Neural networks

Neural networks, backpropagation, optimizers

(Attila Bagoly)

Homework

- Points on the website

- After you commit your solution approx. 1 day till you can see your points on site

- If you correct it and commit new version before the deadline the latest will be evaluated

- Some general notes:

  - Numpy functions (like np.square, np.power, np.exp, np.log etc): element-wise

  - Lot of people: np.square(A): this isn't A^2!

  - Matrix multiplication: np.matmul and not *! (*-element-wise multiplication)

  - The homework notebook: hw01_numpy_solved.ipynb

  - The file must remain in assignments directory (if you move up, the grader won't see it)

  - If you have any questions: ask after class

Sync github

- New homeworks: deadline: first part 03.06, second part 03.13
- To get the new notebooks in your local repo you have to synchronize!
- First: add upstream

git remote add upstream https://github.com/**qati**/DeepLearningCourse

- Second: pull the new commits from upstream

git pull upstream master

- A text will be opened in nano (or vim), close it: CTRL+X

From last lecture: Linear regression

- Problem:
  - Given: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(M)}, y^{(M)})\}, x \in \mathbb{R}^N, y \in \mathbb{R}^K$
  - We want a model: $f: \mathbb{R}^N \to \mathbb{R}^K, f(x^{(i)})\ is\ close\ to\ y^{(i)}, \forall i$
- Linear regression: $Z = Wx + b, where\ W \in \mathbb{R}^{K \times N}, b \in \mathbb{R}^K$
- To model the dataset with Z, we have to solve (MSE loss):

$$\underset{W,b}{\operatorname{argmin}}\, L(W,b) = \underset{W,b}{\operatorname{argmin}} \left[\frac{1}{2M} \sum_{i=1}^{M} \left\| Z^{(i)} - y^{(i)} \right\|^2 \right]$$

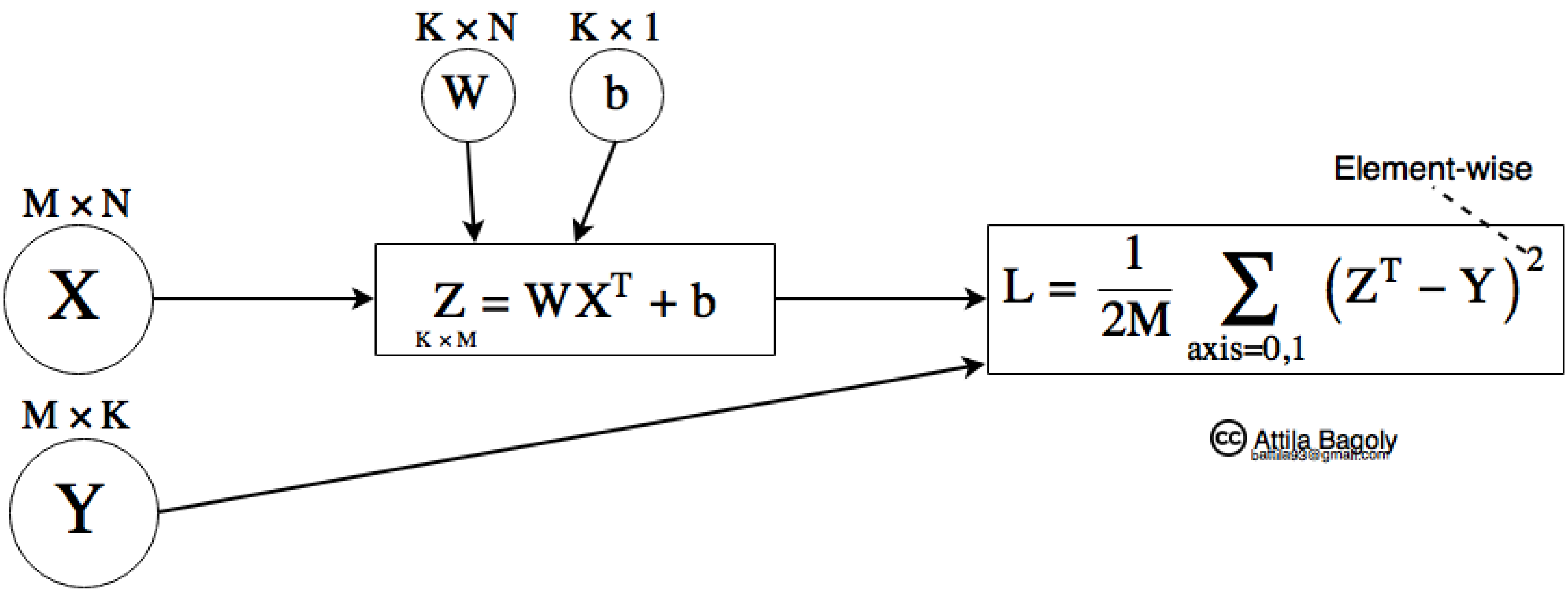
- We need the derivatives:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial W} = \frac{1}{M} (\mathrm{Z} - \mathrm{Y}) \cdot X^T$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial b} = \frac{1}{M} \sum_{i=1}^{M} \left(Z^{(i)} - Y^{(i)} \right)$$

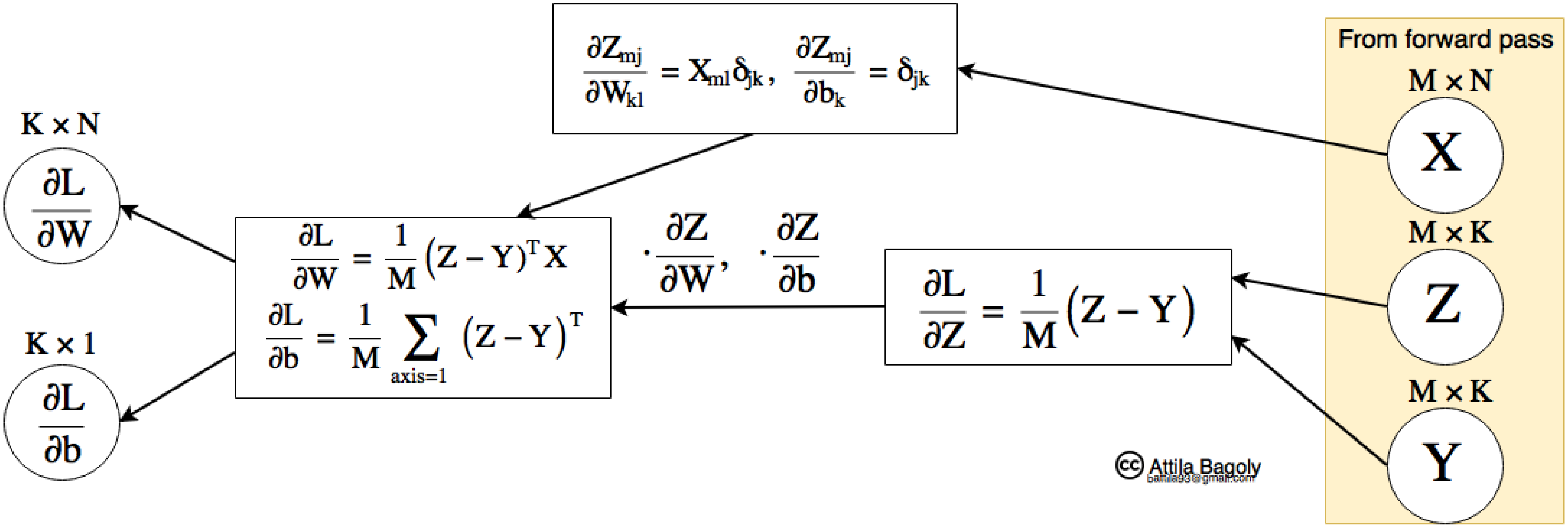
Lin. reg. computational graph: forward pass

- The computation we are doing to evaluate a linear regression model, can be represented as a graph:



- The evaluation goes from LEFT to RIGHT

- Calculating the derivatives also can be expressed as a computational graph



- The evaluation goes from RIGHT to LEFT (backward)
- Tutorial notebook: https://github.com/qati/DeepLearningCourse/blob/master/demo_notebooks/linear_regression.ipynb

K-class logistic regression

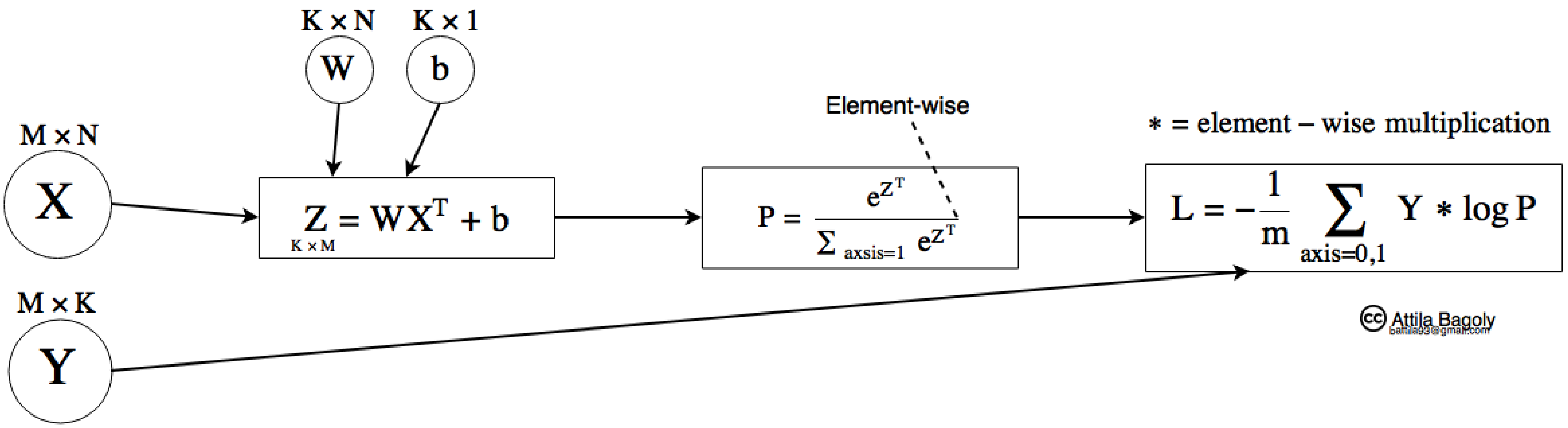
- Dataset: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}, x \in \mathbb{R}^N, y \in \{0,1, \dots, K\}$
- Model (linear): $\mathrm{z} = Wx + b \in \mathbb{R}^K$, $x \in \mathbb{R}^N, W \in \mathbb{R}^{K \times N}$, $b \in \mathbb{R}^K$
- Probability: softmax of z:

$$P(y|x) = \frac{1}{\sum_{j=0}^{K} e^{z_j}} \begin{bmatrix} e^{z_0} \\ \vdots \\ e^{z_K} \end{bmatrix}$$

- Interpretation: $P(y|x)_k = probability\ of\ x\ being\ in\ class\ number\ k$
- To model the dataset with $P(y|x)$, we have to solve (cross-entropy between target and predicted $P(y|x)$ distributions):

$$\underset{W,b}{\operatorname{argmin}} \left[-\frac{1}{2m} \sum_{i=1}^{m} \sum_{k=0}^{K} {y^{(i)}}_k \log P\left(y|x^{(i)}\right)_k \right]$$

- This computation also can be expressed as a graph:



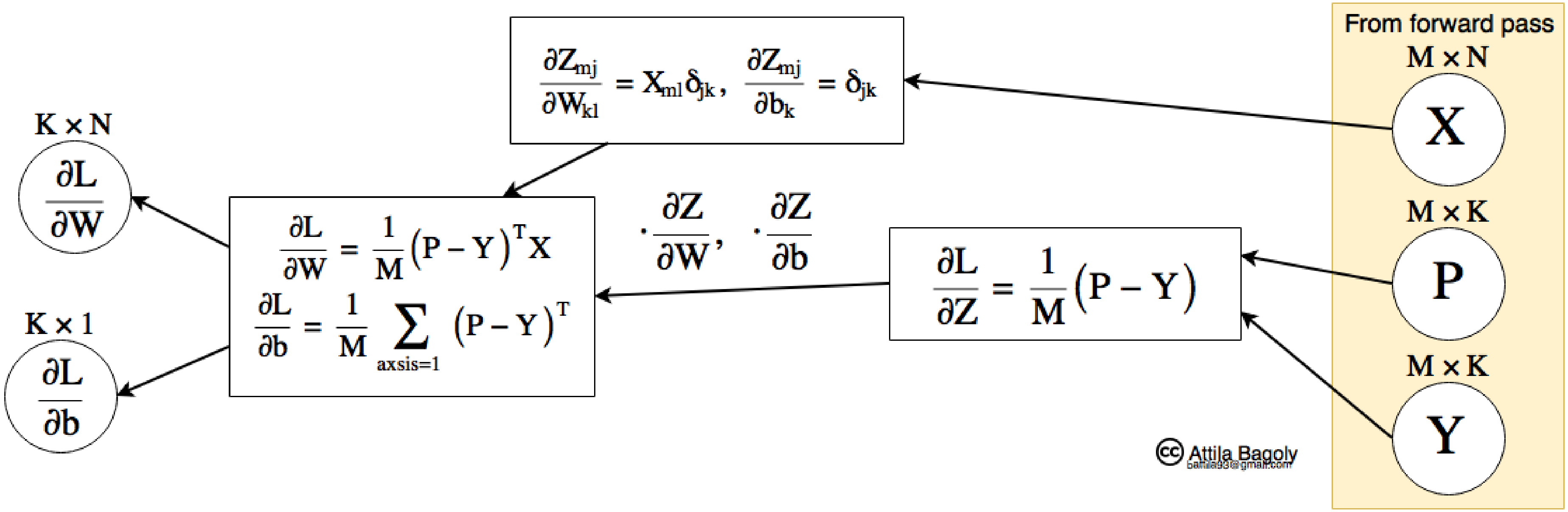
- The evaluation goes from LEFT to RIGHT (forward)

Computational graph of log. reg.: backward step

- Same way as with linear regression, to calculate the derivates we use the chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial P} \cdot \frac{\partial P}{\partial Z} \cdot \frac{\partial Z}{\partial W} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial P} \cdot \frac{\partial P}{\partial Z} \cdot \frac{\partial Z}{\partial b}$$

- The softmax and cross-entropy: $\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial P} \cdot \frac{\partial P}{\partial Z} = \frac{1}{M}(P - Y)$
- The computational graph:



- The computation goes from RIGHT to LEFT (backward)

L-layer neural network

$x \in \mathbb{R}^N, y \in \mathbb{R}^K$, neural network: $\mathbb{R}^N \to \mathbb{R}^K$

$$z^{[1]} = W^{[1]}x + b^{[1]}, \qquad W: n^{[1]} \times N, \qquad b: n^{[1]} \times 1$$
$$a^{[1]} = g(z^{[1]})$$

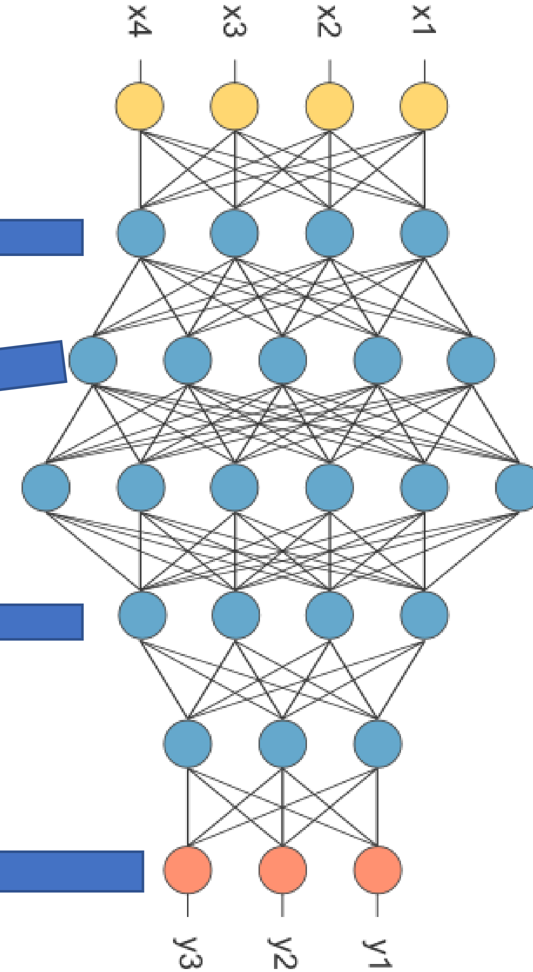
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, \qquad W: n^{[2]} \times n^{[1]}, \qquad b: n^{[2]} \times 1$$
$$a^{[2]} = g(z^{[2]})$$

$\vdots$

$$z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}, \qquad W: n^{[i]} \times n^{[i-1]}, \qquad b: n^{[i]} \times 1$$
$$a^{[i]} = g(z^{[i]})$$

$\vdots$

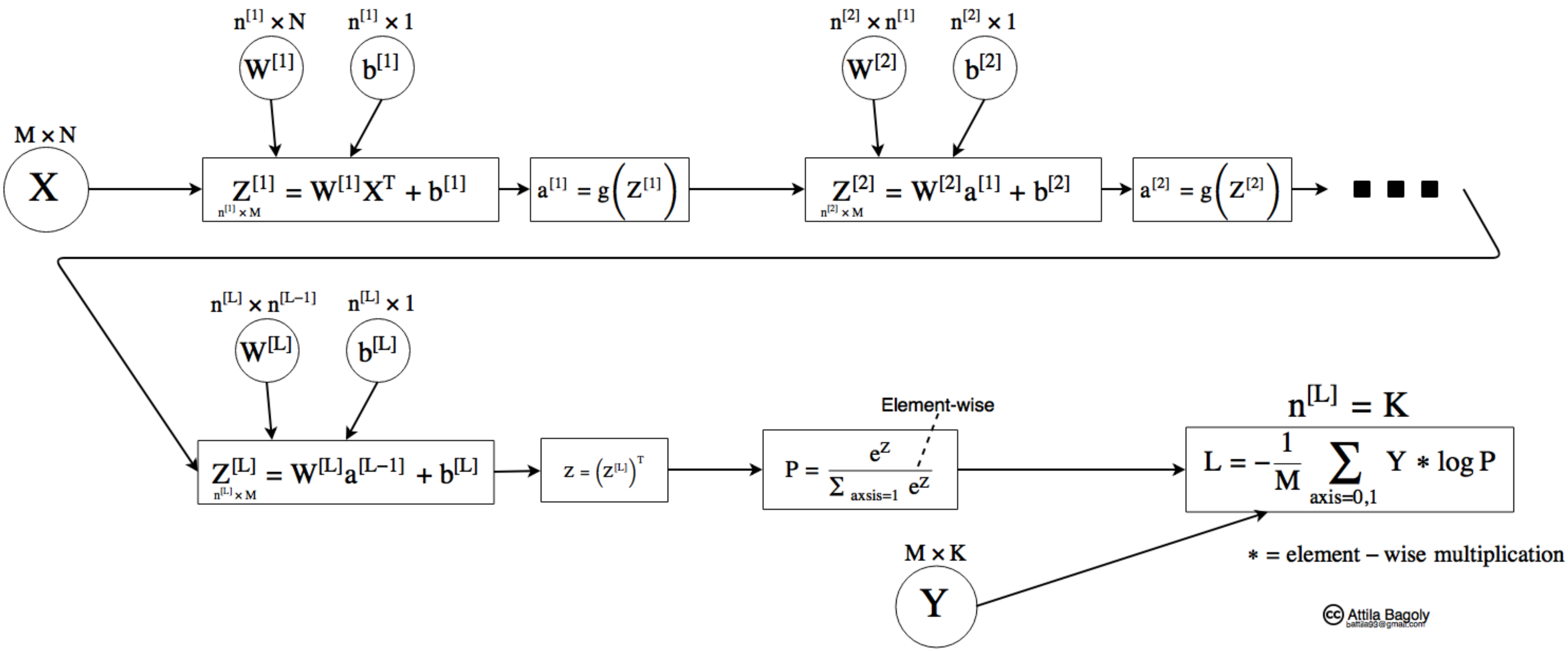
$$z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}, \qquad W: n^{[L]} \times n^{[L-1]}, \qquad b: n^{[L]} \times 1$$
$$y = a^{[L]} = softmax(z^{[L]})$$



Credit: OpenNN

L-layer neural network computational graph

- The whole neural network can be represented as a computational graph:

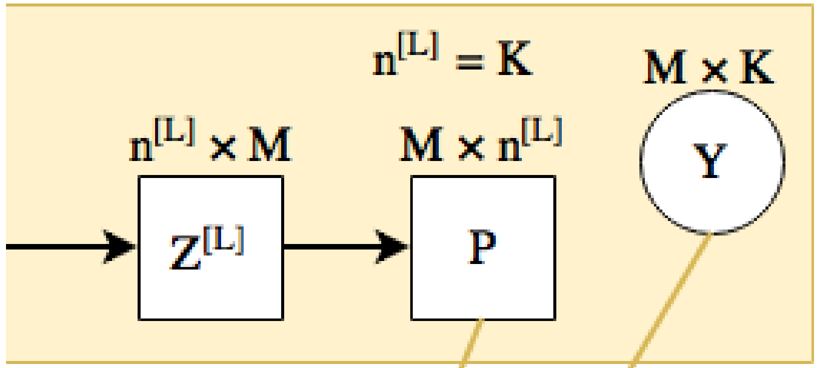


- The calculation goes from LEFT to RIGHT

L-layer neural network: backpropagation

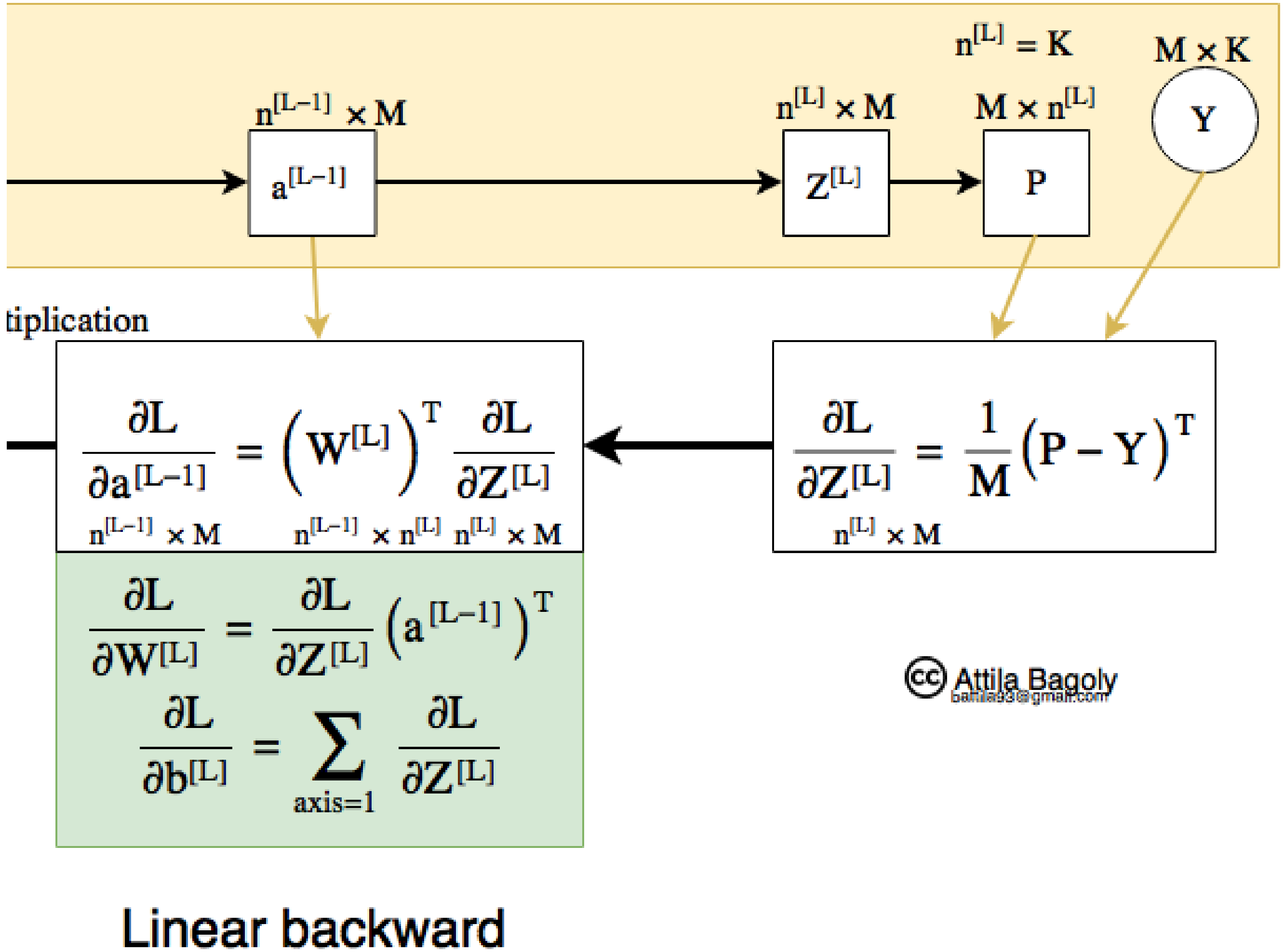
- We start calculating the derivative at the top of the network
- The last unit of the network is a K-class logistic regression unit
- To calculate the derivative we have to evaluate the following node:

This comes from the forward step

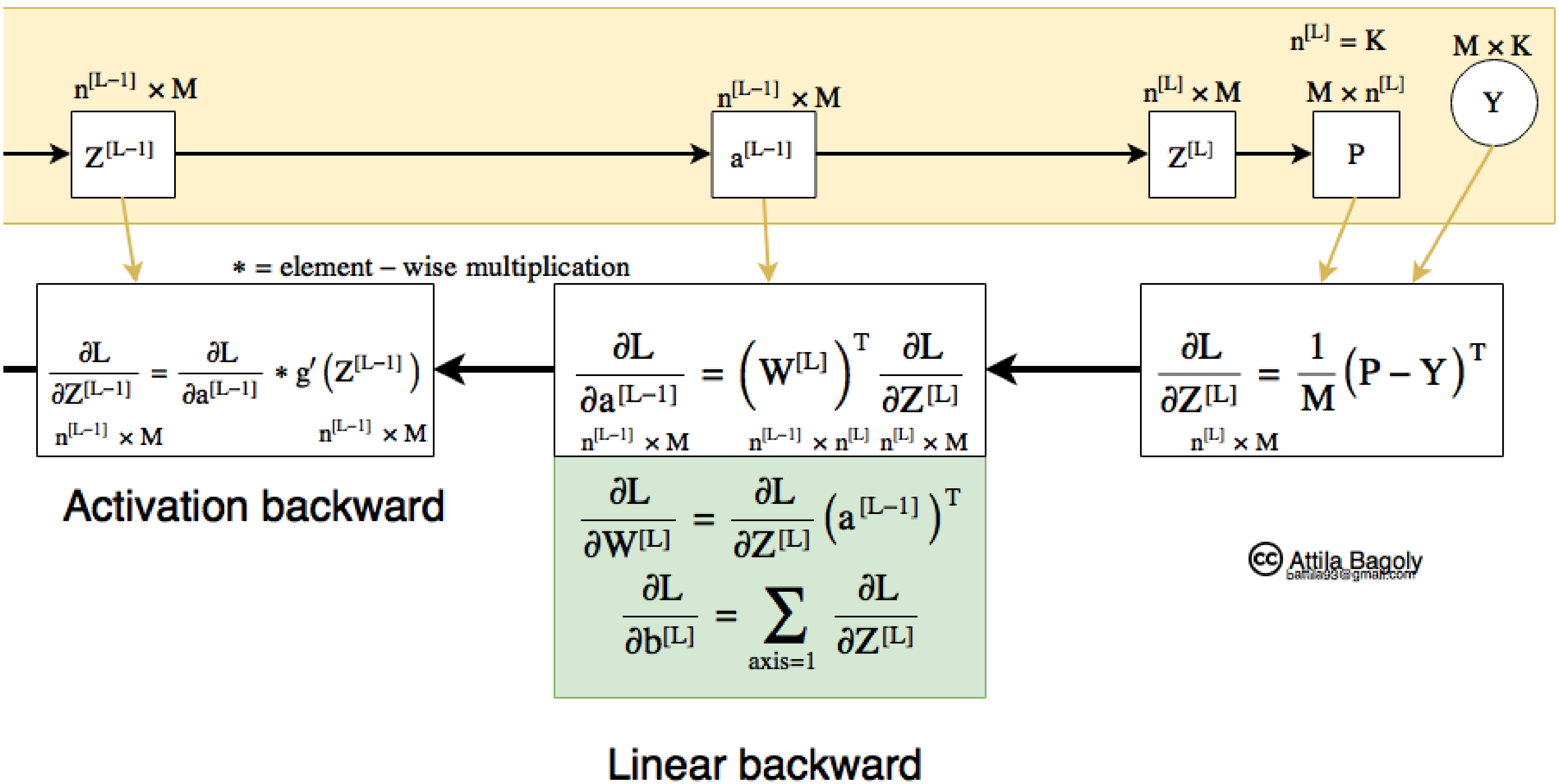


$$\underbrace{\frac{\partial L}{\partial Z^{[L]}}}_{n^{[L]} \times M} = \frac{1}{M}(P - Y)^T$$

- After we calculated the derivative at the top of the network we do a backward pass
- This step is also simply the chain rule:

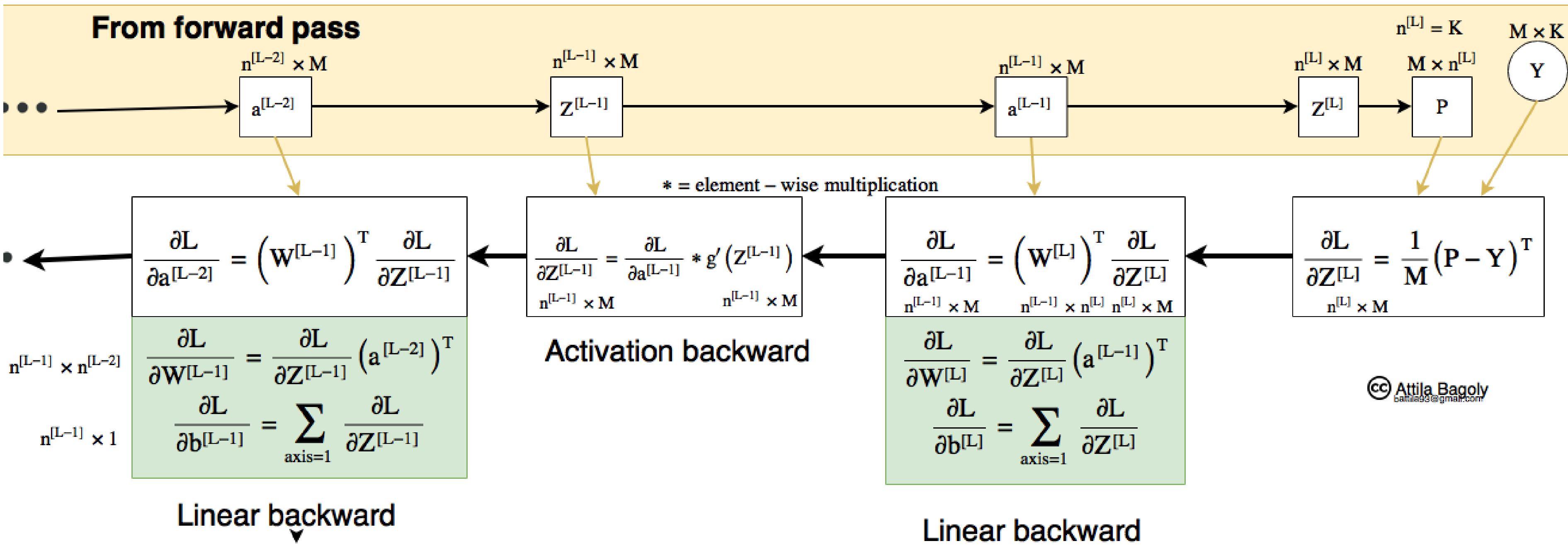


- After the backward pass on a linear unit we have to go trough a non-linear unit
- We use chain rule again

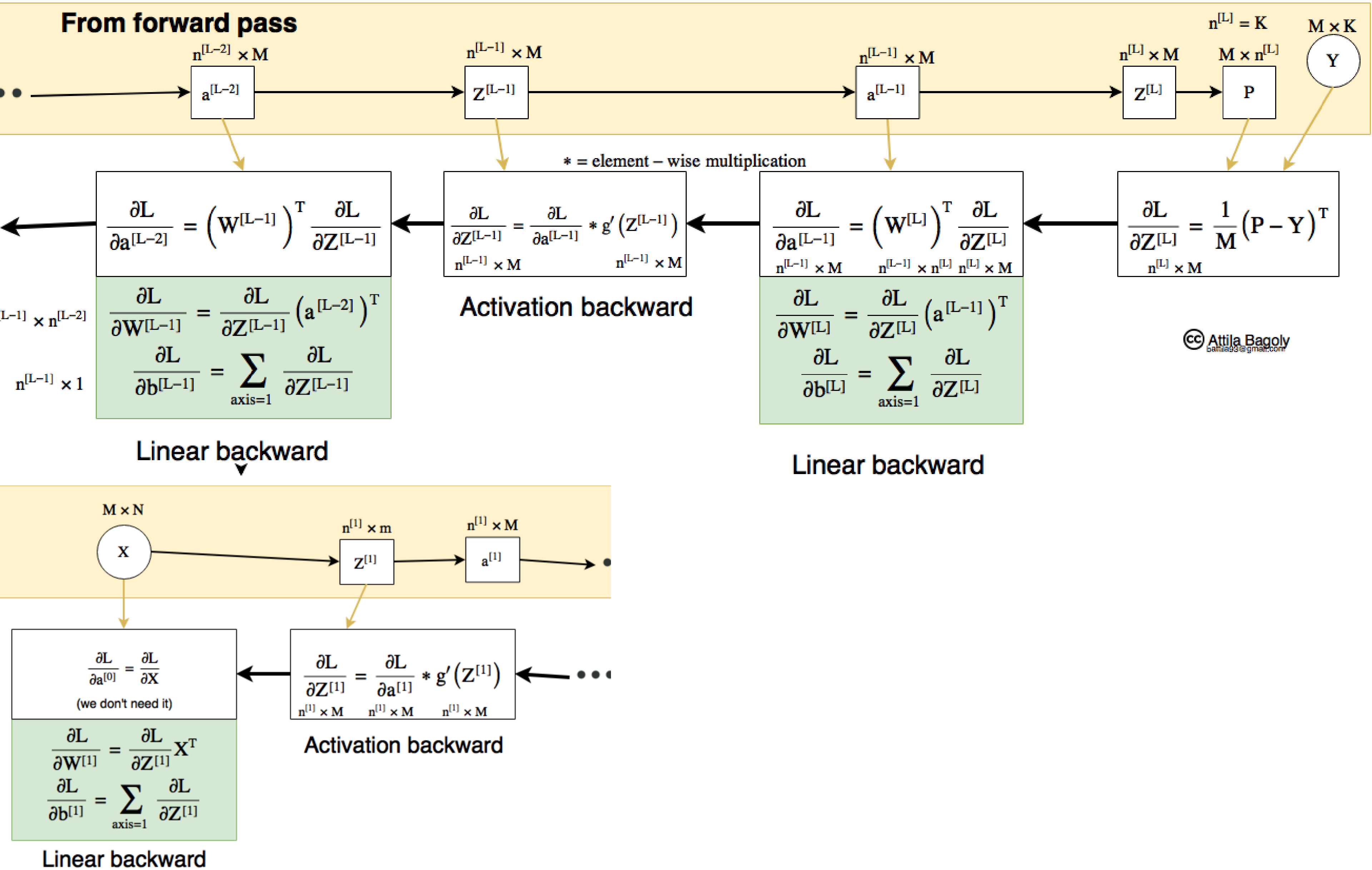


L-layer neural network: backpropagation

- And again we do a backward step on a linear unit:



L-layer neural network: backpropagation



- Problem:

$$\underset{W,b}{\operatorname{argmin}}\, L(W, b)$$

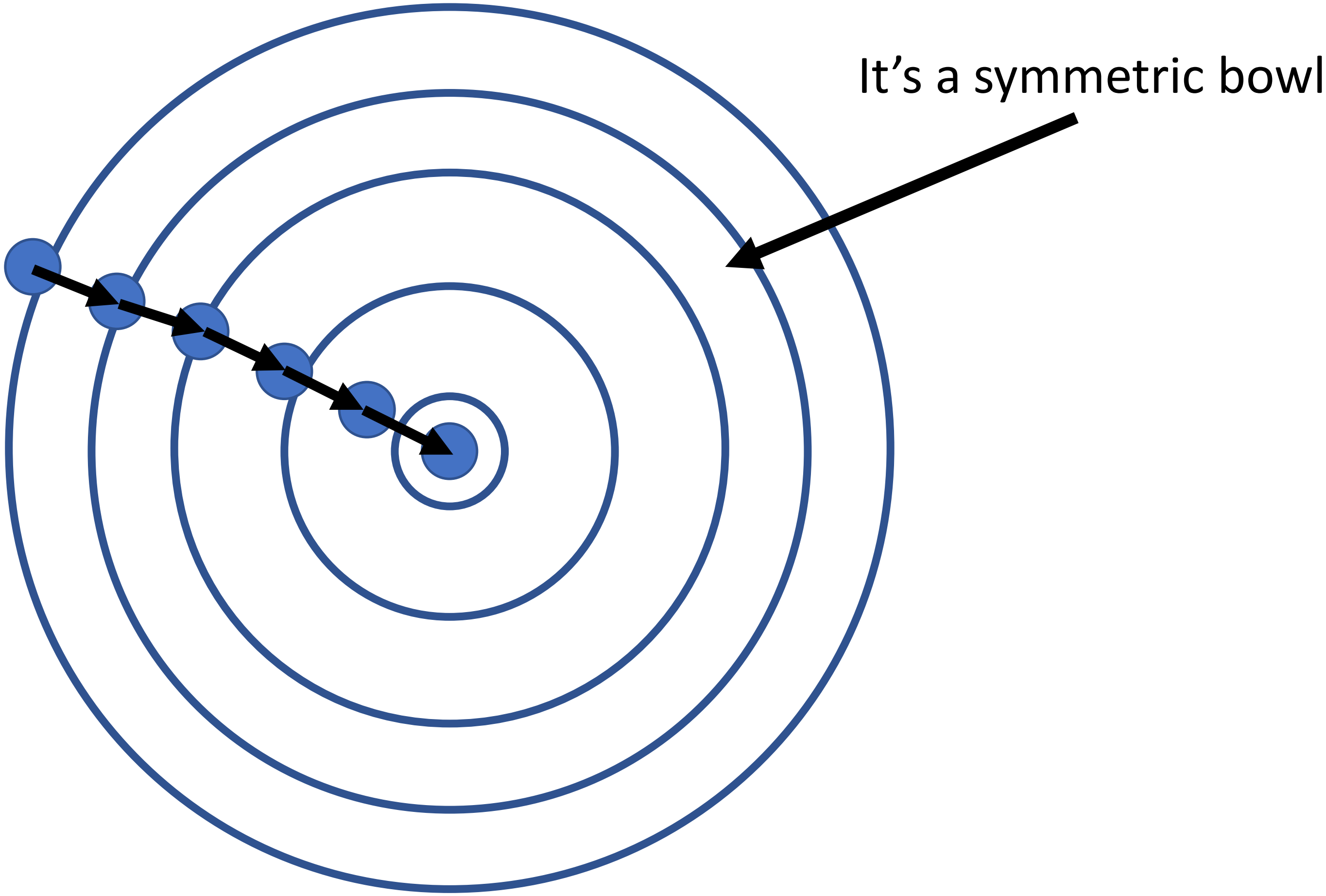
- One solution:

$$\frac{\partial L}{\partial W} = 0, \qquad \frac{\partial L}{\partial b} = 0$$

- Problem: too complicated for neural networks
- Solution: gradient descent

$$\text{repeat} \quad \left\{ \begin{aligned} W &= W - \alpha \frac{\partial L}{\partial W} \\ b &= b - \alpha \frac{\partial L}{\partial b} \end{aligned} \right\}$$

Gradient descent



Mini-batch gradient descent

- The loss function:

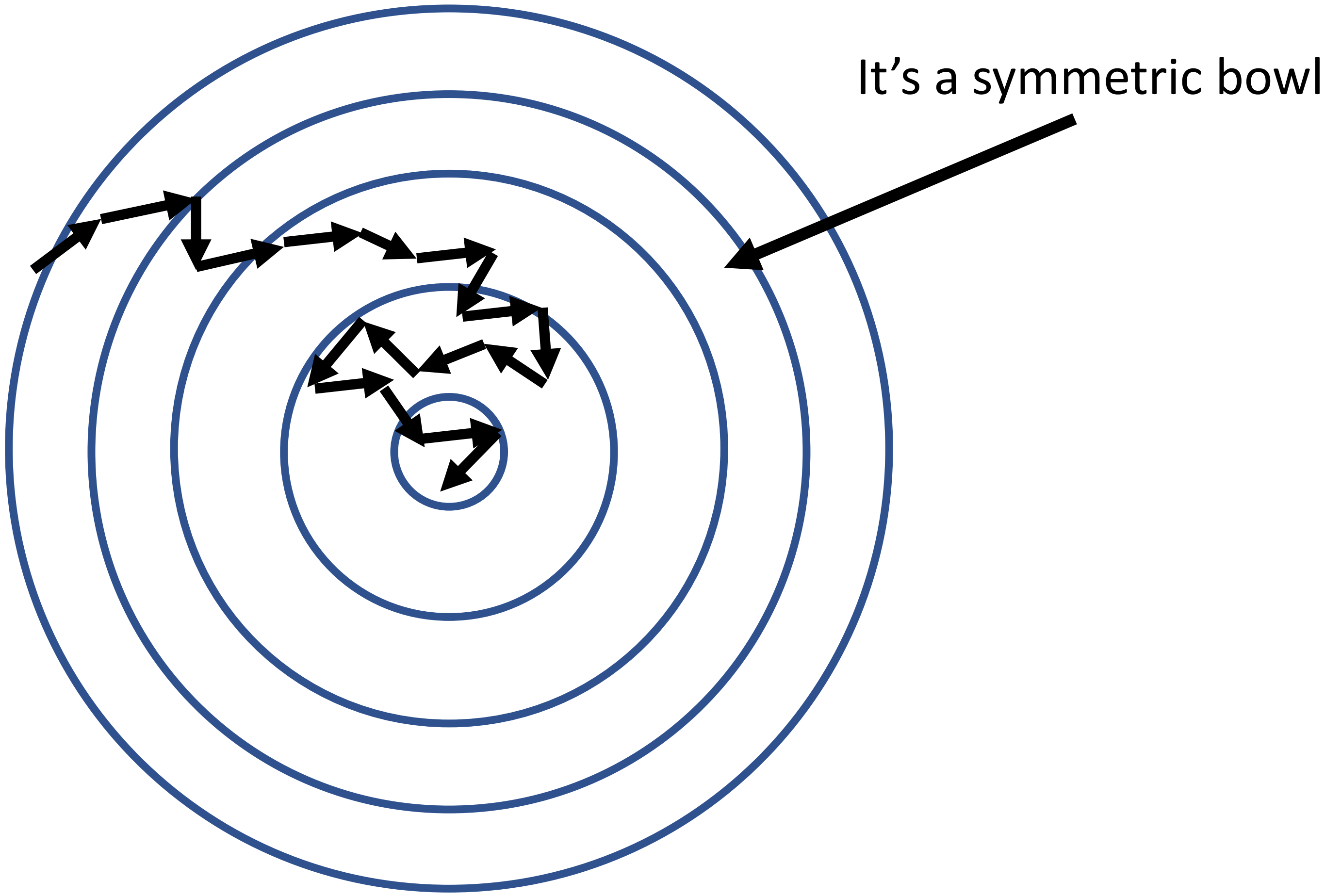
$$L \sim \frac{1}{M_{\text{full}}} \sum_{i=1}^{M_{\text{full}}} \ldots$$

- Large datasets: very slow to compute L for all data (e.g. 10M images)
- We need a lot of steps to find minima
- Instead of the whole dataset let's use a mini-batch:
  - Shuffle the dataset (it's bad, if only one category is present in the batch)
  - Select for every iteration $M < M_{\text{full}}$ small batch (every iteration different batch)
  - And compute the loss on this mini-batch:

$$L \sim \frac{1}{M} \sum_{i=1}^{M} \ldots$$

  - 1 epoch: we went trough the data one time ($M_{\text{full}} = 100$, $M = 10$ then 1 epoch is 10 iteration)

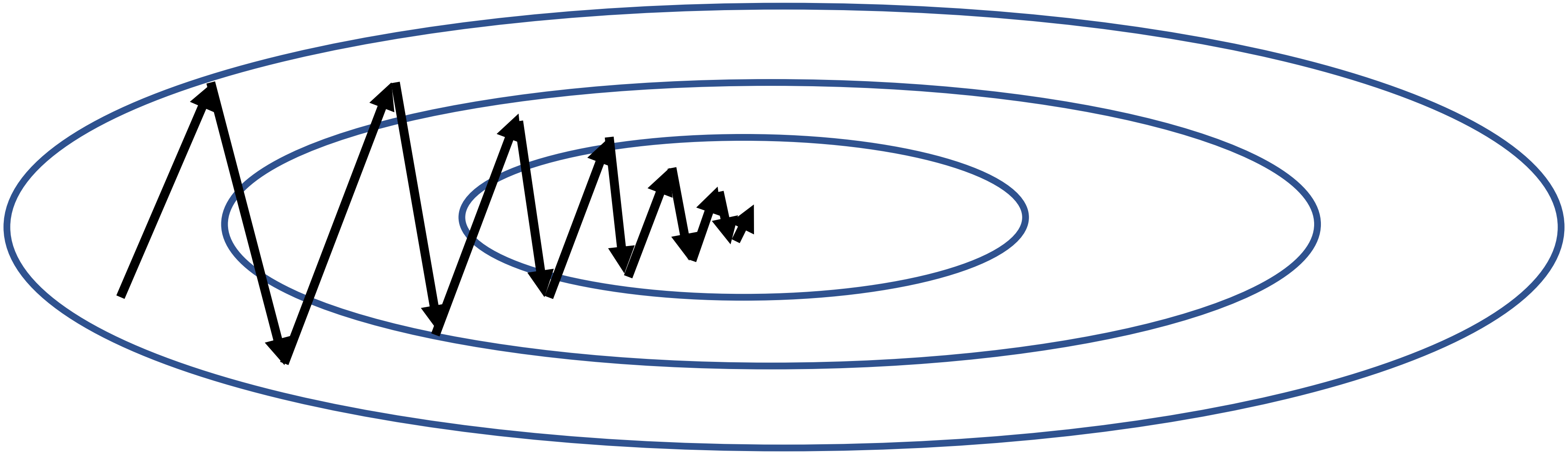
Mini-batch gradient descent



Exponentially weighted averages

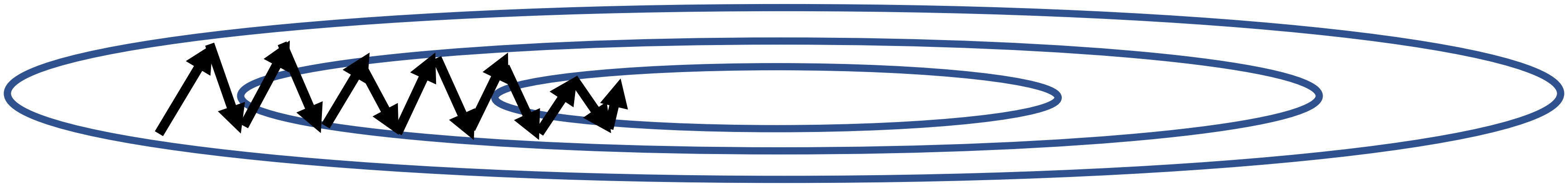
- Exponentially weighted averages notebook
- The same notebook in GitHub

Gradient descent problems: assymmetric potential.



Momentum gradient descent

- Oscillation in y direction: slows down the training
- Higher learning rate? Two big amplitude and diverge
- We want: slow learning in vertical direction and fast learning in x direction



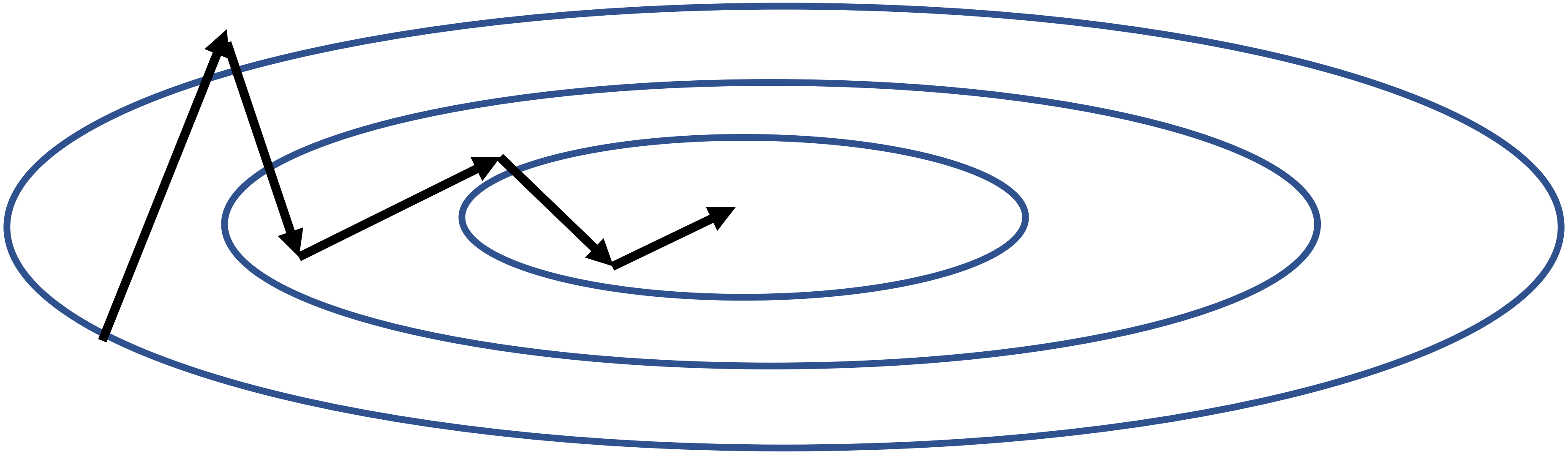
- Derivates ⇒ exponentially weighted averages (smoothing)

$$V_{dW} = \beta V_{dW} + (1-\beta)\frac{\partial L}{\partial W} \qquad V_{db} = \beta V_{db} + (1-\beta)\frac{\partial L}{\partial b}$$

$$W = W - \alpha V_{dW} \qquad \text{b} = b - \alpha V_{db}$$

- Oscillation in y direction: averages out (+,- changes)

Momentum gradient descent



RMSProp

- Other algorithm to achieve slower learning in y direction, and faster learning in x direction

Element-wise

$$S_{dW} = \beta S_{dW} + (1-\beta)\left(\frac{\partial L}{\partial W}\right)^2 \qquad S_{db} = \beta S_{db} + (1-\beta)\left(\frac{\partial L}{\partial b}\right)^2$$

$$W = W - \alpha \frac{1}{\sqrt{S_{dW}}}\left(\frac{\partial L}{\partial W}\right) \qquad \text{b} = b - \alpha \frac{1}{\sqrt{S_{db}}}\left(\frac{\partial L}{\partial b}\right)$$

- In y direction: relatively large derivative ⇒ dividing the learning rate with a larger number ⇒ smaller learning rate

- In x direction: derivative smaller ⇒ bigger learning rate

- You can use higher learning rate

Adam

- Combines the Momentum with RMSProp, so we get an even better algorithm
- In iteration t:

$$V_{dW} = \beta_1 V_{dW} + (1-\beta_1)\frac{\partial L}{\partial W} \qquad S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\left(\frac{\partial L}{\partial W}\right)^2$$

$$V_{dW}^{\text{corrected}} = \frac{V_{dW}}{1-\beta_1^t} \qquad S_{dW}^{\text{corrected}} = \frac{S_{dW}}{1-\beta_2^t}$$

$$W = W - \alpha \frac{1}{\sqrt{S_{dW}^{\text{corrected}}} + \varepsilon} V_{dW}^{\text{corrected}}$$

- Usually:
  - $\beta_1 = 0.9$ (≈ 10 step average)
  - $\beta_2 = 0.999$ (≈ 1000 step average)

Homework

- Linear regression notebook
- Neural networks notebook
- Linear regression and first part of neural networks (until backprop) is due next Tuesday
- Backprop: +1 week