Regularization Techniques for Deep Learning

Materials from

- Intel Deep Learning https://www.intel.com/content/www/us/en/developer/learn/course-deep-learning.html
- Improving Deep Neural Networks https://www.deeplearning.ai/

Legal Notices and Disclaimers

This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

This sample source code is released under the Intel Sample Source Code License Agreement.

Regularizing Neural Networks

We have various techniques to "regularize" neural networks – that is, to reduce overfitting

- **Regularization Penalty**: Add penalties (e.g., L1, L2) to the cost function.
- **Dropout**: Randomly deactivate neurons during training.
- **Early Stopping**: Halt training when performance stops improving on validation data.
- **Stochastic/Mini-Batch Gradient Descent**: Adds implicit regularization through noise.

  by introducing noise into the optimization process, stochastic or mini-batch gradient descent can prevent the model from overfitting to training data by promoting generalization, acting as a form of implicit regularization

Penalized Cost function

- One option is to explicitly add a penalty to the loss function for large weights.
- Analogous to the approach used in **Ridge Regression** (L2 regularization).

$$J = \frac{1}{2n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{m} W_i^2$$

- Can be applied in a similar manner to other loss functions, e.g. Categorical Cross Entropy

$$J = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k}) + \lambda \sum_{j=1}^{m} W_j^2$$

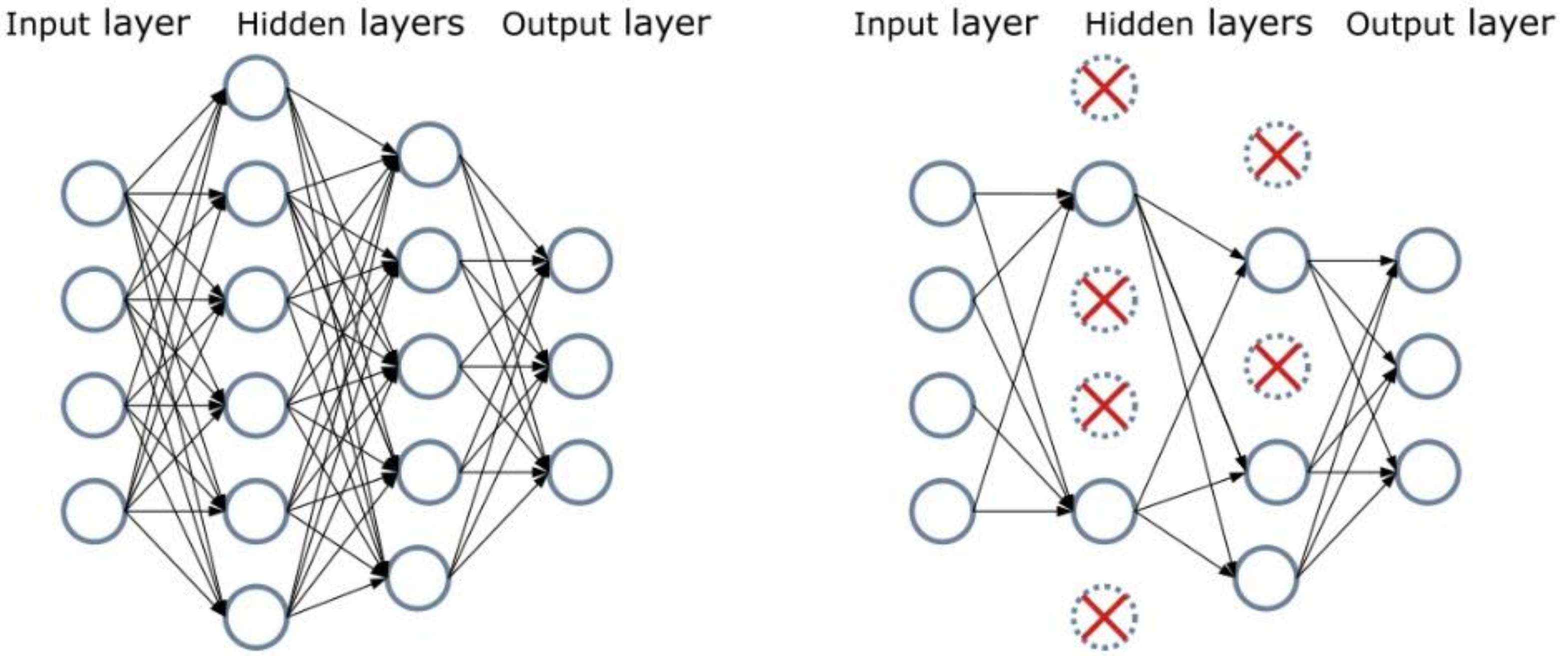
Dropout

- Dropout is a mechanism where we randomly deactivate a subset of neurons in the hidden layers during each training iteration

- This prevents the neural network from relying too much on individual pathways, making it more “robust”

- At test time, we rescale neuron weights to account for the percentage of time they were active during training

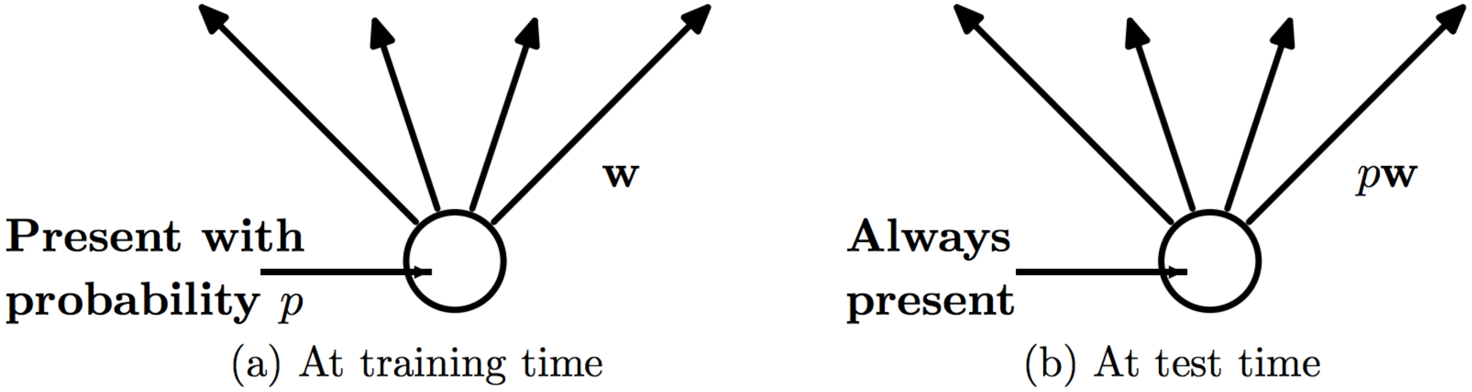
Dropout - Visualization



(a) Standard Neural Net (b) After applying dropout.

Dropout - Visualization

- If the neuron was present with probability *p*, at test time we scale the outbound weights by a factor of *p*.



Dropout – Inverted Dropout Implementation

- We typically implement dropout using inverted dropout because it simplifies the computation during inference (test time).
- If dropout = 0.2, then keep_prop = 1 – dropout = 1 – 0.2 = 0.8
- In inverted dropout, the activations are scaled during training by 1/keep_prob.
- This ensures that the output distributions remain consistent between training and testing, so no scaling is needed during inference.
- Scaling during training avoids extra operations at test time, making the implementation cleaner and more efficient.

Dropout – Inverted Dropout Implementation

considering at layer $\ell = 3$, dropout = 0.2 → keep_prop = 0.8

$$\mathrm{d}^{[3]} = \mathrm{random}\left(\mathrm{a}^{[3]}.\mathrm{shape}[0]\right) < \mathrm{keep\_prob}$$

$$a^{[3]} = a^{[3]} \cdot d^{[3]}$$

$$a^{[3]} = \frac{a^{[3]}}{keep\_prob}$$

$$z^{[4]} = W^{[4]}a^{[3]} + b^{[4]}$$

- The activations $a^{[3]}$ are scaled by $\frac{1}{keep\_prob} = \frac{1}{0.8}$ during training.
- This scaling compensates for the 20% of neurons that are randomly dropped out ($1 - \mathrm{keep\_prob} = 0.2$).
- By scaling $a^{[3]}$, the expected value of $z^{[4]}$ remains consistent with the scenario where no dropout occurs. This ensures that the network behaves similarly during inference (test time) without additional adjustments.

Early Stopping

- A heuristic regularization method to prevent overfitting.
- Stop training when validation performance stops improving.

- Example:
  - Check validation loss every 10 epochs.
  - If the loss increases, stop training and use the model from the best epoch.

Optimizers

- We have considered approaches to gradient descent which vary the number of data points involved in a step.
- However, they have all used the standard update formula:

$$\mathrm{W} := W - \alpha \cdot \nabla J$$

- There are several variants to updating the weights which give better performance in practice.
- These successive “tweaks” each attempt to improve on the previous idea.
- The resulting (often complicated) methods are referred to as “optimizers”.

Exponentially Weighted Averages

$\theta_1 = 40°F$ 4°C ←

$\theta_2 = 49°F$ 9°C

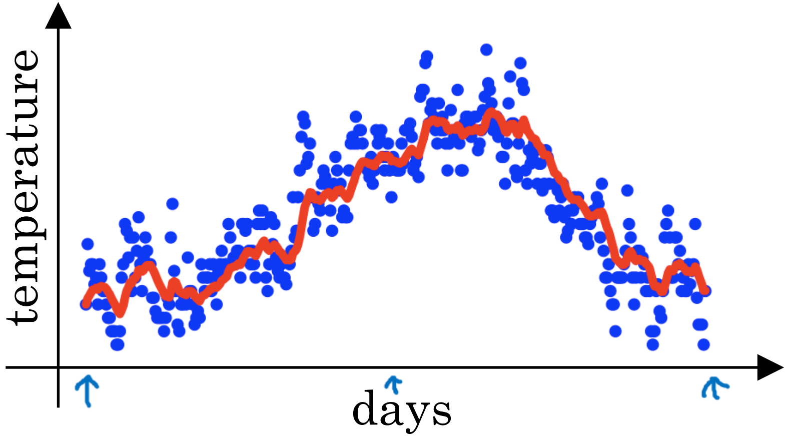
$\theta_3 = 45°F$ ⋮

⋮

$\theta_{180} = 60°F$ 15°C

$\theta_{181} = 56°F$ ⋮

⋮



$V_0 = 0$

$V_1 = 0.9\,V_0 + 0.1\,\theta_1$

$V_2 = 0.9\,V_1 + 0.1\,\theta_2$

$V_3 = 0.9\,V_2 + 0.1\,\theta_3$

⋮

$$\boxed{V_t = 0.9\,V_{t-1} + 0.1\,\theta_t}$$

Andrew Ng

Exponentially weighted averages

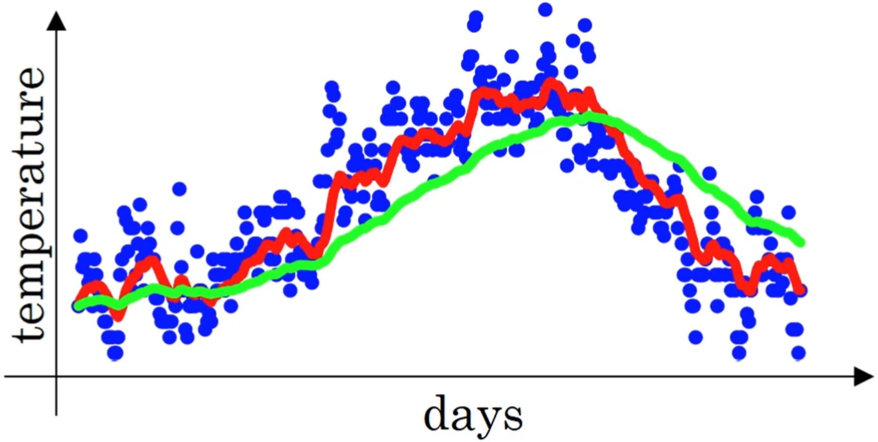
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$\beta = 0.9$: ≈ 10 days' temperature.

$\beta = 0.98$: ≈ 50 days

V_t as approximately average over $\approx \frac{1}{1-\beta}$ days' temperature.

$$\frac{1}{1-0.98} = 50$$



Andrew Ng

Exponentially weighted averages

moving

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t \leftarrow$$

$\beta = 0.9$: ≈ 10 days' temperature.

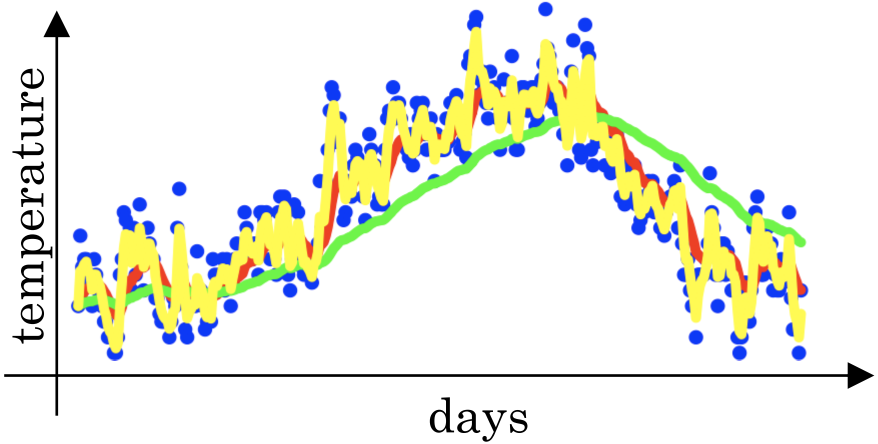
$\beta = 0.98$: ≈ 50 days

$\beta = 0.5$: ≈ 2 days

V_t as approximately average over

$\rightarrow \approx \frac{1}{1-\beta}$ days' temperature.

$$\frac{1}{1-0.98} = 50$$



Andrew Ng

Momentum

- Idea: Only change direction slightly with each step to stabilize updates.
- Use a "running average" of previous step directions to smooth out variations in gradients.

$$v_{\mathrm{t}} = \beta \cdot v_{t-1} + (1-\beta) \cdot \nabla J$$

$$\mathrm{W} = W - \alpha \cdot v_t$$

- Here, β is referred to as the "momentum". It is generally given a value <1 (e.g., 0.9 is a common value)

Momentum

- Idea: Only change direction slightly with each step to stabilize updates.
- Use a "running average" of previous step directions to smooth out variations in gradients.

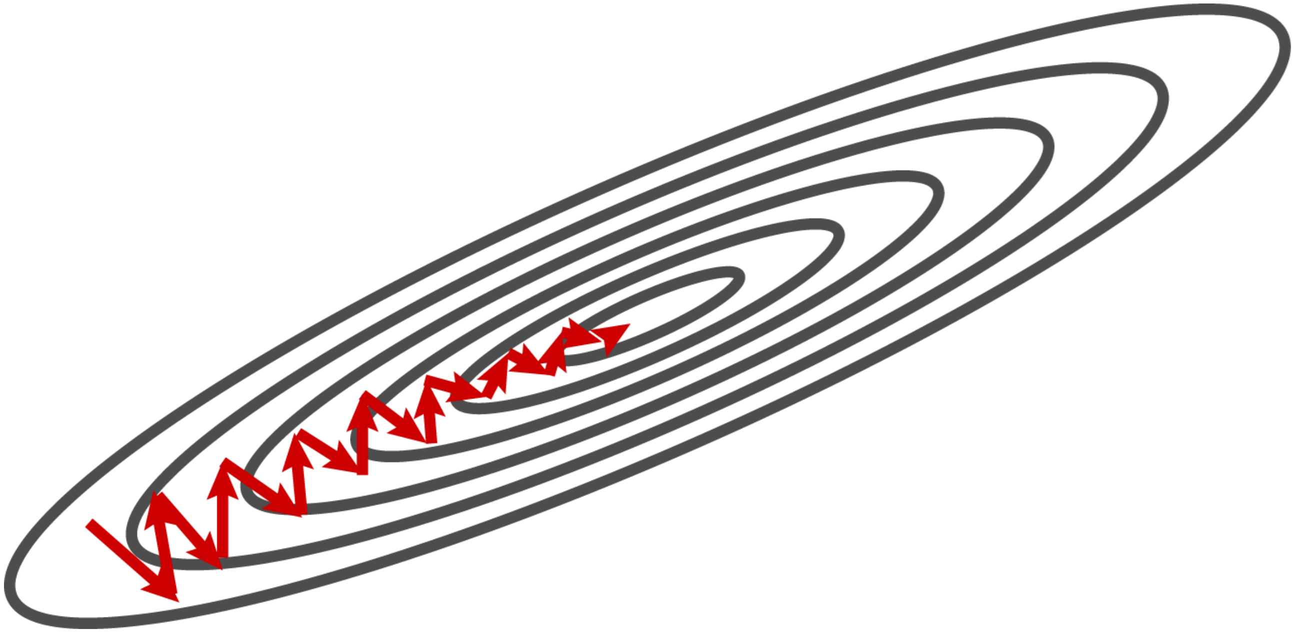
$$v_t = \beta \cdot v_{t-1} + (\cancel{1 - \beta}) \cdot \nabla J$$

often omitted in literatures

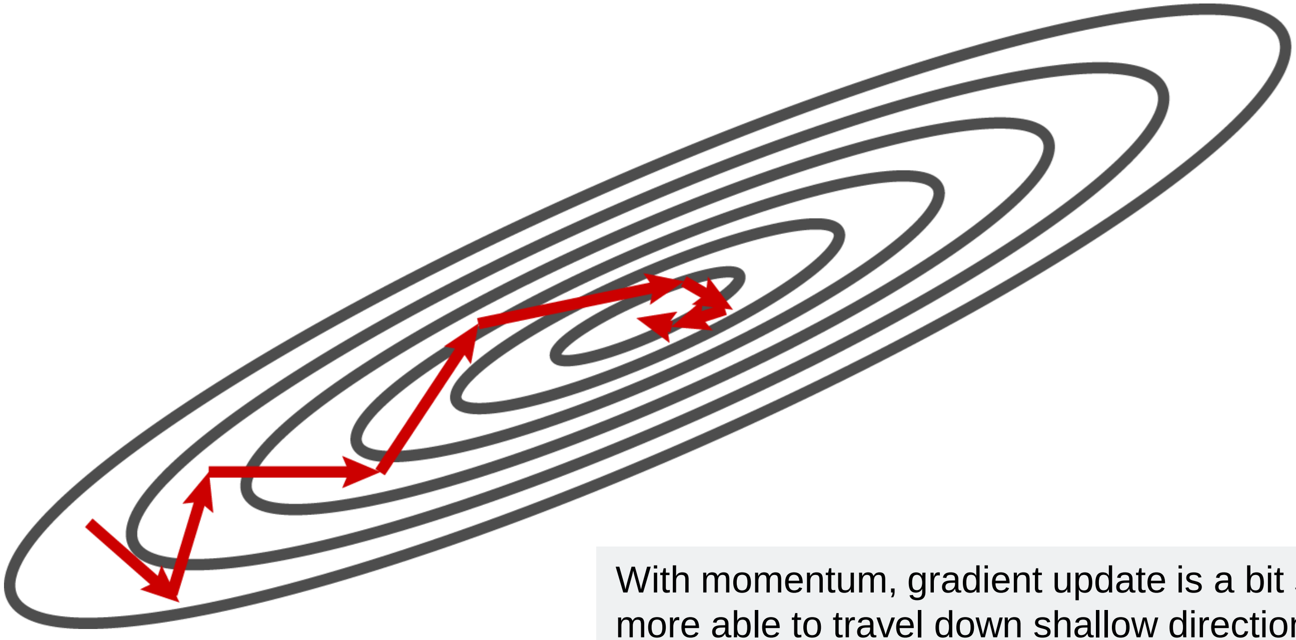
$$W = W - \alpha \cdot v_t$$

- Here, β is referred to as the "momentum". It is generally given a value <1 (e.g., 0.9 is a common value)

Gradient Descent vs Momentum



Gradient Descent vs Momentum



With momentum, gradient update is a bit smoother and more able to travel down shallow directions.

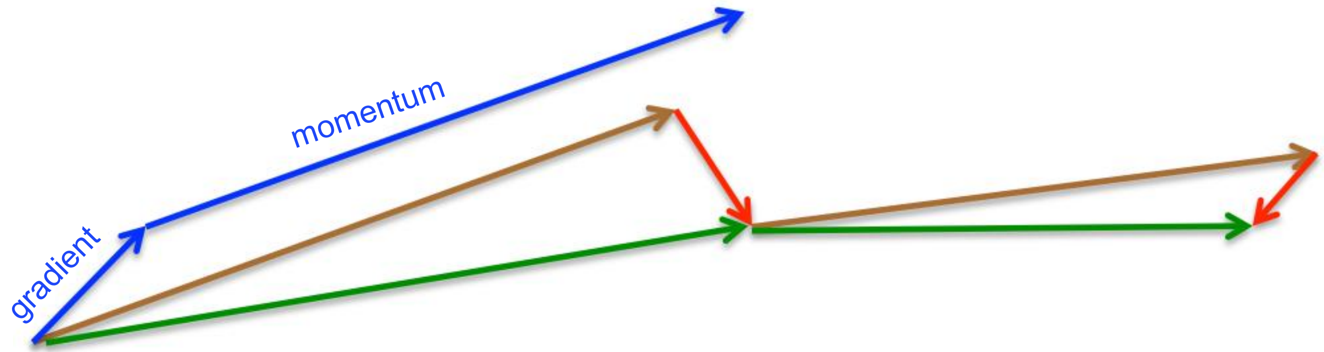
Nesterov Momentum

- Idea: Control "overshooting" by anticipating the next position (lookahead).
- Lookahead Concept: Ensures smoother and more accurate updates by accounting for the momentum's effect before applying the gradient.
- Apply the gradient to the "lookahead" position rather than the current position.

$$v_t = \beta \cdot v_{t-1} + \alpha \, \nabla J(\overbrace{W - \beta \cdot v_{t-1}}^{\text{lookahead}})$$

$$\mathrm{W} = W - v_t$$

Nesterov Momentum



Momentum Vector

Nesterov steps

Gradient/correction

Standard momentum steps

AdaGrad (Adaptive Gradient Optimizer)

- Idea: Scale updates for each weight independently by normalizing with past gradients.
- Divide new updates by factor of previous sum

$$W = W - \frac{\alpha}{\sqrt{G_t + \epsilon}} \nabla J \qquad G_t = G_{t-1} + (\nabla J)^2$$

- Instead of a constant learning rate, the effective learning rate is then divided by the square root of the sum of each component separately.
- Weights with **larger gradients** get **lower learning rates**, while those with smaller gradients get **higher learning rates.**
- However, the **aggressive decay** in the learning rate can slow down convergence, especially in later stages of training.

RMSProp (Root Mean Square Propagation)

- Idea: Similar to Adagrad, but **decays older gradients** exponentially to prioritize recent updates, rather than using the sum of previous gradients
- Adapts learning rates to recent gradient trends, making it more robust to shifts in the shape of the loss function, such as steep valleys.
- Suppresses learning rates for weights with frequent large gradients, helping stabilize training.

$$W = W - \frac{\alpha}{\sqrt{S_t} + \epsilon} \nabla J$$

$$S_t = \beta\, S_{t-1} + (1-\beta)(\nabla J)^2$$

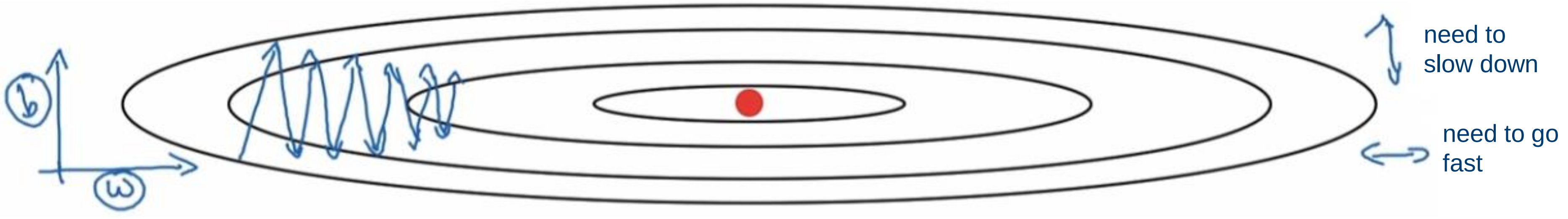
RMSProp (Case Study)

$$S_{dW} = \beta\, S_{dW_{prev}} + (1-\beta)(dW)^2$$

$$S_{db} = \beta\, S_{db_{prev}} + (1-\beta)(db)^2$$

$$W = W - \alpha \frac{dW}{\sqrt{S_{dW}} + \epsilon}$$

$$b = b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$$



Adam (Adaptive Moment Estimation)

- Idea: blending between momentum and RMSprop.
- For iteration t:

$$\mathrm{V_{dW}} = \beta_1 \mathrm{V}_{\mathrm{dW}_{prev}} + (1-\beta_1)\mathrm{dW} \qquad \mathrm{S_{dW}} = \beta_2 \mathrm{S}_{\mathrm{dW}_{prev}} + (1-\beta_2)(\mathrm{dW})^2$$

$$\hat{\mathrm{V}}_{\mathrm{dW}} = \frac{\mathrm{V_{dW}}}{1-\beta_1^t} \qquad \hat{\mathrm{S}}_{\mathrm{dW}} = \frac{\mathrm{S_{dW}}}{1-\beta_2^t}$$

$$W := W - \alpha \frac{\hat{\mathrm{V}}_{\mathrm{dW}}}{\sqrt{\hat{\mathrm{S}}_{\mathrm{dW}}} + \epsilon}$$

Which one should I use?

- **RMSProp** and **Adam** are widely used and effective for most problems.
- The best choice depends on the specific problem and dataset, and it is often hard to predict in advance.
- Optimization algorithm selection remains an **active area of research and experimentation.**

Software