}^2 e_i^2 \quad \dots \quad e_j^n = g_n{}'(h_j^n)\sum_i W_{ij}^{n+1} e_i^{n+1} \quad \dots \quad e^L = g_L{}'(h^L)(p-y)$$

Once this is done, gradients are given by $\frac{\partial L}{\partial W_{ab}^l} = x_b^{l-1} e_a^l$

Minimising the cost function by gradients descent

$$\vec{\theta}^{t+1} = \vec{\theta}^{t} - \gamma \nabla R(\vec{\theta}^{t})$$

If γ small enough, converge to a (possible local) minima

Standard (or "batch") gradient descent

Compute the gradient by averaging the derivative of the loss is the entire training set

$$\vec{\theta}^{t+1} = \vec{\theta}^{t} - \gamma \sum_{i} \frac{1}{N} \nabla l(\vec{\theta}^{t}; \vec{x}_i, y_i)$$

Minimising the cost function by gradients descent

$$\vec{\theta}^{t+1} = \vec{\theta}^{t} - \gamma \nabla R(\vec{\theta}^{t})$$

If γ small enough, converge to a (possible local) minima

Stochastic (or « mini-batch") gradient descent

Compute the gradient by averaging the derivative of the loss in a **mini-batch**

1) Divide the training set into P batch of size B

2) For each batch, do

$$\vec{\theta}^{t+\frac{1}{P}} = \vec{\theta}^{t} - \gamma \sum_{i \text{ in mini batch}} \frac{1}{B} \nabla l(\vec{\theta}^{t}; \vec{x}_i, y_i)$$

3) One « epoch » (t->t+1) means running the algorithm through all mini-batches

Why Mini-batch gradient descent?

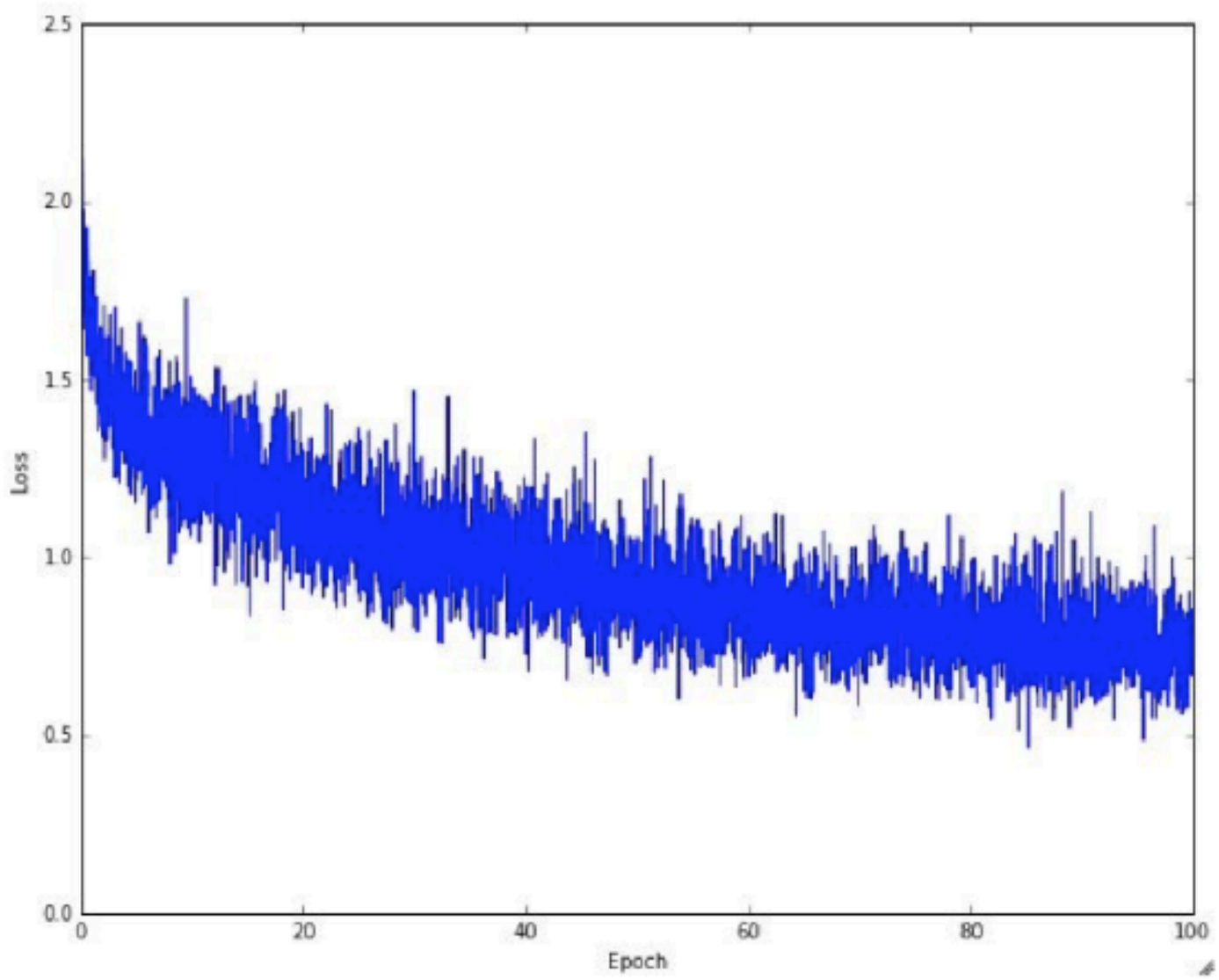
$$\vec{\theta}^{t+\frac{1}{P}} = \vec{\theta}^{t} - \gamma \sum_{i \text{ in mini batch}} \frac{1}{B} \nabla l(\vec{\theta}^{t}; \vec{x}_i, y_i)$$

- The model update frequency is higher than batch gradient descent: **faster** and **memory efficient** *(often nothing else is actually possible)*

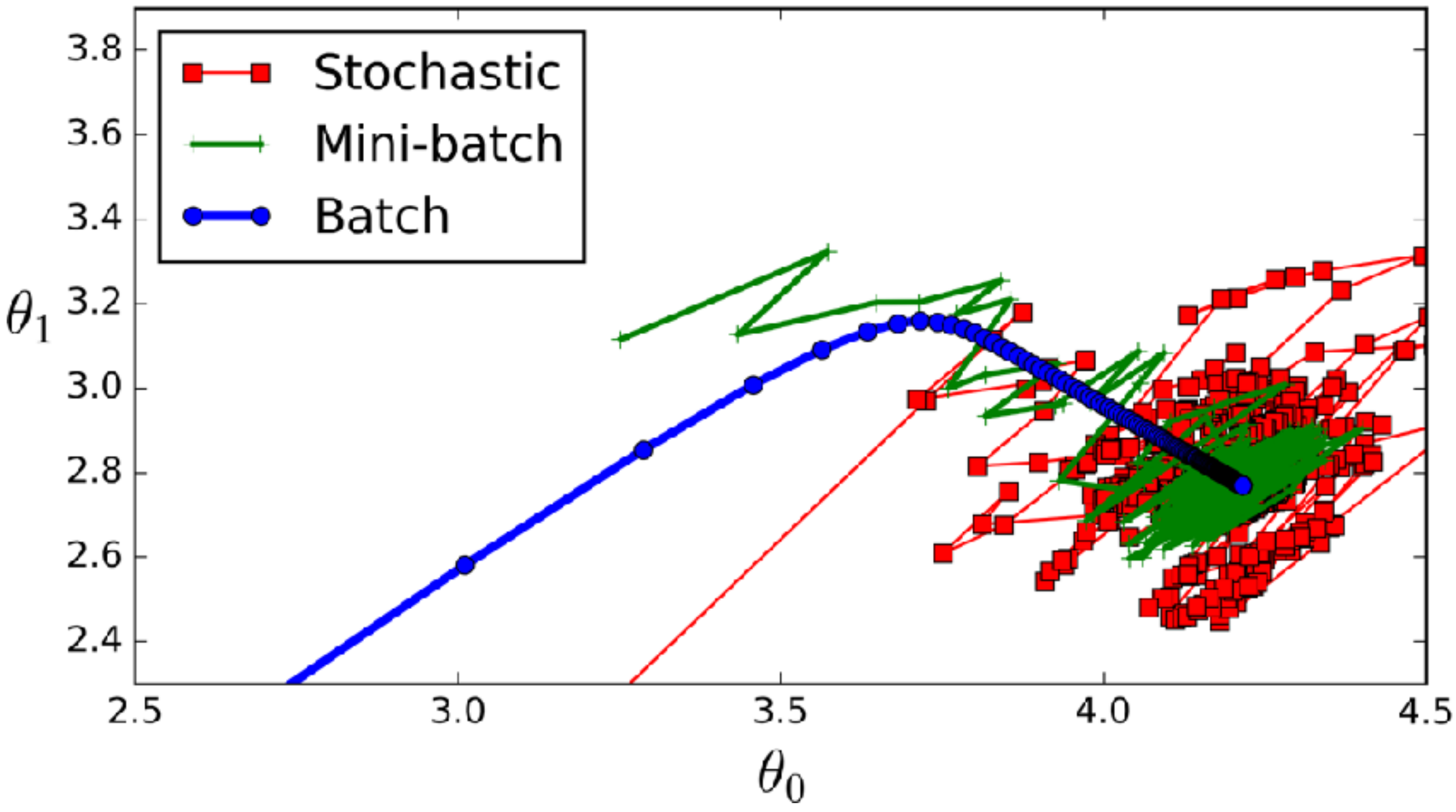
- Effective noise in the dynamics helps optimization/regularization: works better than full batch minimisation in practice

Mini-batch gradient descent

- Example of optimization progress while training a neural network
- Showing loss over mini-batches as it goes down over time

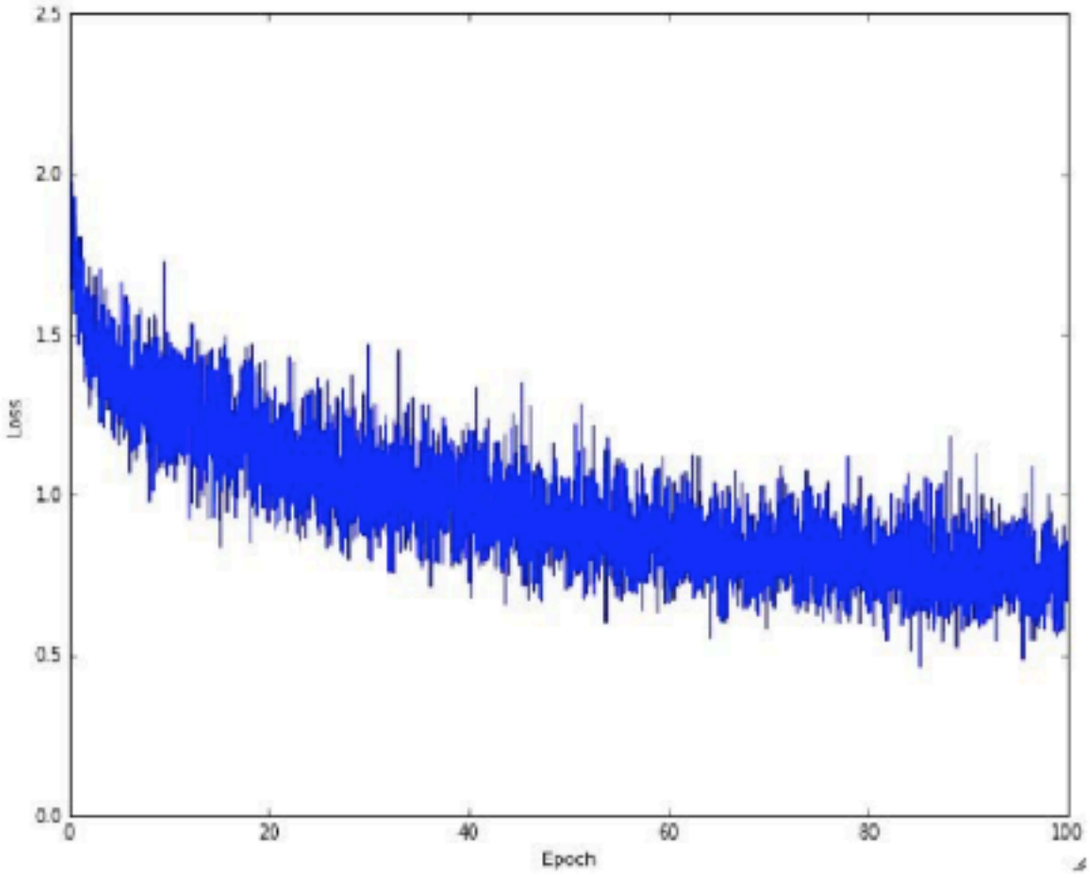


Batch vs mini-batches

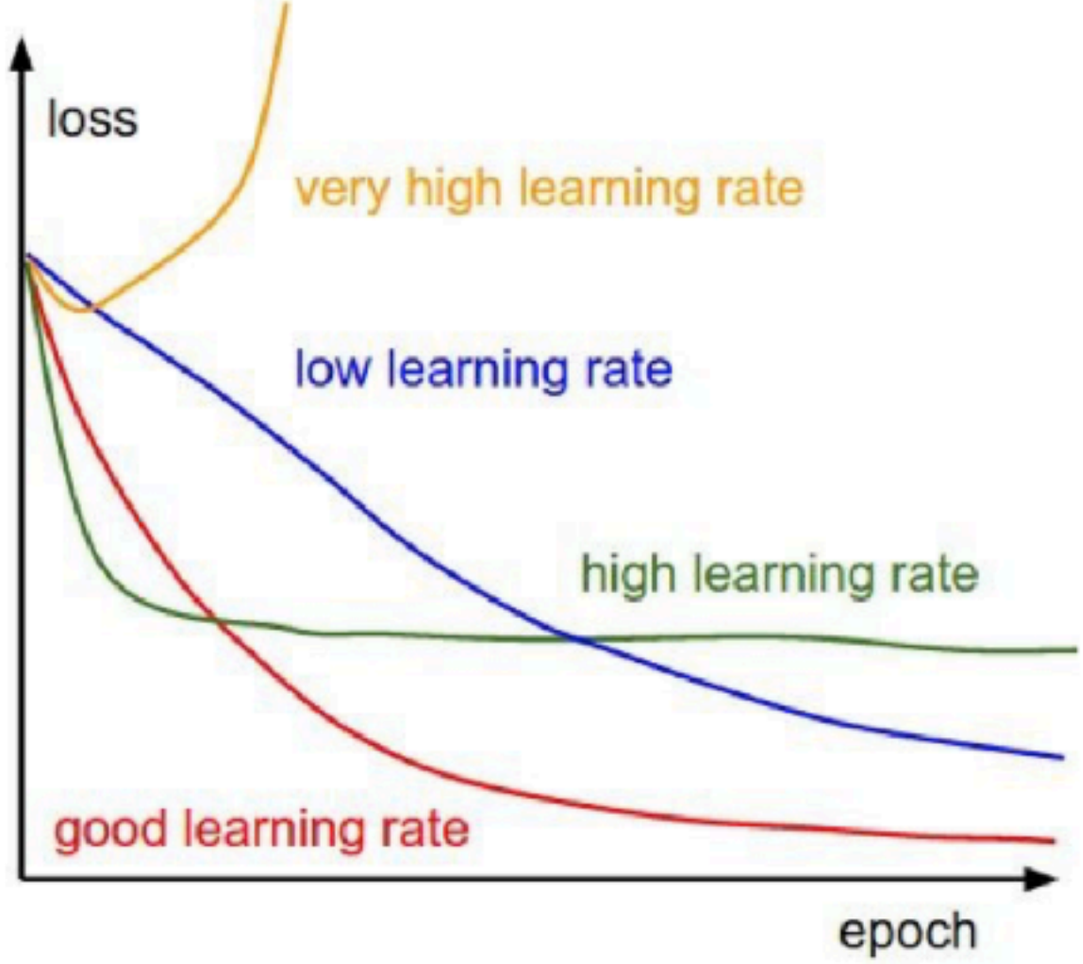


Mini-batch gradient descent

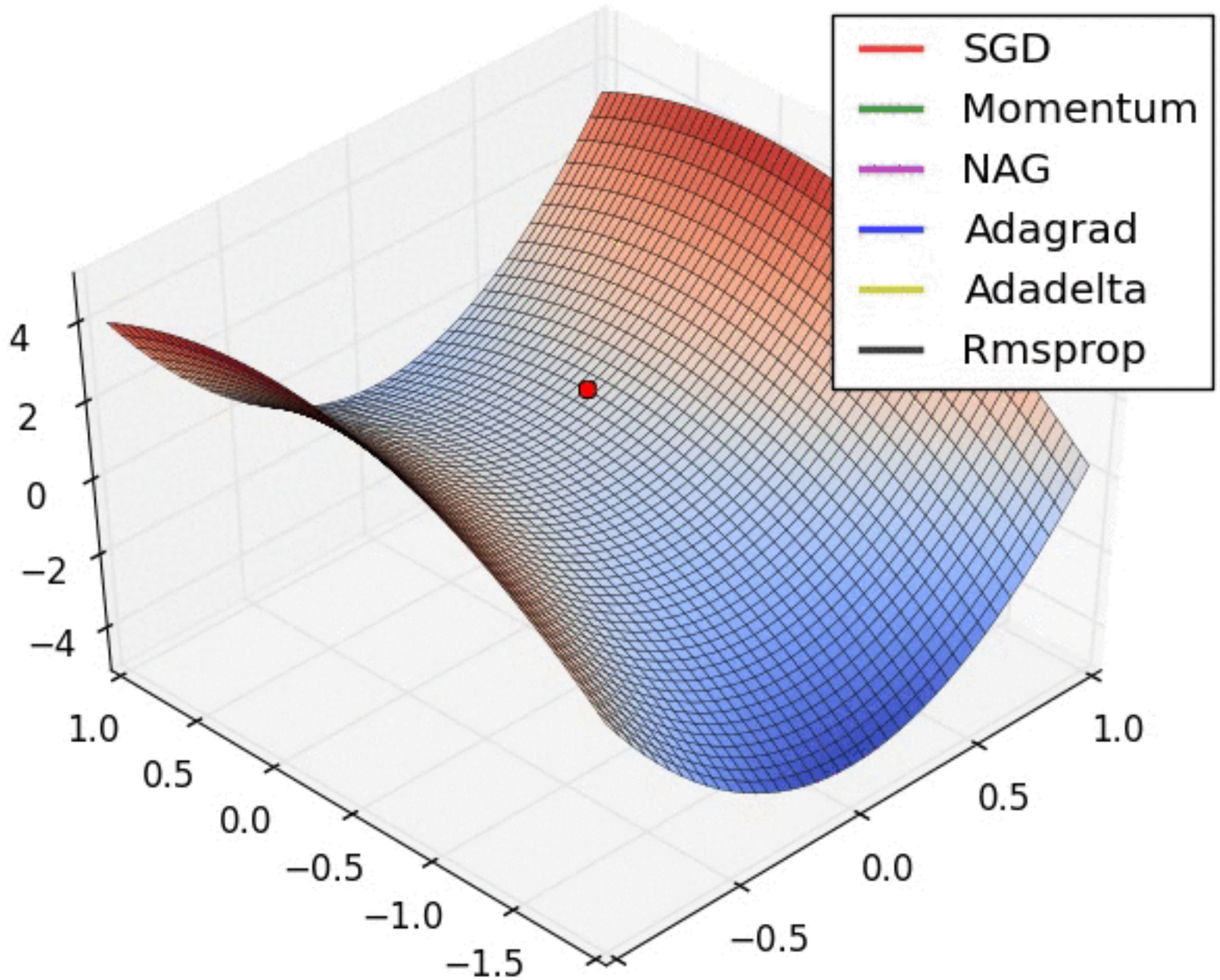
- Example of optimization progress while training a neural network
- **Epoch** = one full pass of the training dataset through the network



The effects of step size (or "learning rate")



Many mini-batch algorithms
(but we shall discuss them later)



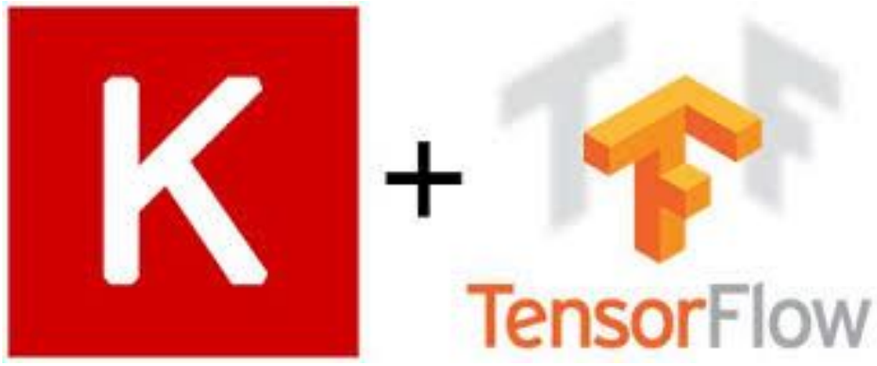
Using Neural nets!

Many Python Frameworks

- Pytorch & Torch
- TensorFlow
- Caffe
- Caffe2
- Chainer
- CNTK
- DSSTNE
- DyNet
- Gensim
- Gluon
- Keras
- Mxnet
- Paddle
- BigDL
- RIP: Theano & Ecosystem

Keras

Keras is an open source neural network library written in Python.
It is capable of running on top of MXNet, Deeplearning4j, Tensorflow, CNTK or Theano

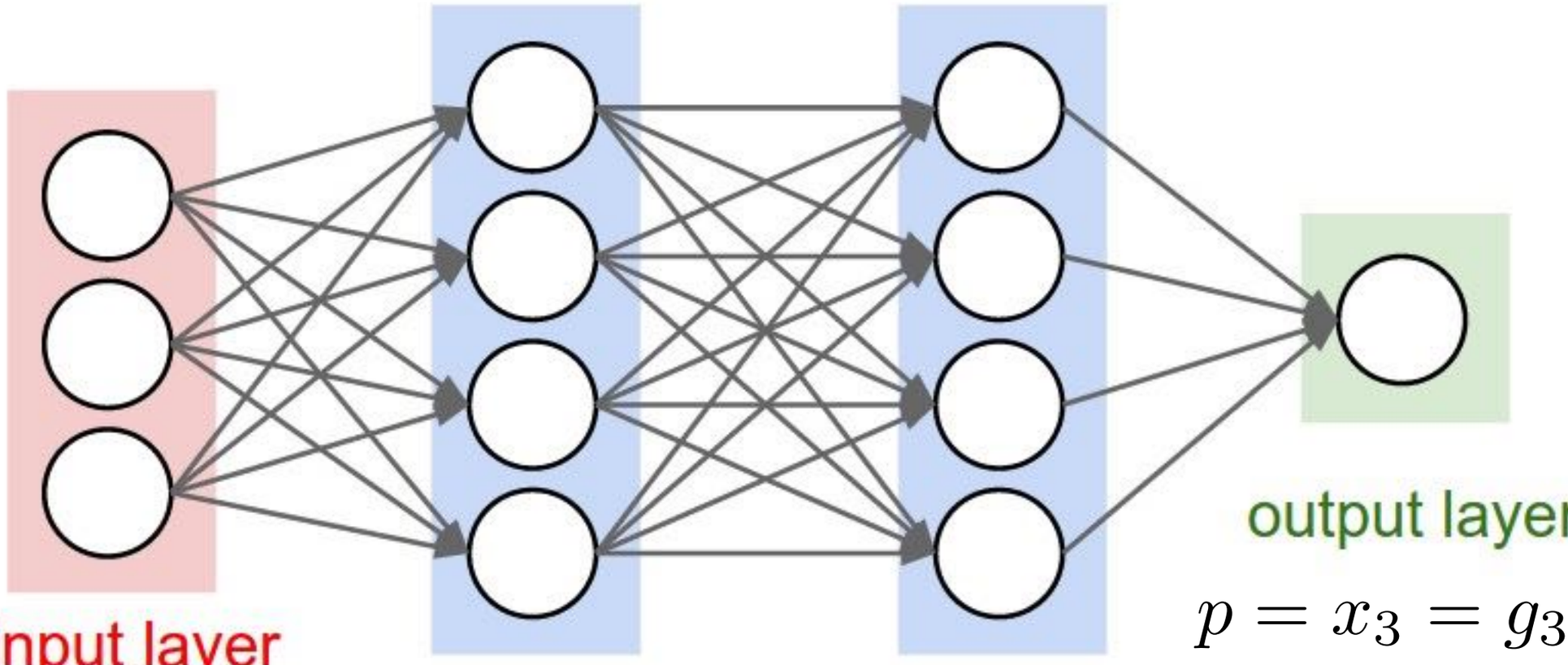



```
# Parameters
batch_size = 128
epochs = 20

# Perform fit
history = model.fit(X_train, y_train_b,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    shuffle=False,
                    validation_data=(X_test, y_test_b))


# Print results
score = model.evaluate(X_test, y_test_b, verbose=0)
print('Test loss/accuracy: %g, %g' % (score[0], score[1]))
```

```
# Specify model
model = Sequential()
model.add(Dense(512, activation="relu", input_shape=(784,)))
model.add(Dense(n_classes, activation="softmax"))

# Print model and compile it
model.summary()
model.compile(loss="categorical_crossentropy",
              optimizer=RMSprop(),
              metrics=["accuracy"])
```

Gradient descents a gogo

Feed-forward Neural networks



$$p = x_3 = g_3(W_3\vec{x}_2)$$

4x1 matrix

$\vec{x}_0$

$$\vec{x}_1 = g_1(W_1\vec{x}_0) \quad \vec{x}_2 = g_2(W_2\vec{x}_1)$$

4x3 matrix **4x4 matrix**

$$p = f(\vec{x}_0) = g_3(W_3\ g_2(W_2\ g_1(W_1\vec{x}_0)))$$

W matrices are called the « *weights* »
The functions g_n ( ) are called « *activation functions* »

How to compute the gradient efficiently?

$$\vec{x}_0 \quad \vec{x}_1 = g_1(\overbrace{W_1\vec{x}_0}^{\vec{h}_1}) \quad \dots \quad \vec{x}_n = g_n(\overbrace{W_n\vec{x}_{n-1}}^{\vec{h}_n}) \quad \dots \quad p = g_L(\overbrace{W_L\vec{x}_{L-1}}^{\vec{h}_L})$$

Feed-forward

Compute the loss $L = \frac{(y-p)^2}{2}$

Back-propagation of errors

$$e_j^1 = g_1{}'(h_j^1)\sum_i W_{ij}^2 e_i^2 \quad \dots \quad e_j^n = g_n{}'(h_j^n)\sum_i W_{ij}^{n+1} e_i^{n+1} \quad \dots \quad e^L = g_L{}'(h^L)(p-y)$$

Once this is done, gradients are given by $\frac{\partial L}{\partial W_{ab}^l} = x_b^{l-1} e_a^l$

Gradient descent vs Newton

Gradient descent

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \gamma_t \nabla f(\mathbf{W}_t)$$

Newton *(requires the Hessian)*

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \gamma [\mathbf{H} f(\mathbf{W}_t)]^{-1} \nabla f(\mathbf{W}_t)$$

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial W_1^2} & \frac{\partial^2 f}{\partial W_1 \, \partial W_2} & \cdots & \frac{\partial^2 f}{\partial W_1 \, \partial W_n} \\ \frac{\partial^2 f}{\partial W_2 \, \partial W_1} & \frac{\partial^2 f}{\partial W_2^2} & \cdots & \frac{\partial^2 f}{\partial W_2 \, \partial W_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial W_n \, \partial W_1} & \frac{\partial^2 f}{\partial W_n \, \partial W_2} & \cdots & \frac{\partial^2 f}{\partial W_n^2} \end{bmatrix}$$

Newton converges faster to local minima…

… but no one wants to compute a Hessian (or worst: inverse it)

Solutions exist:

- Quasi-newton methods such as L-BFGS approximate the inverse
- Conjugate gradient technics allows to by-pass the inversion

But most people tend to use gradient descent anyway for large problems

Gradient descent

Batch gradient descent

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \gamma_t \nabla f(\mathbf{W}_t)$$

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

Mini-batch gradient descent

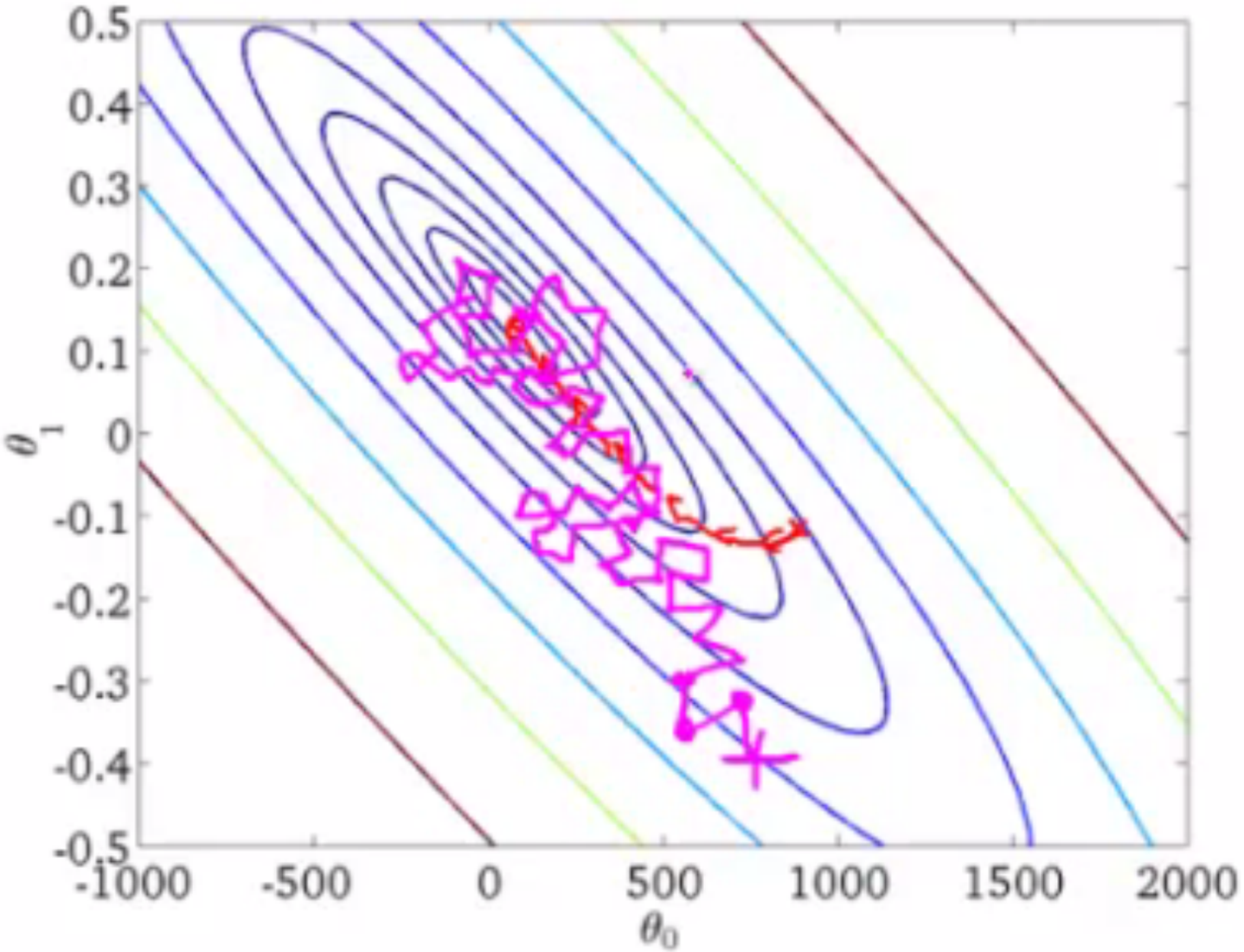
$$\mathbf{W}_{t+1/num} = \mathbf{W}_t - \gamma_t \nabla f(\mathbf{W}_t; x^{(i,i+b)}, y^{(i,i+b)})$$

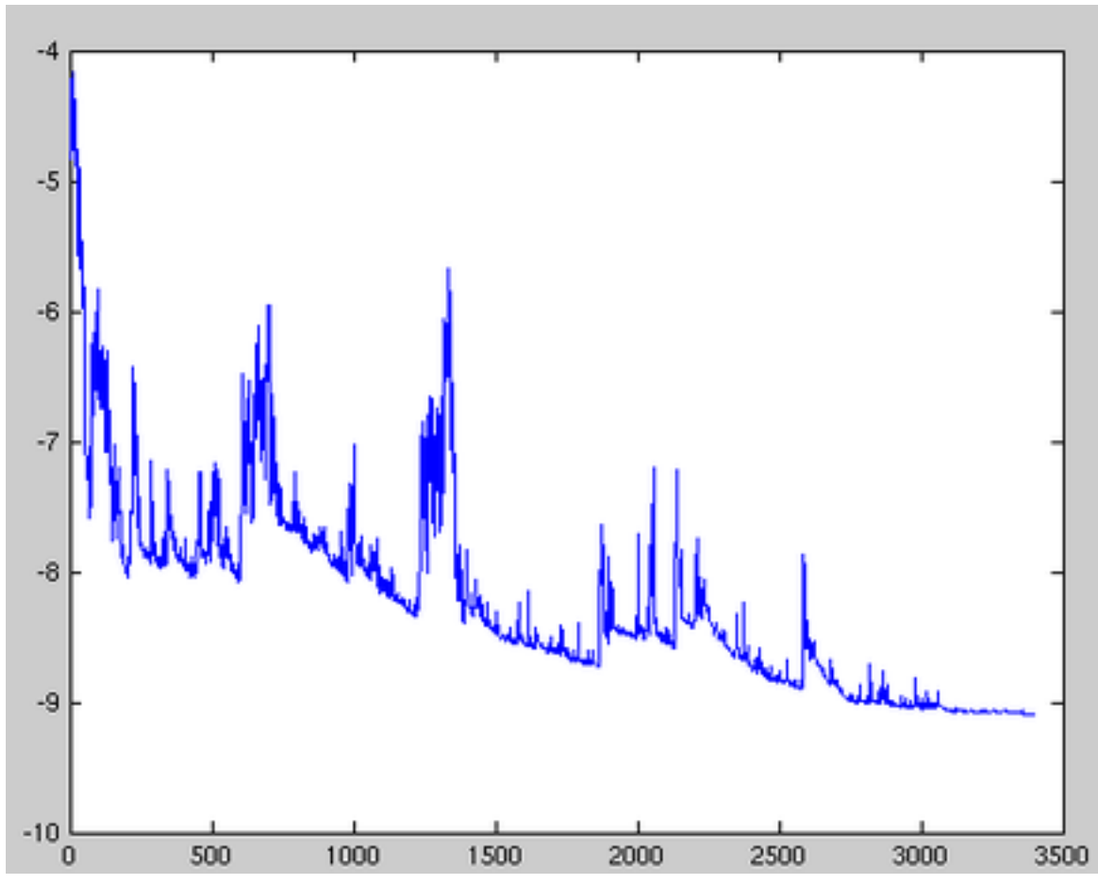
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Stochastic gradient descent

$$\mathbf{W}_{t+1/N} = \mathbf{W}_t - \gamma_t \nabla f(\mathbf{W}_t; x^{(i)}, y^{(i)})$$

```
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
```

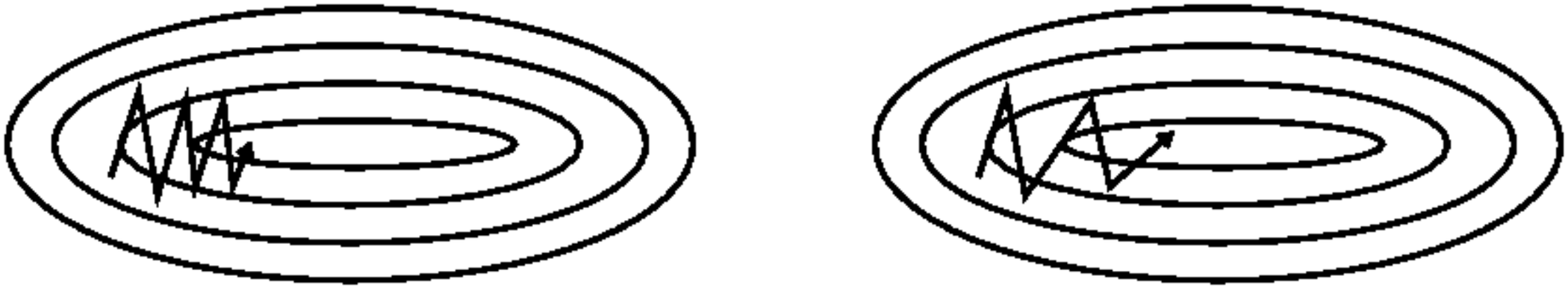




Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

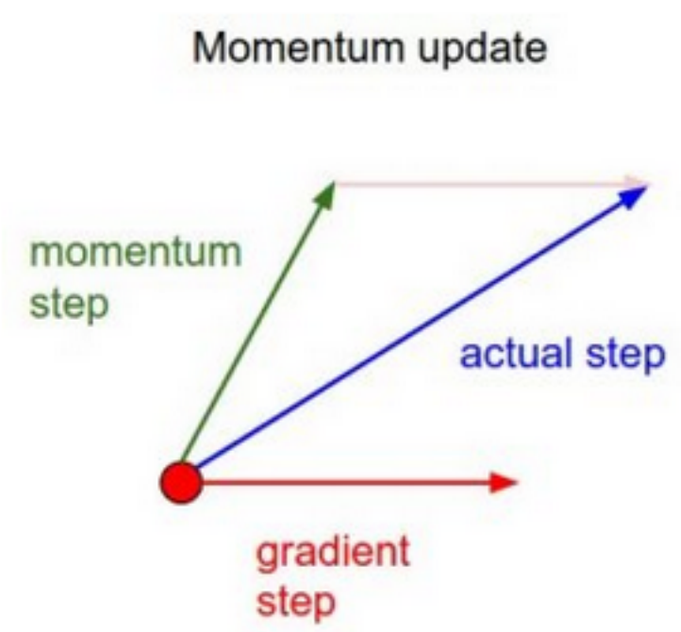
Momentum

Keep the ball rolling on the same direction



$$\mathbf{v}^{t+1} = \eta\ \mathbf{v}^{t} + \gamma \nabla f(\mathbf{W})$$

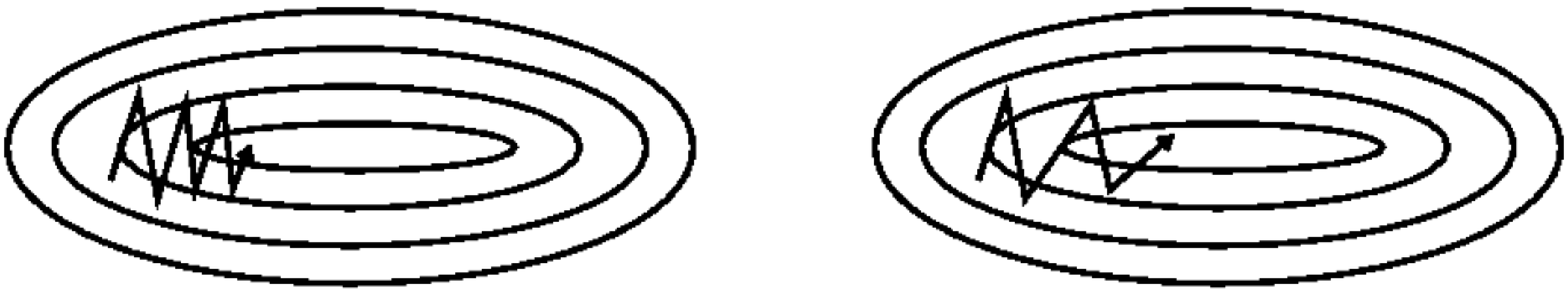
$$\mathbf{W} = \mathbf{W} - \mathbf{v}^{t+1}$$



« Effective averaging of previous directions »

Nesterov acceleration

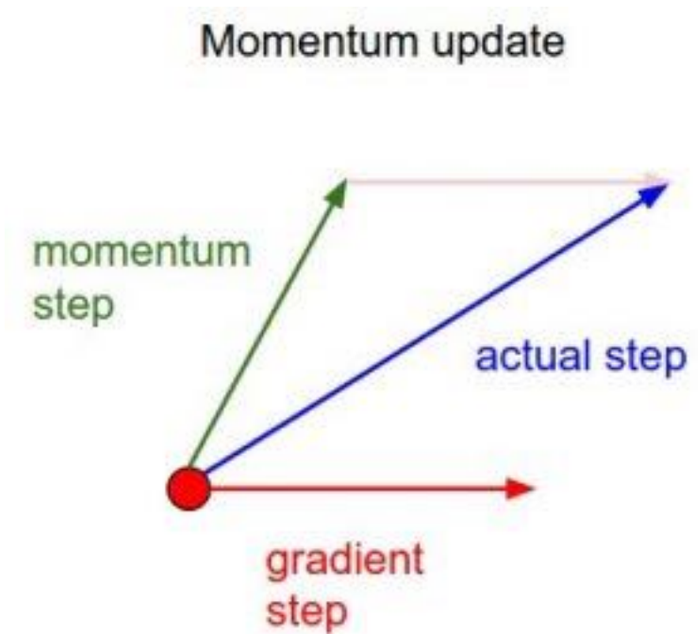
A slightly more clever ball

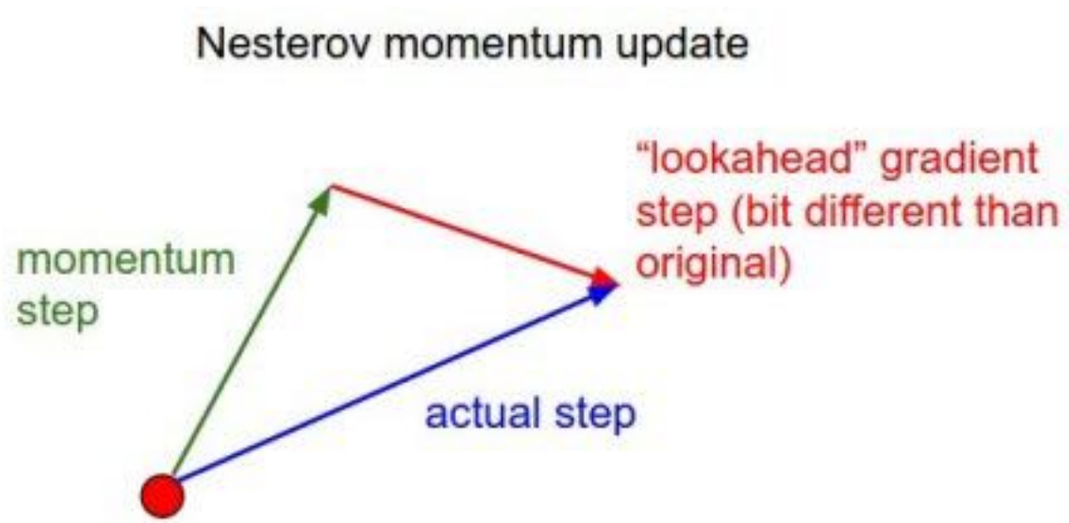


In many problems
(in fact, convex ones)
NAG is almost as fast as Newton!

$$\mathbf{v}^{t+1} = \eta\ \mathbf{v}^{t} + \gamma \nabla f(\mathbf{W} - \eta\ \mathbf{v}^{\mathbf{t}})$$

$$\mathbf{W} = \mathbf{W} - \mathbf{v}^{t+1}$$





Pytorch optimizer

class `torch.optim.SGD`(*params, lr=<object object>, momentum=0, dampening=0, weight_decay=0, nesterov=False*) [source]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

Keras optimizers

Usage of optimizers

An optimizer is one of the two arguments required for compiling a Keras model:

```
from keras import optimizers

model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('tanh'))
model.add(Activation('softmax'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

SGD

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

Stochastic gradient descent optimizer.

Includes support for momentum, learning rate decay, and Nesterov momentum.

Arguments

- **lr**: float >= 0. Learning rate.
- **momentum**: float >= 0. Parameter updates momentum.
- **decay**: float >= 0. Learning rate decay over each update.
- **nesterov**: boolean. Whether to apply Nesterov momentum.

Adaptive learning rates

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \gamma_t \nabla f(\mathbf{W}_t)$$

What about this guy ?

Adagrad:

Adagrad scales γ for each parameter according to the history of gradients (previous steps)

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \frac{\gamma}{\sqrt{G_t + \epsilon}} \nabla f(\mathbf{W}^t)$$

G is a diagonal matrix that contrains the sum of all (squared) gradient so far
When the gradient is very large, learning rate is reduced and vice-versa.

$$G_{t+1} = G_t + (\nabla f)^2$$

With adagrad, one does not need to manually adapt γ at each steps…

… but the problem is that eventually all gradients goes to zero!

Adaptive learning rates

Adagrad:

Adagrad scales γ for each parameter according to the history of gradients (previous steps)

$$\mathbf{W}^{t+1} = \mathbf{W}^{t} - \frac{\gamma}{\sqrt{G_t + \epsilon}} \nabla f(\mathbf{W}^{t})$$

G is a diagonal matrix that contrains the sum of all (squared) gradient so far
When the gradient is very large, learning rate is reduced and vice-versa.

$$G_{t+1} = G_t + (\nabla f)^2$$

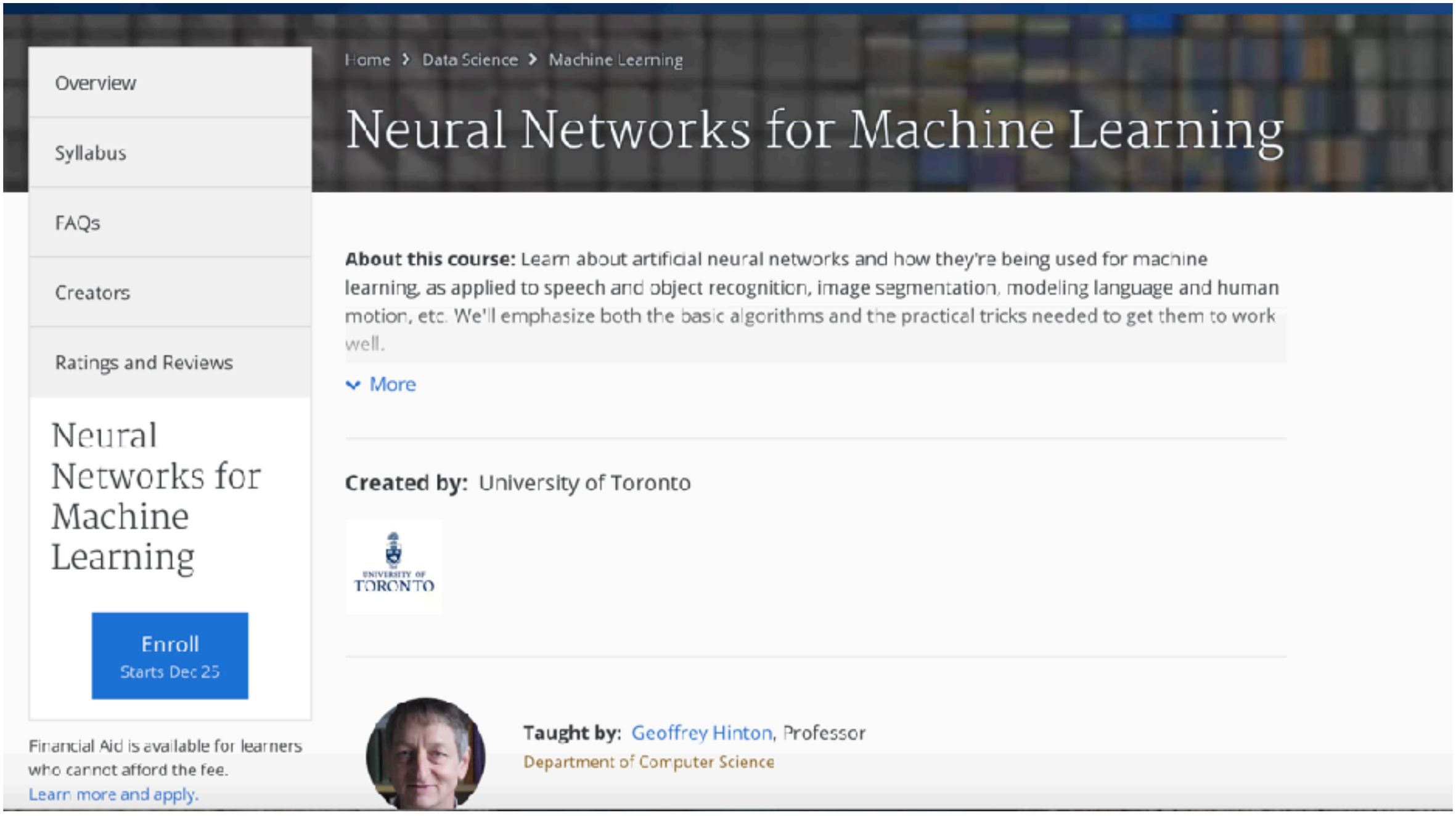
RMSprop

The only difference RMSprop has with Adagrad is that the term is calculated by exponentially decaying moving average (like we did in momentum for the gradient itself!) instead of the sum of gradients.

$$G_{t+1} = \gamma G_t + (1-\gamma)(\nabla f)^2$$

RMSprop

Proposed by G. Hinton during his coursera lecture



Adaptive learning rates

Adam: Adaptive Moment Estimation

Adam also keeps an exponentially decaying average of past gradients, similar to momentum

$$G_t = \beta_2 G_{t-1} + (1-\beta_2)(\nabla f)^2$$
$$M_t = \beta_1 M_{t-1} + (1-\beta_1)(\nabla f)$$

These are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

$$\hat{M}_t = \frac{M_t}{1-\beta_1^{\mathbf{t}}} \qquad \hat{G}_t = \frac{G_t}{1-\beta_2^{\mathbf{t}}}$$

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \frac{\gamma}{\sqrt{\hat{G}_t + \epsilon}} \hat{M}_t$$

Keras

Adam [source]

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
```

Adam optimizer.

Default parameters follow those provided in the original paper.

Arguments

- lr: float >= 0. Learning rate.
- **beta_1**: float, 0 < beta < 1. Generally close to 1.
- **beta_2**: float, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor.
- **decay**: float >= 0. Learning rate decay over each update.

References

Keras

Nadam [source]

```
keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08, schedule_decay=0.004)
```

Nesterov Adam optimizer.

Much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum.

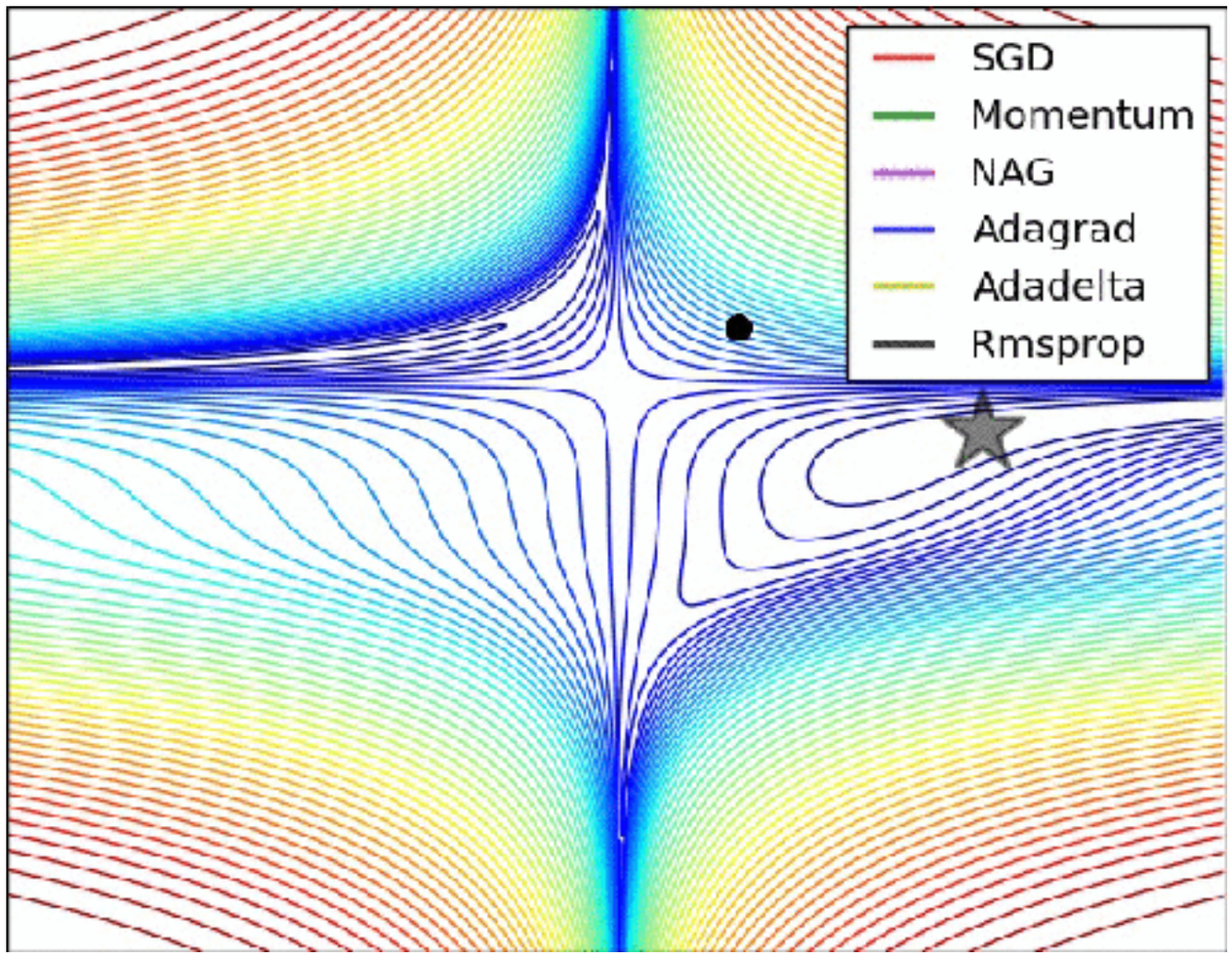
Default parameters follow those provided in the paper. It is recommended to leave the parameters of this optimizer at their default values.

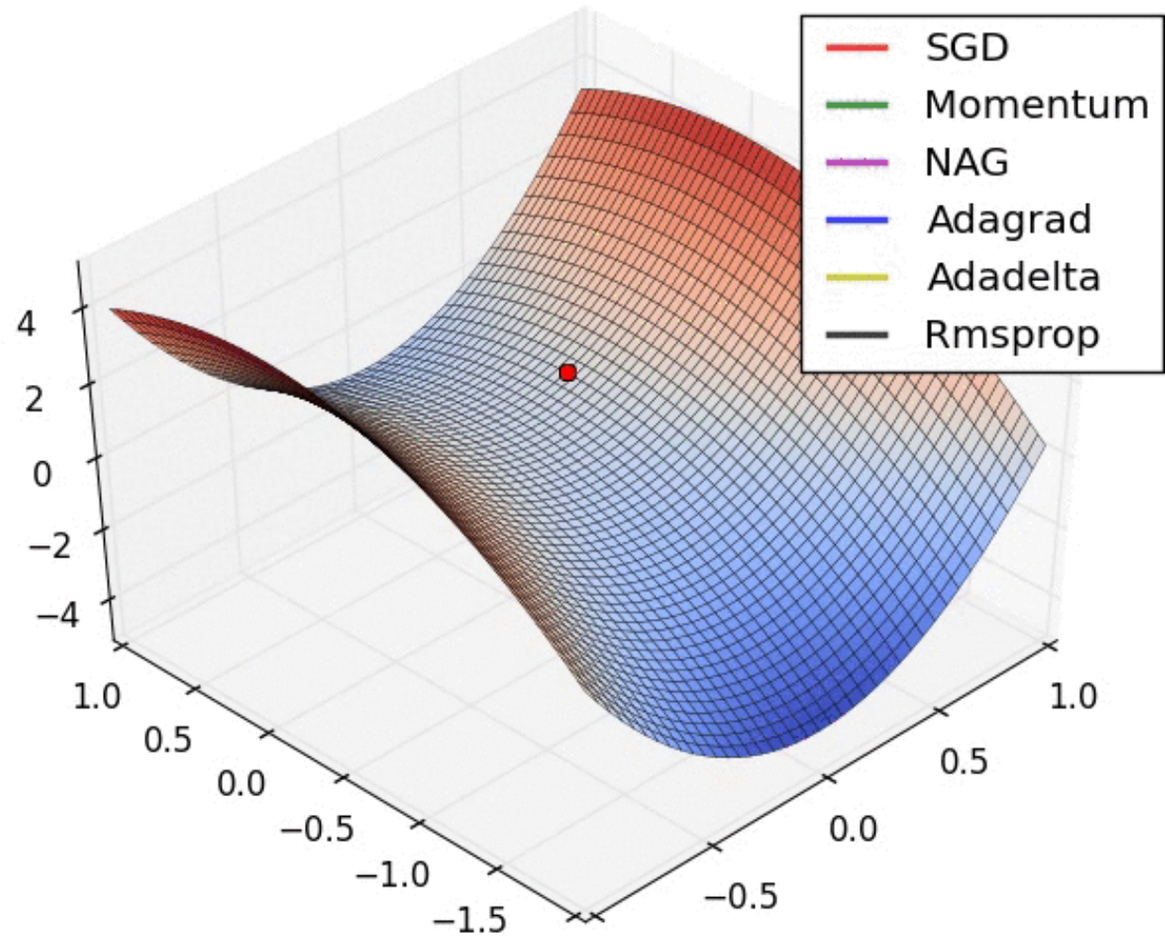
Arguments

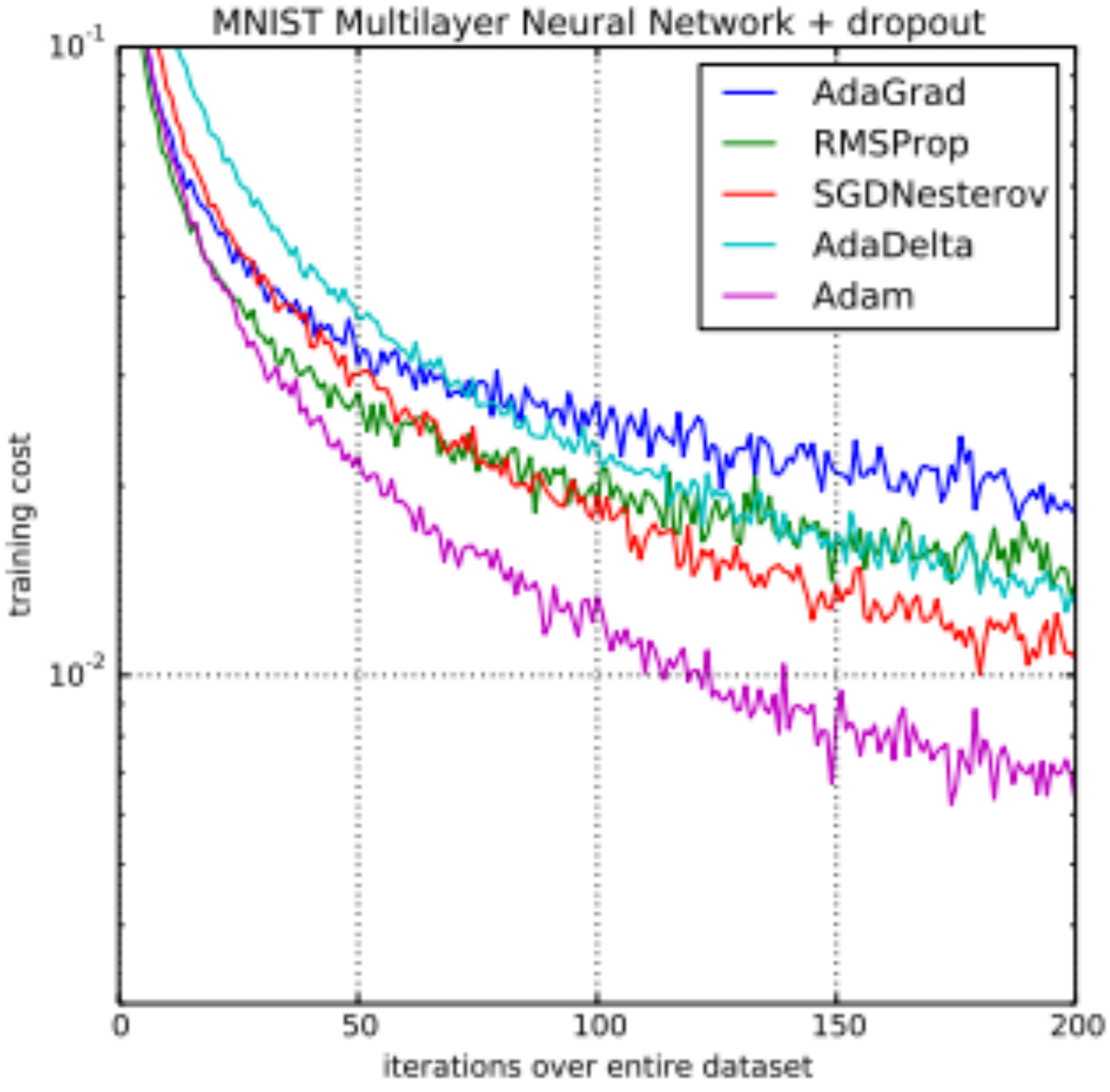
- **lr**: float >= 0. Learning rate.
- **beta_1/beta_2**: floats, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor.

References

- Nadam report
- On the importance of initialization and momentum in deep learning


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
10^{-1}
10^{-2}
training cost
0
50
100
150
200
iterations over entire dataset

Preview of next week

- A bag of tricks: dropout, batchnorm, etc…
- Special layers: embedding, convolutions, pooling
- Convolution Networks
- Sequence models, RNN, …