Deep-Learning Do-It-Yourself

Optimization

The workhorse: Empirical Risk Minimisation

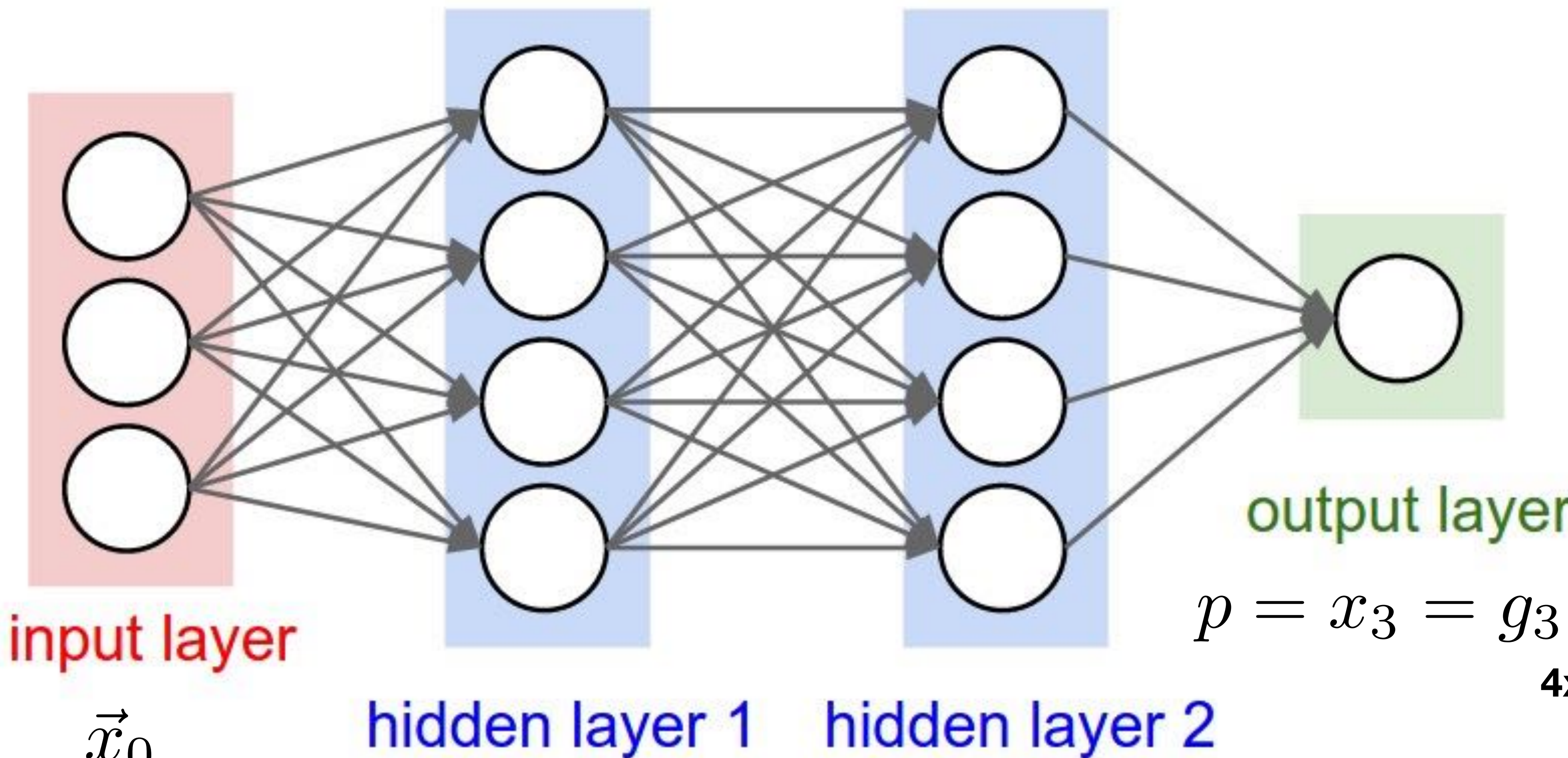
Minimize

$$\mathcal{R}_{\text{empirical}}(W) = \frac{1}{N} \sum_{i}^{\text{dataset}} \ell(W, (\vec{x}_i), y_i)$$

Rationale: it should be close to

$$\mathcal{R}_{\text{population}}(W) = \mathbb{E}\ell(W, (\vec{x}), y)$$

Feed-forward Neural networks



$\vec{x}_0$

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

4x1 matrix

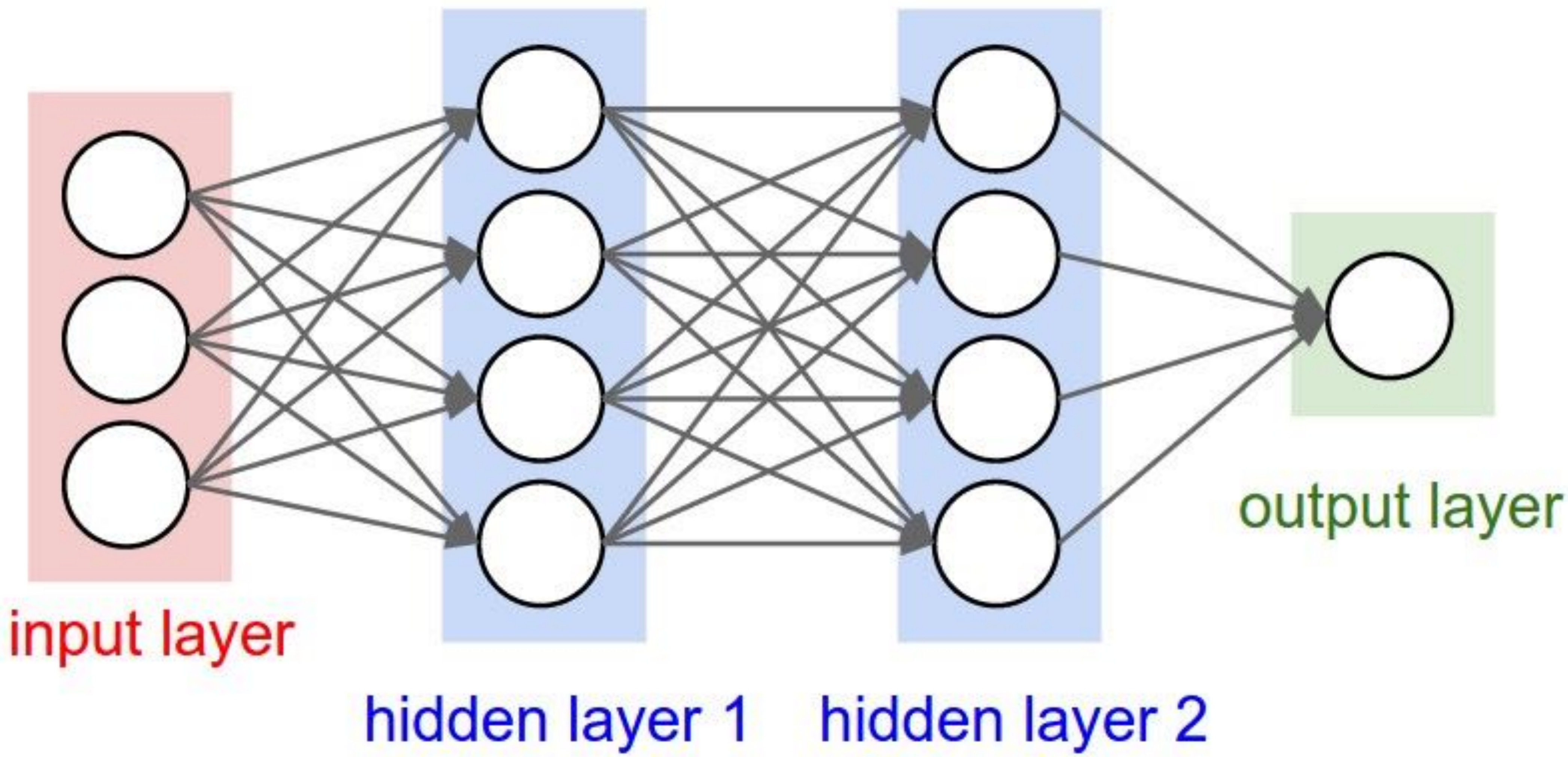
$$\vec{x}_1 = g_1(W_1 \vec{x}_0) \quad \vec{x}_2 = g_2(W_2 \vec{x}_1)$$

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 »*

Feed-forward Neural networks



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

Choose a loss function, for instance the *quadratic loss*, then one has to minimise:

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - p_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - g_3(W_3 \; g_2(W_2 \; g_1(W_1 \vec{x}_0^{\,i}))))^2$$

Gradient descent vs Newton

Gradient descent

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

Newton *(requires the Hessian)*

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

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

Newton converges faster to local minima…

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

Solution 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

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

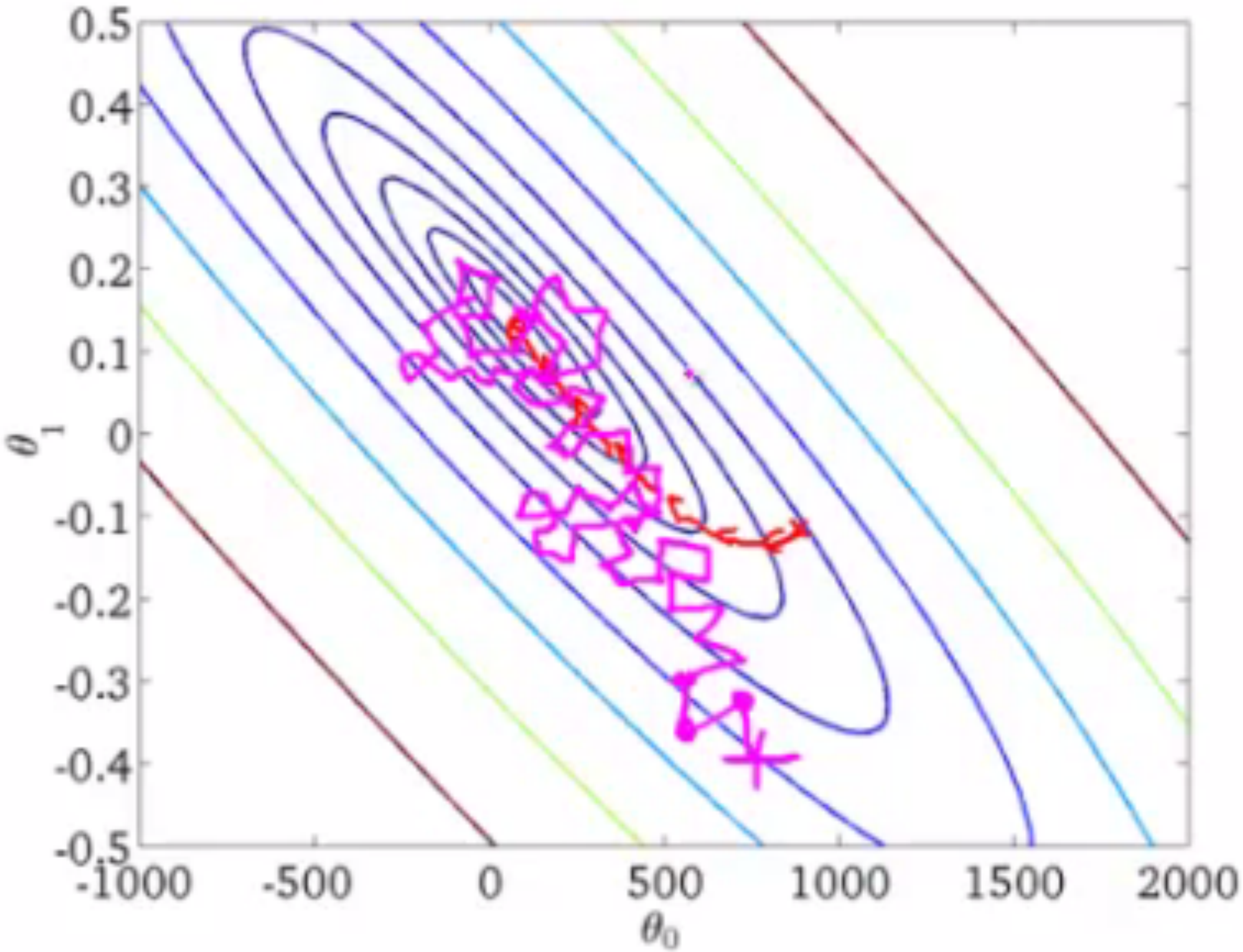
$$\mathbf{W}_{t+1/num} = \mathbf{W}_t - \gamma_t \nabla f(\mathbf{W}_t; x^{(i,i+b)}, y^{(i,i+b)})$$

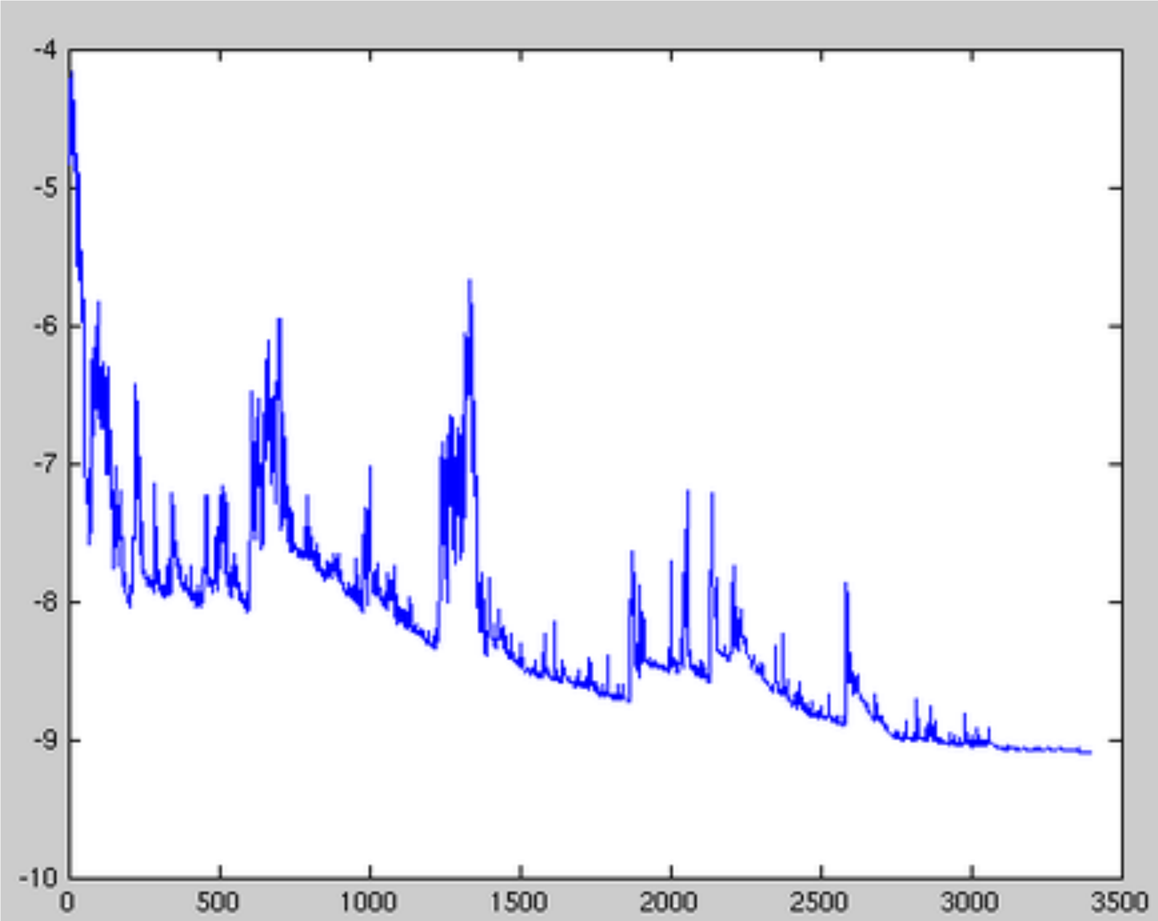
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

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
```

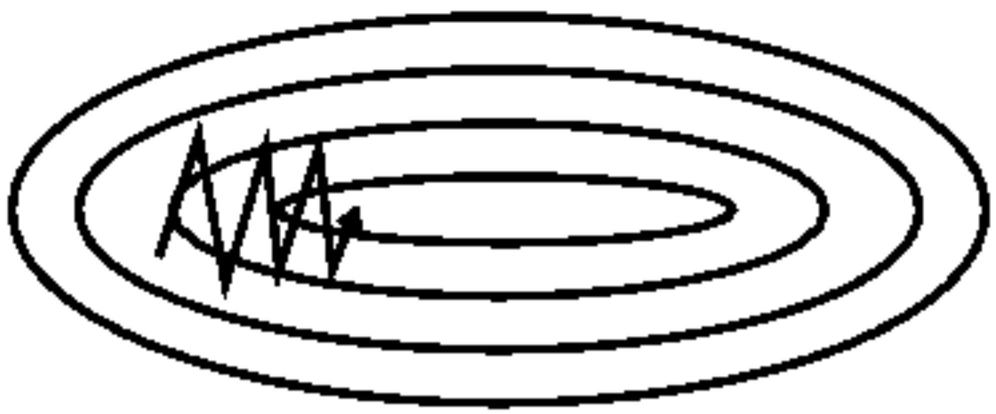


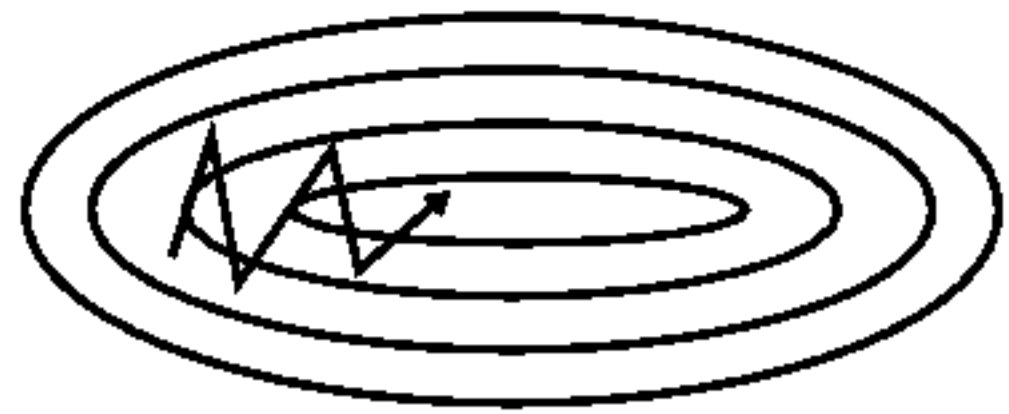


Fluctuations in the total objective function as gradient steps with respect to mini-batches are taken.

Momentum

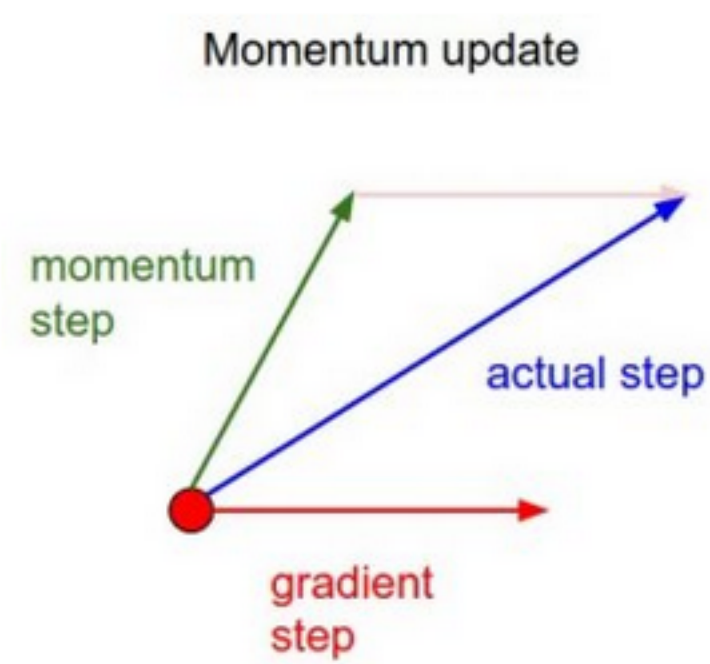
Keep the ball rolling on the same direction





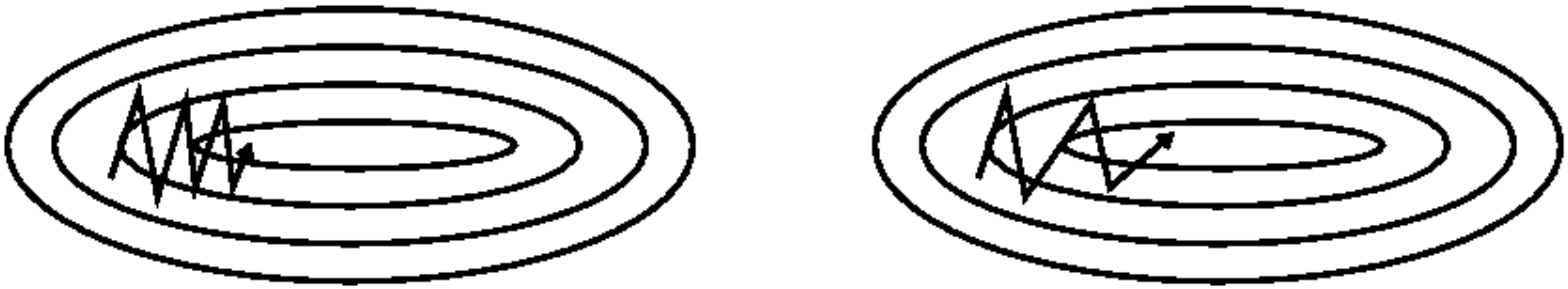
$$\mathbf{v}^{t+1} = \eta\ \mathbf{v}^{t} + \gamma \nabla f(\mathbf{W})$$

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

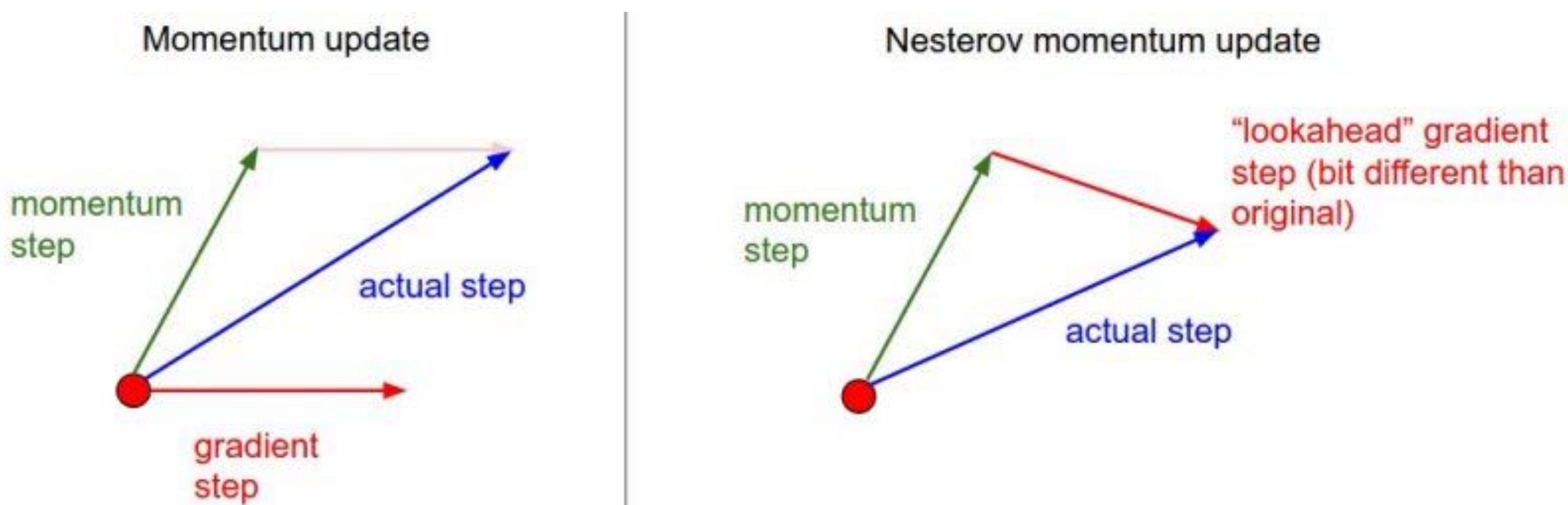


Nesterov acceleration

A slightly more clever ball



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



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()
```

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 <u>each</u> 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.

$$G_t = G_t + (\nabla f)^2$$

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.

$$G_t = G_t + (\nabla f)^2$$

RMSprop

The only difference RMSprop has with Adagrad is that the term is calculated by exponentially decaying average and not the sum of gradients.

$$G_t = \gamma G_t + (1 - \gamma)(\nabla f)^2$$

Adaptive learning rates

Adam: Adaptive Moment Estimation

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

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

$$M_t = \beta_1 M_t + (1 - \beta_1)(\nabla f)$$

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.

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

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

Pytorch optimizer

class `torch.optim.Adagrad`(*params, lr=0.01, lr_decay=0, weight_decay=0*) [source]

Implements Adagrad algorithm.

It has been proposed in Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*, *optional*) – learning rate (default: 1e-2)
- **lr_decay** (*float*, *optional*) – learning rate decay (default: 0)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)

`step`(*closure=None*) [source]

Performs a single optimization step.

Parameters: **closure** (*callable*, *optional*) – A closure that reevaluates the model and returns the loss.

Pytorch optimizer

class `torch.optim.RMSprop`(*params, lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False*) [source]

Implements RMSprop algorithm.

Proposed by G. Hinton in his course.

The centered version first appears in Generating Sequences With Recurrent Neural Networks.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-2)
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **alpha** (*float, optional*) – smoothing constant (default: 0.99)
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **centered** (*bool, optional*) – if True, compute the centered RMSProp, the gradient is normalized by an estimation of its variance
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

Pytorch optimizer

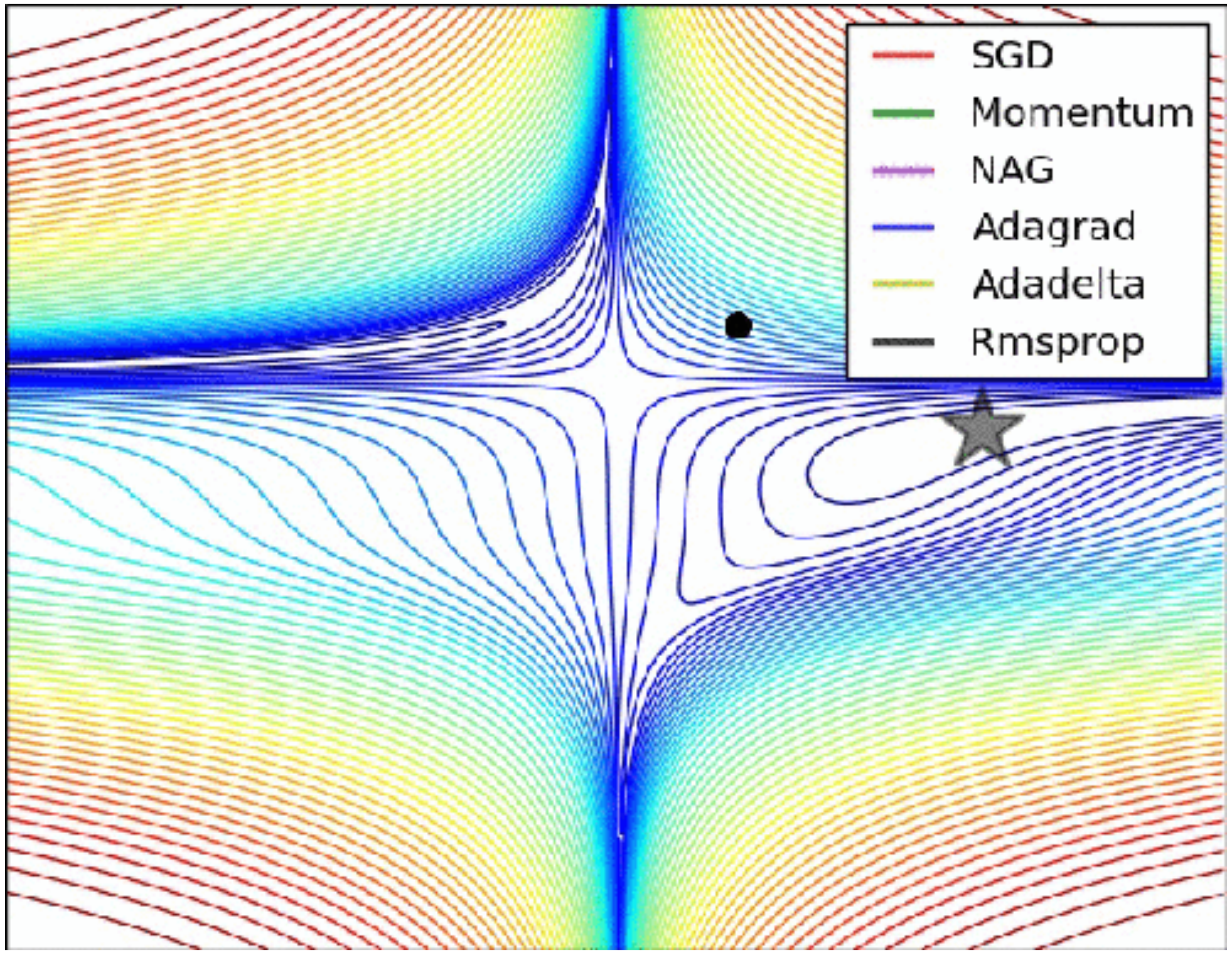
class `torch.optim.Adam`(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*)
[source]

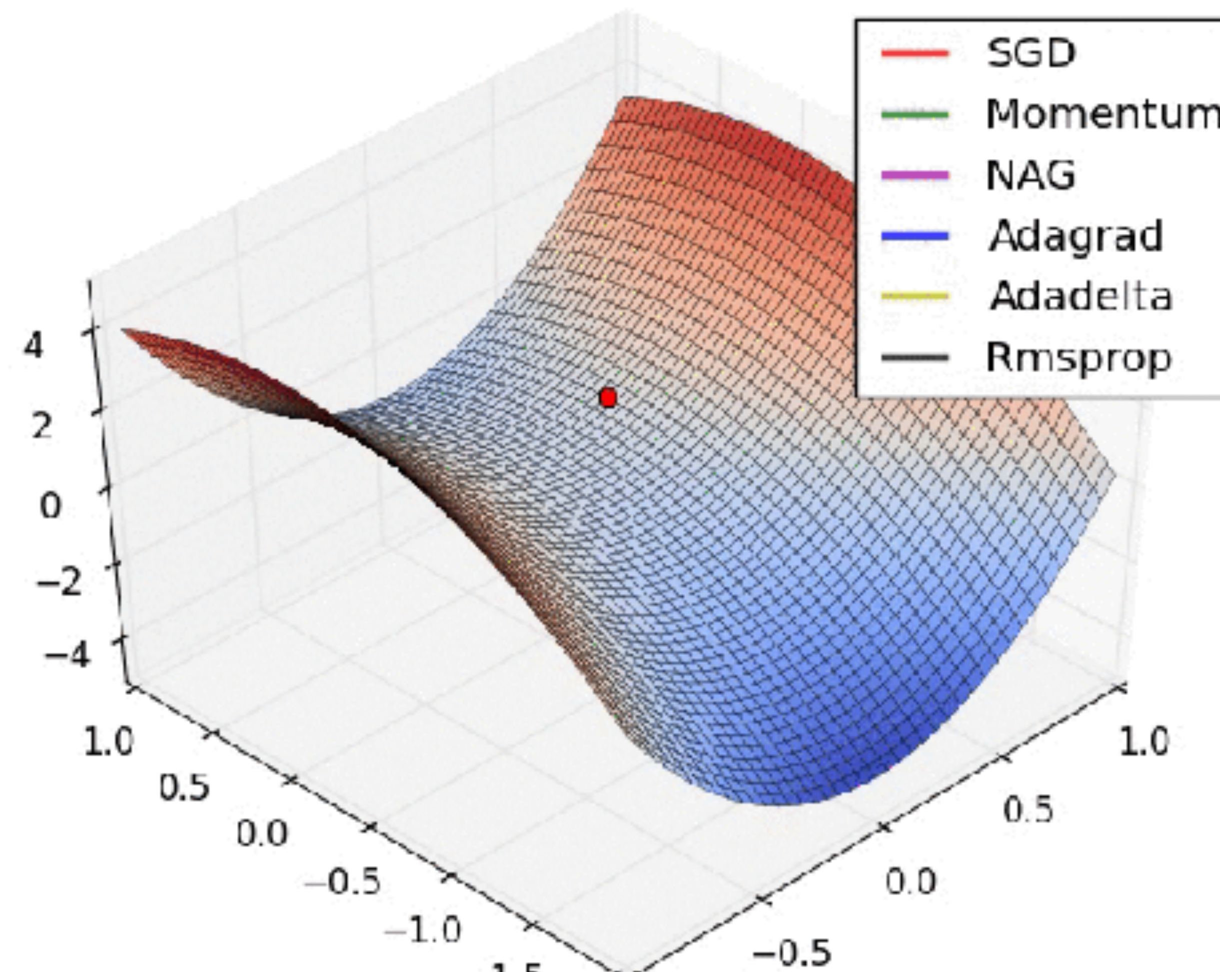
Implements Adam algorithm.

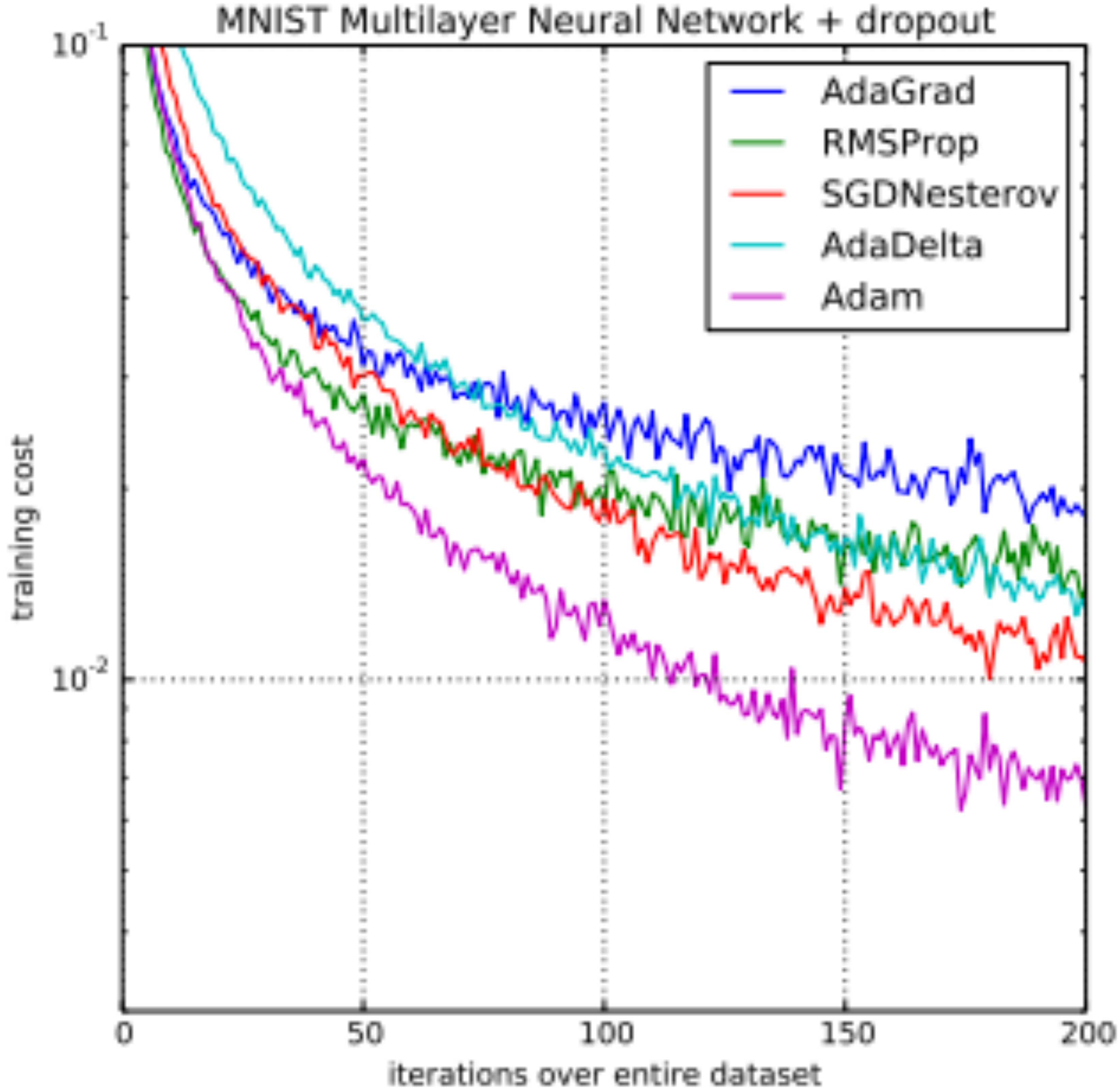
It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters:

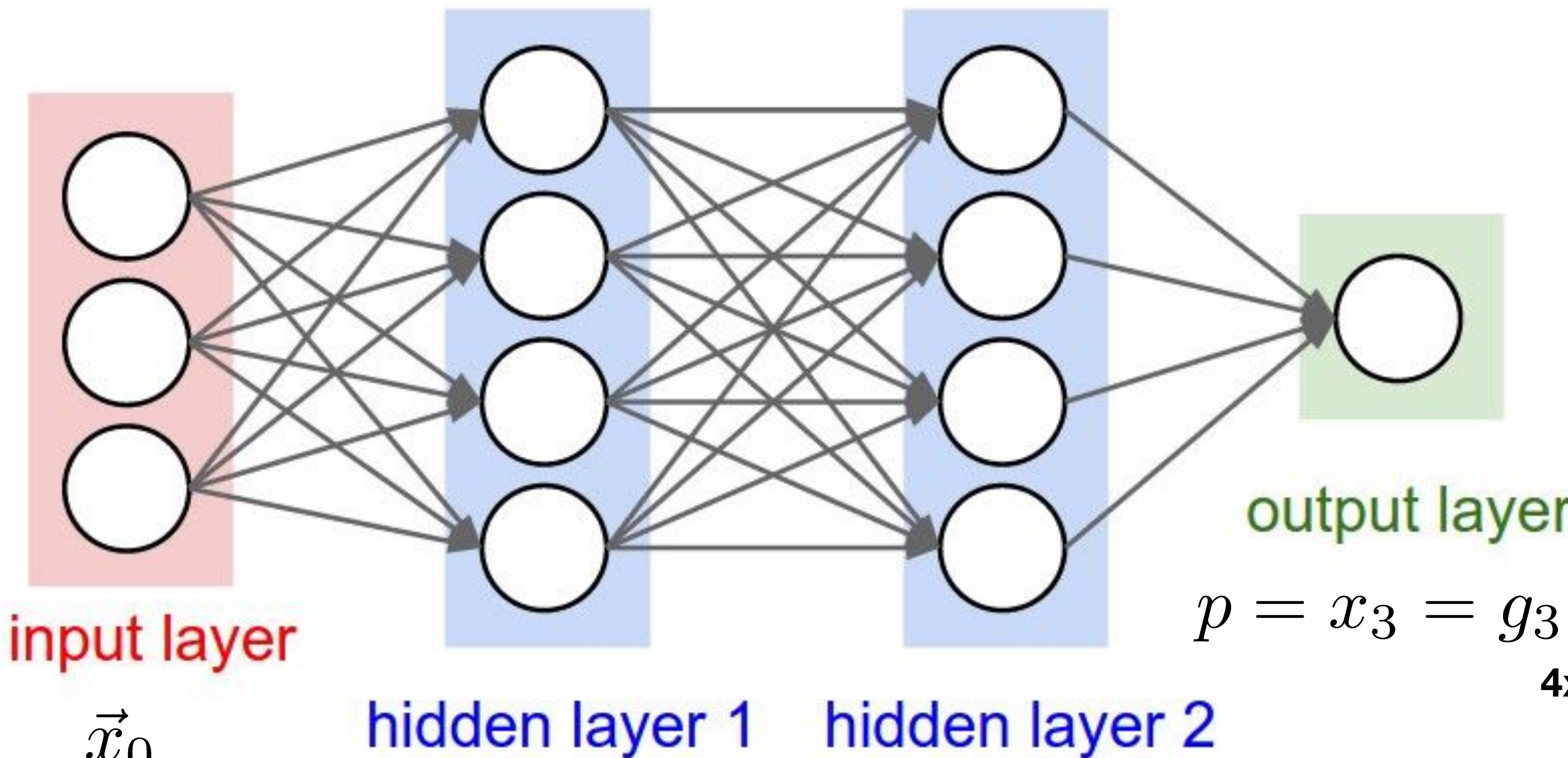
- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*, *optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float]*, *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float*, *optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)


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
training cost
iterations over entire dataset
10^{-1}
10^{-2}
0
50
100
150
200

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 $\quad$ **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 <u>« *weights* »</u>
The functions g_n ( ) are called <u>« *activation functions* »</u>

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 $L = \dfrac{(y-p)^2}{2}$

Back-propagation of errors

$$e^1_j = {g_1}'(h^1_j)\sum_i W^2_{ij} e^2_i \quad \dots \quad e^n_j = {g_n}'(h^n_j)\sum_i W^{n+1}_{ij} e^{n+1}_i \quad \dots \quad e^L = {g_L}'(h^L)(p-y)$$

Once this is done, gradients are given by $\dfrac{\partial L}{\partial W^l_{ab}} = x^{l-1}_b e^l_a$

Demonstration by the chain rule of derivatives

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

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

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

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

...

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

...

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

How to compute the gradient efficiently?

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

Feed-forward →

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

← **Back-propagation of errors**

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

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