Regularization in Neural Networks

Designed to prevent over-fitting on a specific training set.

Three methods

- Penalize large weights via an additional term in the objective on the norm of the parameters. (Traditional method of regularization) Ridge regularization
- Early stopping:
- Dropout:

Penalizing norm of parameters in the objective

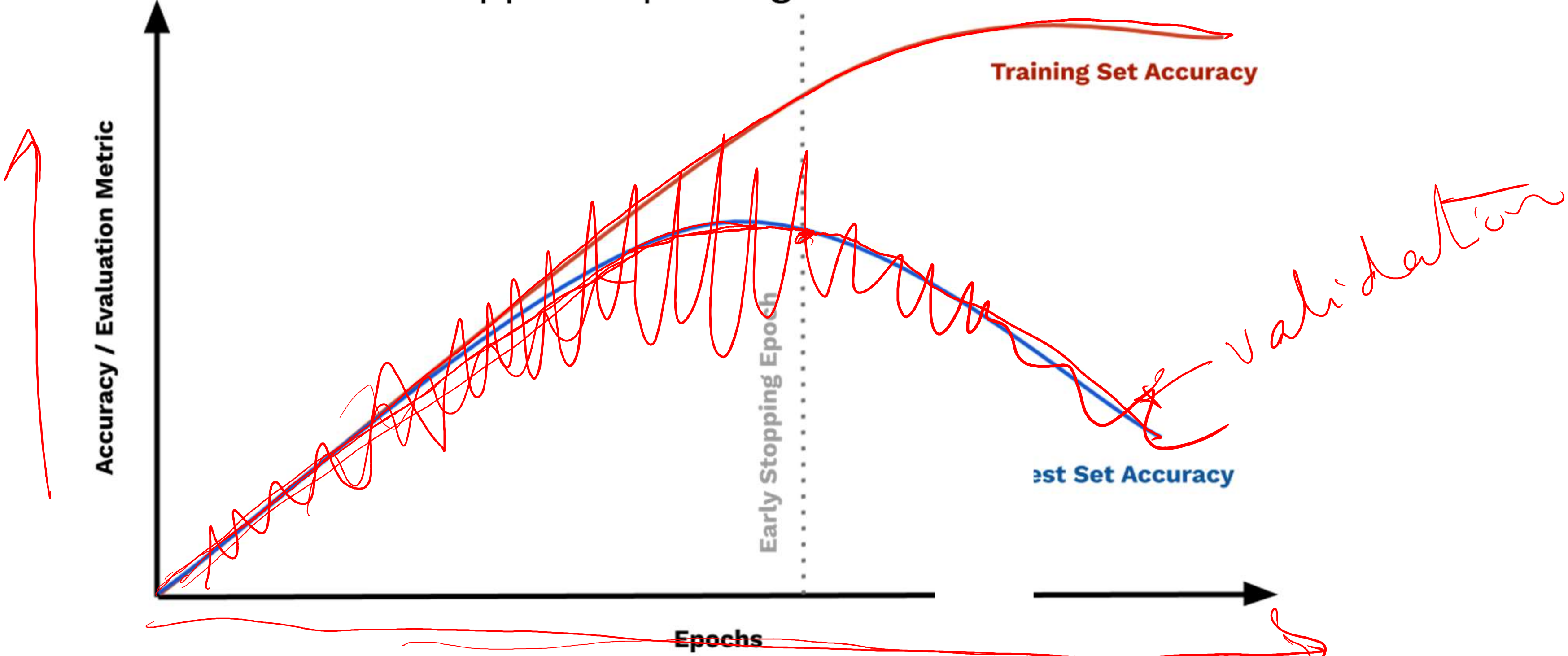
$$\underset{\underline{W}}{\text{minimize}} \sum_{(x^i, y^i) \sim B} L(\underline{W}, (x^i, y^i)) + \underbrace{C\|\underline{W}\|^2}_{\text{Regularizer}}$$

keeps parameters close to zero.

- Does not allow any one dimension to assume a large importance
- Does not allow parameters to swing much to small changes in the training set shrinkage
-

Regularization: Early stopping

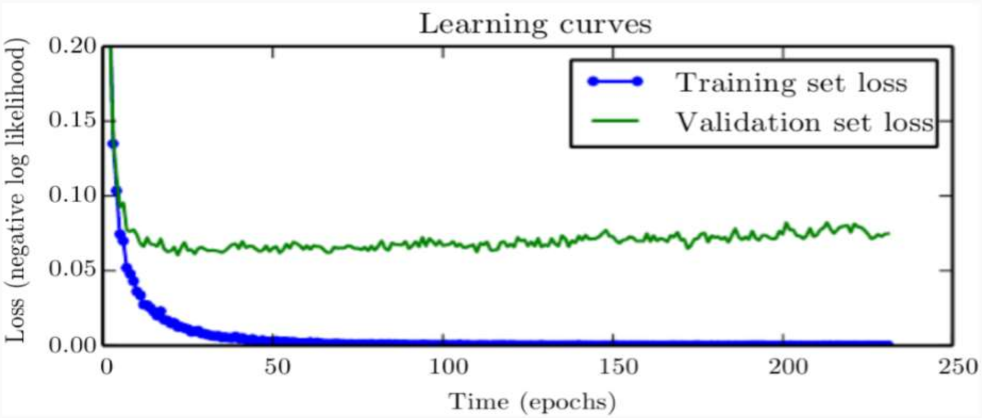
Early stopping: Stop training when performance on a validation set has stopped improving



Cut off training when the validation error has not decreased by more than some small amount ϵ for some number of epochs

Motivated by observations that even a huge network first fits the clean labels in the data and then the noise

Regularization: Early stopping

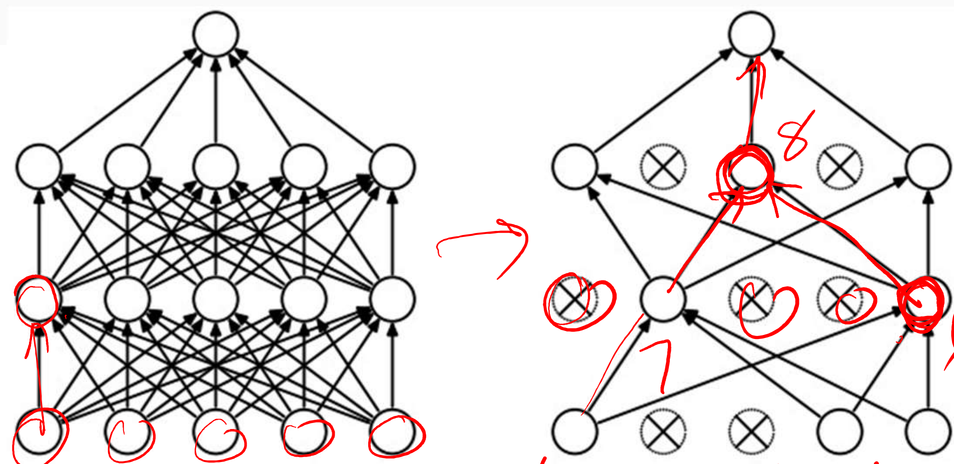


- Training error reduces with training iteration but validation error dips and then increases
- Stop training when error on a set-aside validation set increases more than a certain number of times.
- Why does early stopping help? restricts search space among parameters reachable within a limited capacity of the starting point. Prevents over-fitting.

Regularization: Dropout

Benign over-fitting
To know more search for

- Cheap, yet effective method of creating ensembles.
- During training: randomly zero-out the outputs from a subset (half) of the hidden units. Ensures no over-fitting on any single unit. Forces other units to learn useful outputs.
- During deployment: no drop-out but adjust for the dropped units.

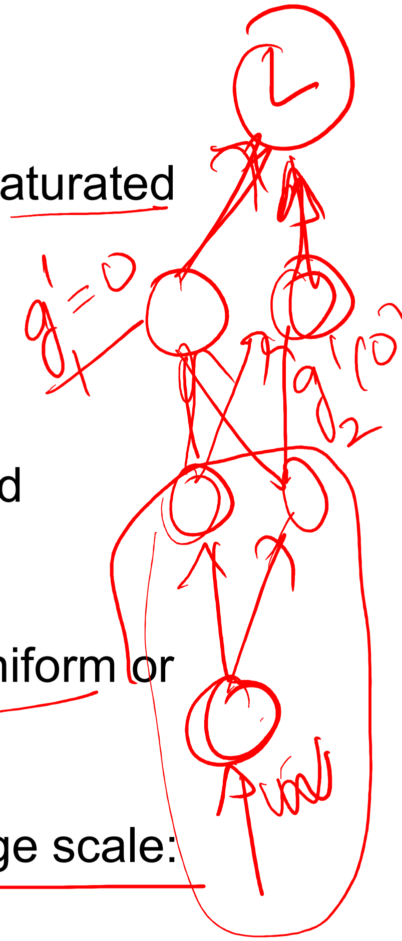


(a) Standard Neural Net (b) After applying dropout.

$w_{61} x_1 * 5/3 + w_{62} x_2 * + w_{63} x_3 * + w_{64} x_4 * 5/3 + w_{64} x_5 * 5/3$

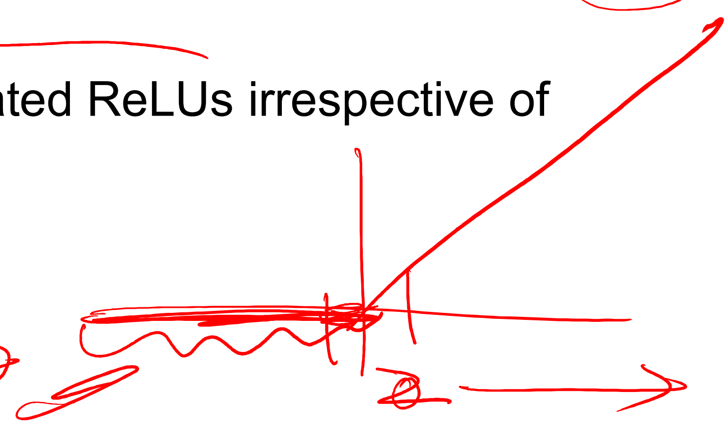
Another explanation: Training parameters with small perturbations of the inputs leads to smooth functions

Parameter Initialization



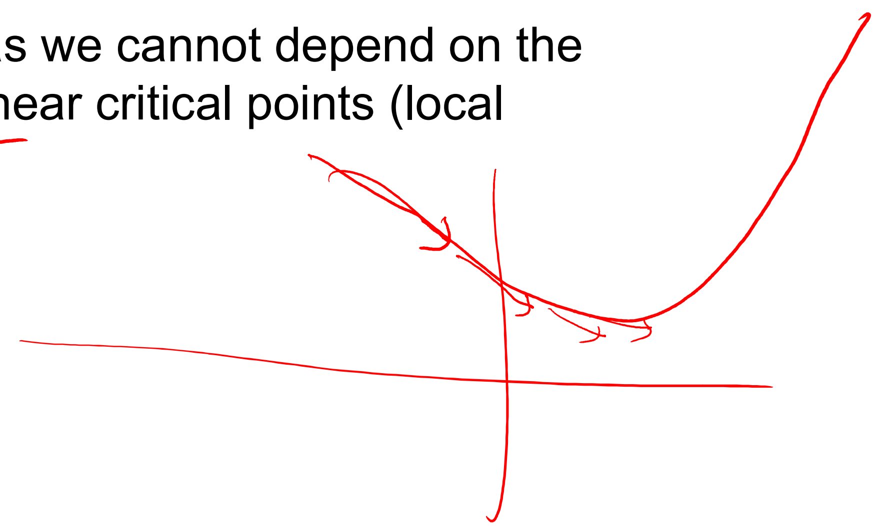
- Bad initialization could hinder convergence, cause numerical problems, saturated activation layers,slow training.
- Some guiding principles
  - Need to break symmetry for hidden units of fully connected feed-forward networks.
    - Sample from high-entropy distribution. • Typical distributions used: Uniform or Gaussian.
    - Scale of distribution important: small-scale symmetry not broken. Large scale: saturated ReLUs. Hyper-parameter tuned via a validationset.
  - Small values for bias parameters to ensure non-saturated ReLUs irrespective of the input

saturated region →



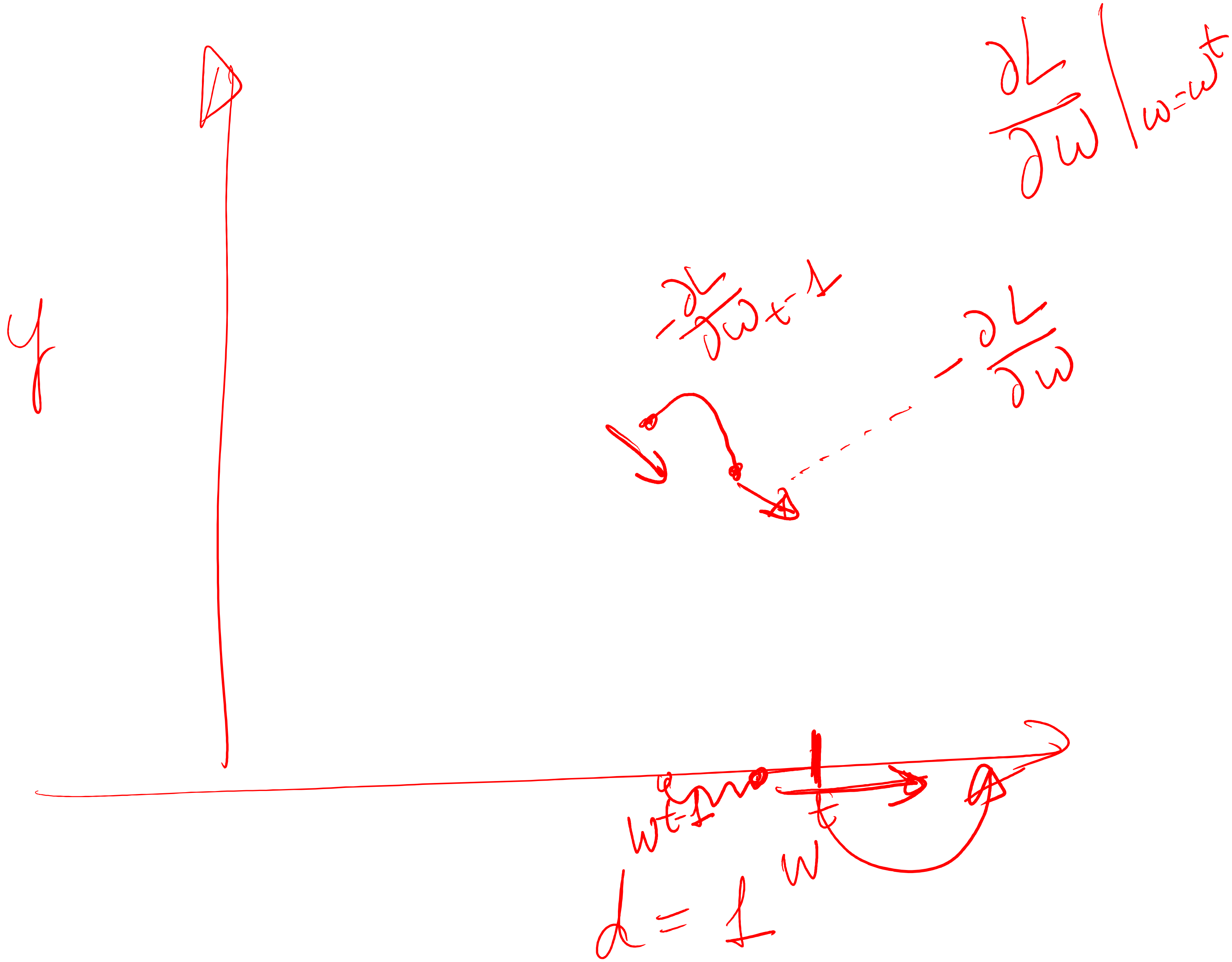
Limitations of plain Stochastic Gradient descent

- SGD works with a noisy approximation of the real gradients.
- Errors in estimated gradients reduce as $\sqrt{Batch\ size}$.
- Learning rate cannot be kept constant as we cannot depend on the gradient being small to proceed slowly near critical points (local minima)



Variants of SGD

- Momentum methods: modify the gradients based on previous gradients. Helps overcomes two problems:
  - ill-conditioned loss curvature where the rate of change varies a lot along different directions
  - In-exact gradients based on small samples.
- Choosing adaptive learning rate becomes important (Adagrad and Adam)


$\frac{\partial L}{\partial w}\Big|_{w=w^t}$
y
$-\frac{\partial L}{\partial w_{t-1}}$
$-\frac{\partial L}{\partial w}$
w^{t-1}
w^t
w
$\alpha = 1$

SGD with Momentum

- Noise in SGD is reduced by averaging gradients over multiple examples, however, limited by batch size.
- Momentum: weighted average from previous gradients.
- $$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\nabla_{\mathbf{w}_t} L(\mathbf{w}_t) \qquad \text{for } 0 \leq \beta < 1$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta \mathbf{v}_t$$

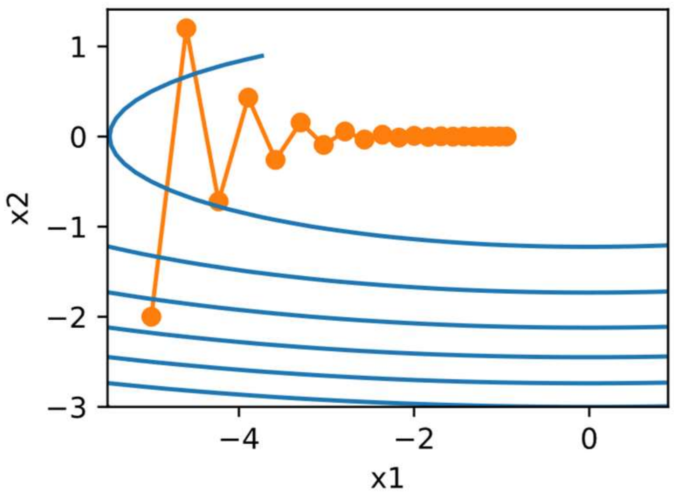
The $\mathbf{v_t}$ is like a “velocity” that simulates how a heavy ball rolling down a hill accumulates velocity.

Why does momentum work?

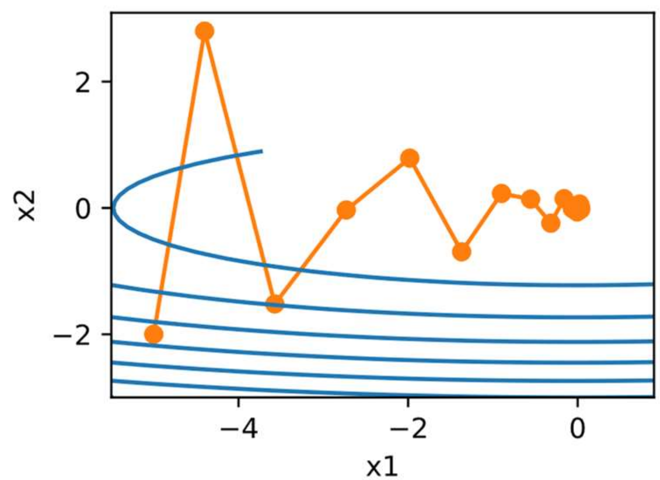
- When a function changes very rapidly in one dimension and slowly in another dimension (Ill-conditioned functions), a common learning rate across all dimensions is either too slow for all, or causes oscillation in one dimension.

- Example: $F(w_1, w_2) = 0.1w_1^2 + 2w_2^2$

- Minima at (0,0).



Standard gradient descent



https://colab.research.google.com/drive/104UVC56ZKVAt0HDGQyqv5lsg_PsEewRK?usp=sharing

Optimization Algorithms (II)

- "**RMSProp (Root Mean Squared Propagation)**" weight update rule:

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta}{\sqrt{\mathbf{s}_t} + \epsilon} \odot \mathbf{g}_t$$

$$\mathbf{g}_t = \nabla_{\mathbf{w}_t} L(\mathbf{w}_t)$$

Element-wise multiplication

- Need an adaptive learning rate that adapts to each dimension. Particularly useful for sparse features which might get updates infrequently.

Optimization Algorithms (III)

- "**Adam**" weight update rule: Makes use of both momentum and adaptive learning rate

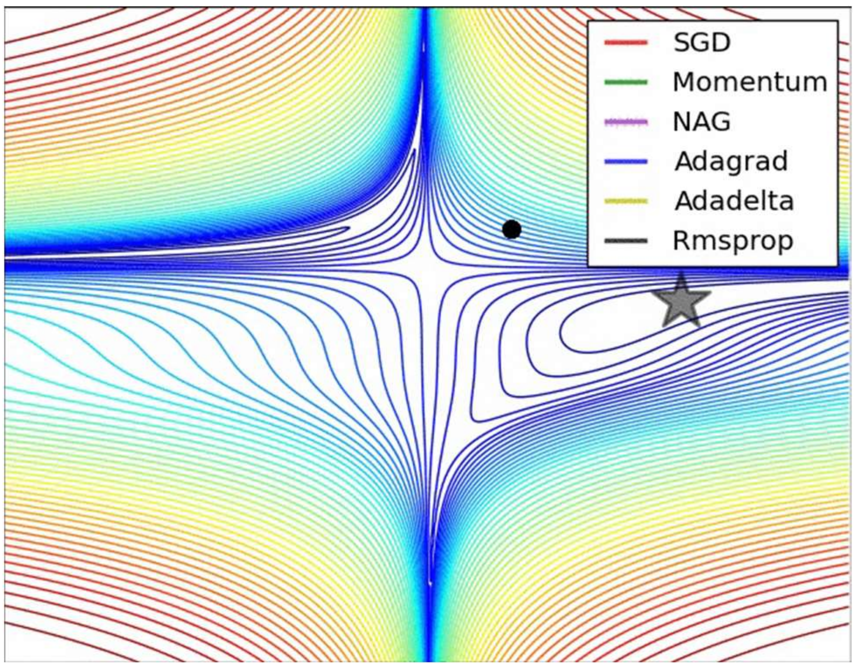
$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\mathbf{g}_t$$

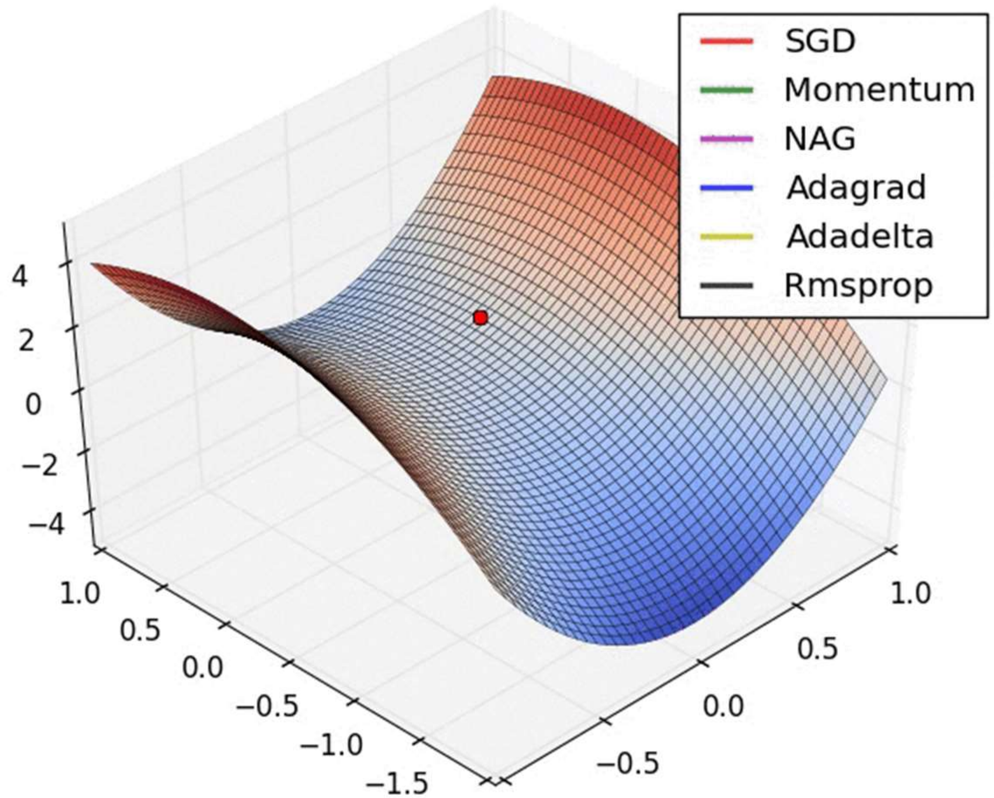
$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t$$

$$\hat{\mathbf{s}}_t \leftarrow \frac{\mathbf{s}_t}{1-\gamma^t} \qquad \hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1-\beta^t}$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$$

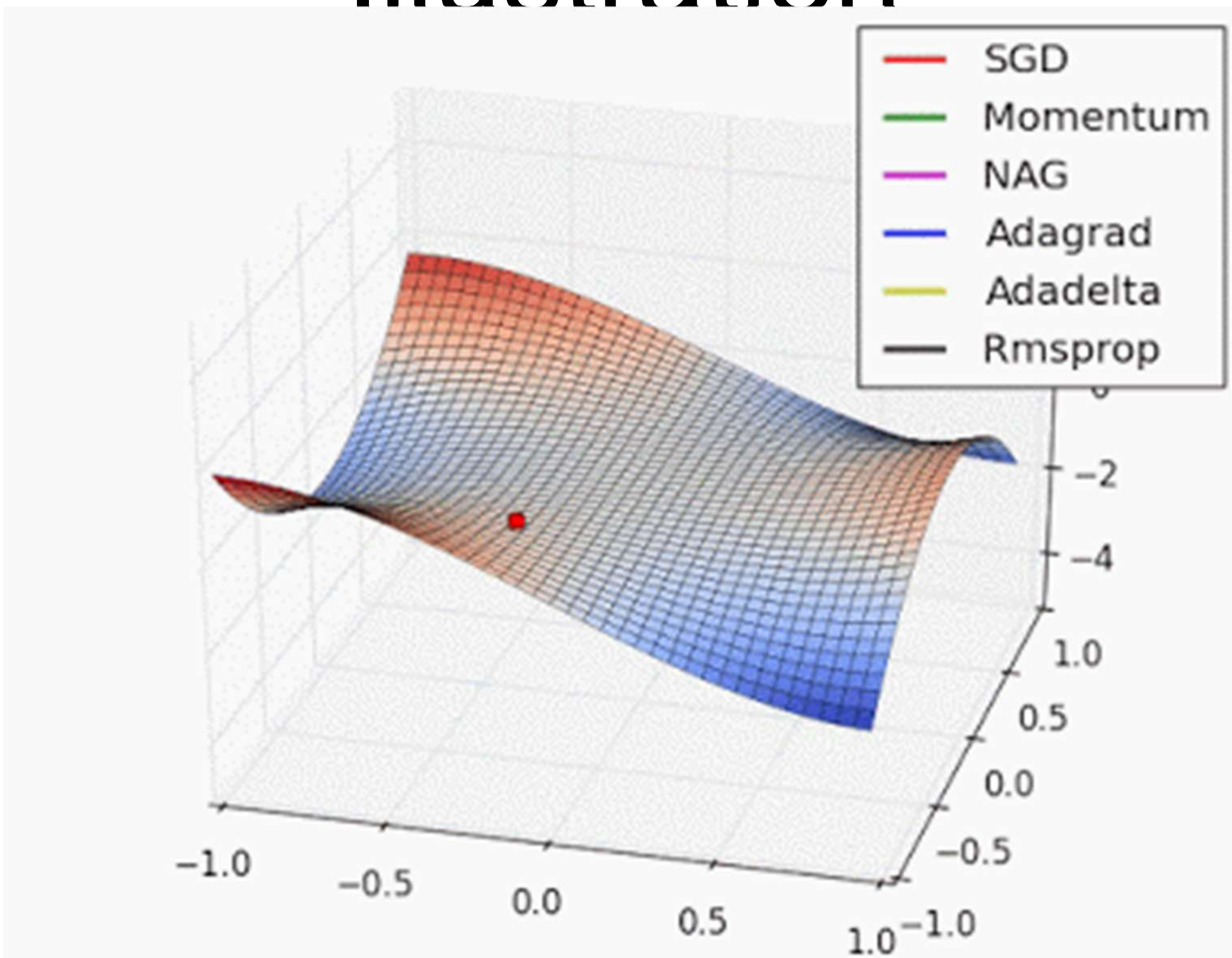
Illustration





Images credit: [Alec Radford](#).

Illustration

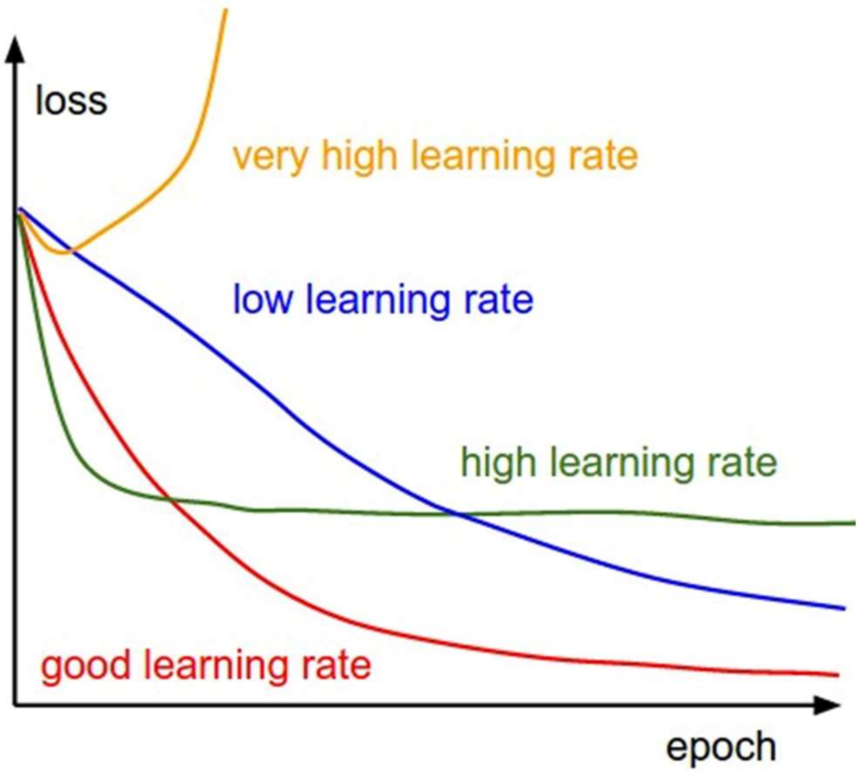


Images credit: Alec Radford.

distil.pub search for momentum.

Learning Rate Schedule

- Observe training losses to understand the effect of different learning rates
- Helpful to decay the learning rate over time. E.g. step decay, exponential decay, etc.
- Adaptive learning rate methods like Adagrad, Adam are popular optimizers.



Animation from: http://cs231n.github.io/neural-networks-3/

Good reference for optimizers: https://ruder.io/optimizing-gradient-descent/