Gradient Descent and Optimization

Deep Learning

Brad Quinton, Scott Chin

Learning Objectives

- Gain a deeper understanding of the optimization problem of training a neural network
- Learn about the various popular variants of Gradient Descent
  - Mini-batch Gradient Descent
  - Stochastic Gradient Descent
  - Gradient Descent with Momentum and Nesterov Momentum
  - Adagrad (Adaptive Learning Rates)
  - RMSProp
  - Adam

Deep Learning So Far

- Architecture (we have spent a lot of time on this)
  - Provides the capacity to learn the approximate mapping function

Deep Learning So Far

- Architecture (we have spent a lot of time on this)
  - Provides the capacity to learn the approximate mapping function

- Optimizer
  - Get the network to reach it’s potential by finding good parameter values

Optimization

- In general, define a Cost Function (aka Objective Function) to measure the quality of a solution
  - Cost function models desired traits (objectives) of the solution
  - In our case, the solution is a set of parameter values. The objective is to minimize difference between model prediction and actual labels
- Use an optimization algorithm (aka optimizer) to search/solve for a solution that minimizes/maximizes the Cost Function
  - In our case, infeasible to “solve” for an optimal solution
  - Instead, iteratively search for a “good-enough” solution

Training a Neural Network Classifier

$$J = f(y, \hat{y})$$

- Objective is to minimize difference between true label y and predicted label $\hat{y}$

Training a Neural Network Classifier

$$J = f(y, \hat{y})$$

$$\hat{y} = h(x, w, b)$$

- Objective is to minimize difference between true label y and predicted label $\hat{y}$
- Prediction is a function of the input x and network parameters w, b

Training a Neural Network Classifier

$$J = f(y, \hat{y})$$

$$\hat{y} = h(x, w, b)$$

$$J = f(y, x, w, b)$$

- Objective is to minimize difference between true label y and predicted label $\hat{y}$
- Prediction is a function of the input x and network parameters w, b
- Ultimately, training objective is a function of current parameter values w, b, sample x, and true label y

Training a Neural Network Classifier

$$J = f(y, \hat{y})$$

$$\hat{y} = h(x, w, b)$$

$$J = f(y, x, w, b)$$

$$J = f(w, b)$$

- Objective is to minimize difference between true label y and predicted label $\hat{y}$
- Prediction is a function of the input x and network parameters w, b
- Ultimately, training objective is a function of current parameter values w, b, sample x, and true label y
- For a given training set, x, y are constants

Training a Neural Network Classifier

$$J = f(y, \hat{y})$$

$$\hat{y} = h(x, w, b)$$

$$J = f(y, x, w, b)$$

$$J = f(w, b)$$

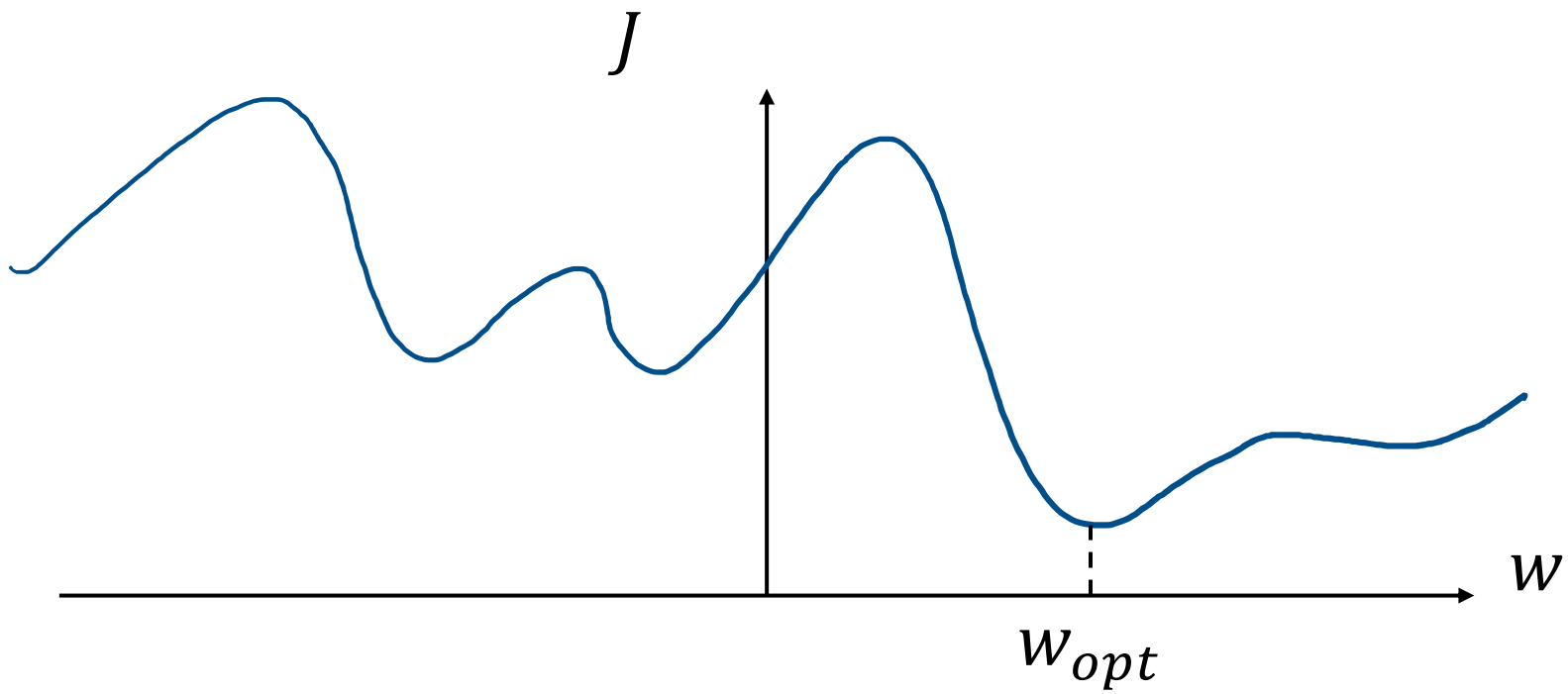
$$J = f(w)$$

- Objective is to minimize difference between true label y and predicted label $\hat{y}$
- Prediction is a function of the input x and network parameters w, b
- Ultimately, training objective is a function of current parameter values w, b, sample x, and true label y
- For a given training set, x, y are constants
- For this discussion, let's use w to represent all parameters (including biases b)

Training a Neural Network Classifier

$$J(w)$$

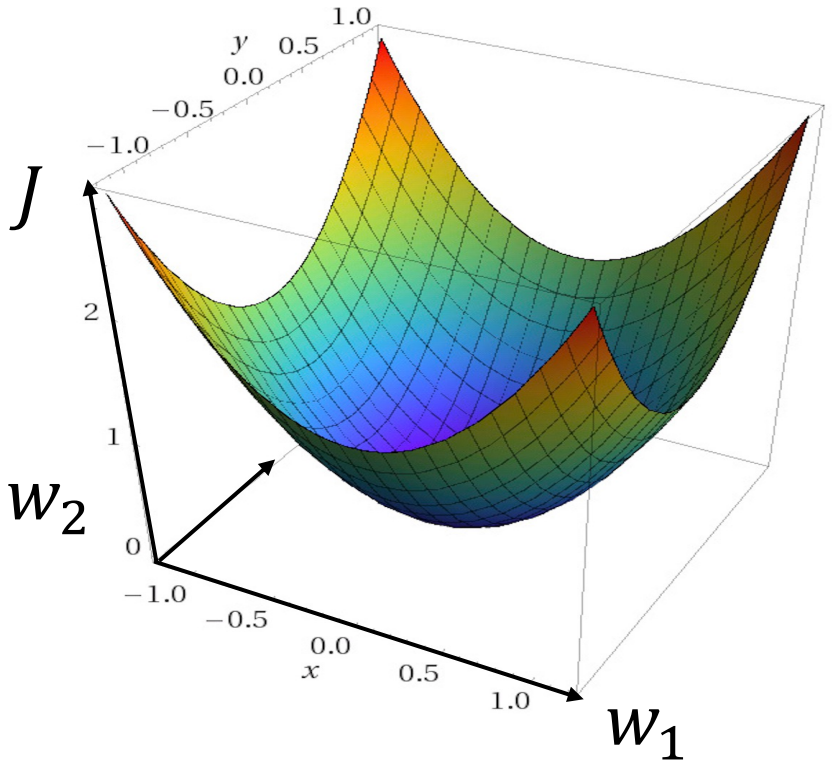
- For a given network architecture and training set, the Cost function is only a function of the network parameters, w

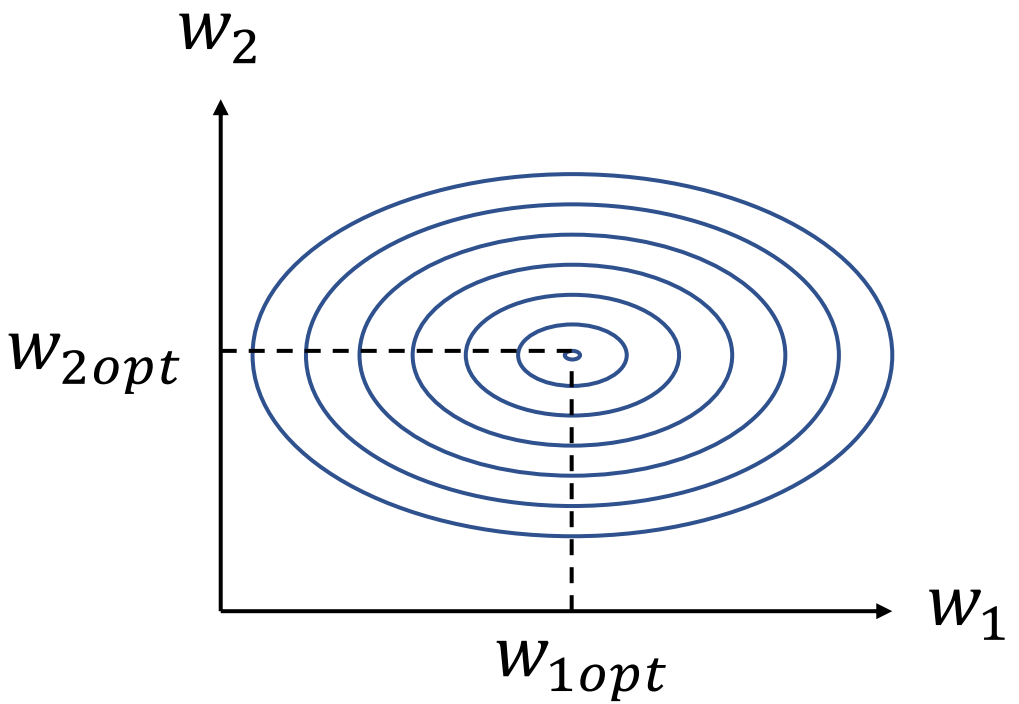


Training a Neural Network Classifier

$$J(w)$$

- For a given network architecture and training set, the Cost function is only a function of the network parameters, w





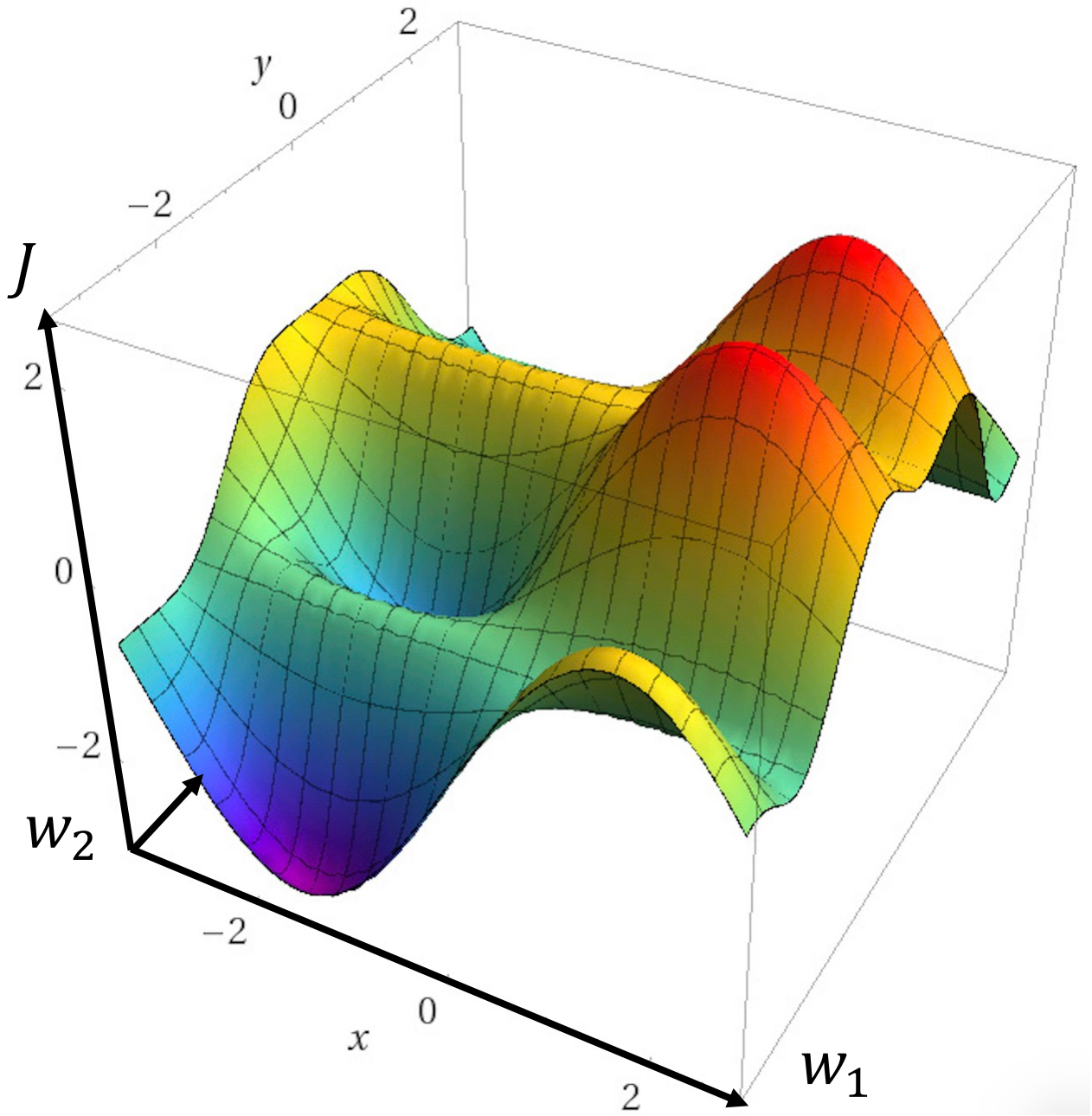
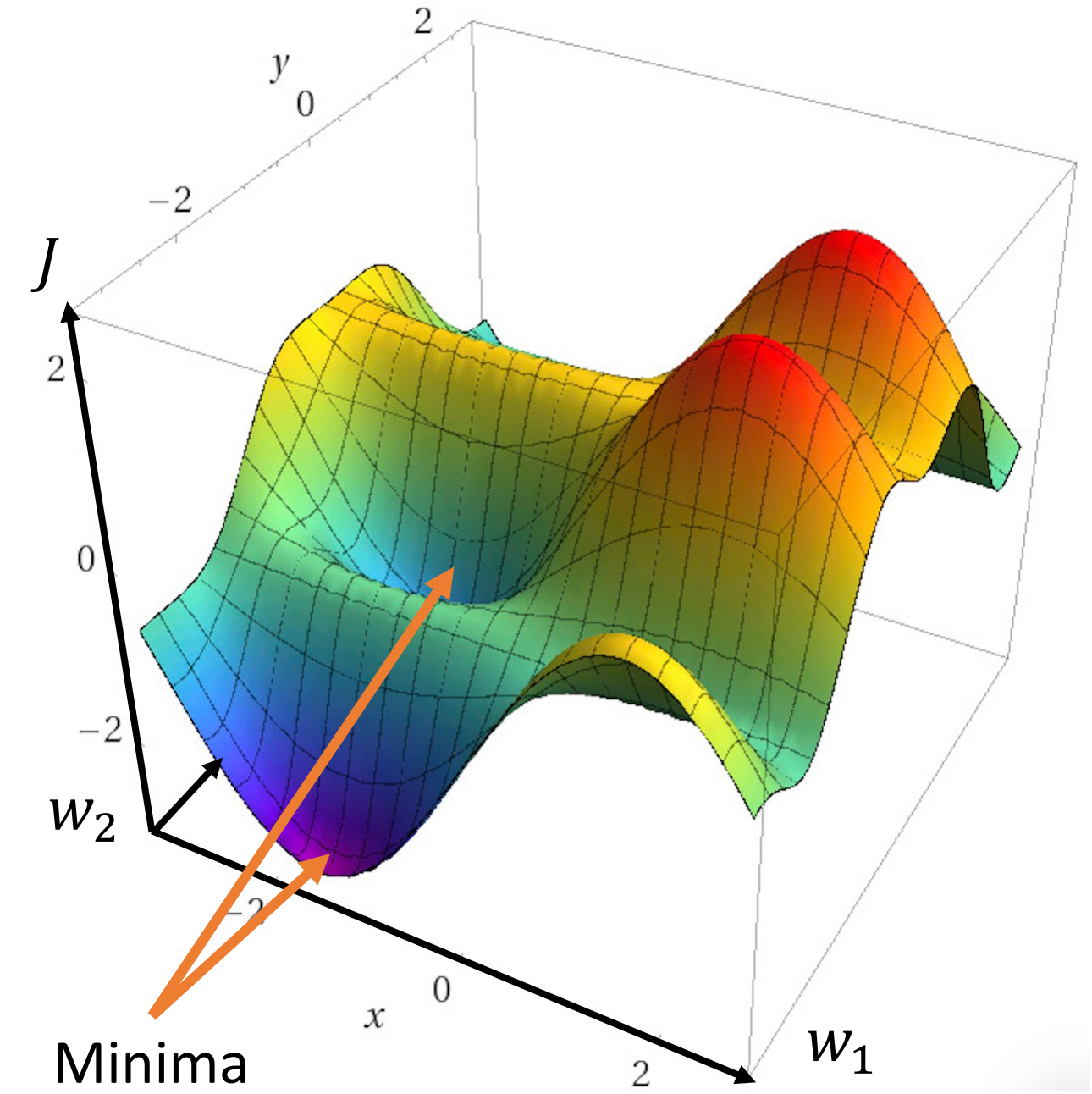


Image Credit: https://www.oreilly.com/radar/the-hard-thing-about-deep-learning/



Minima

- Convex in all input variables

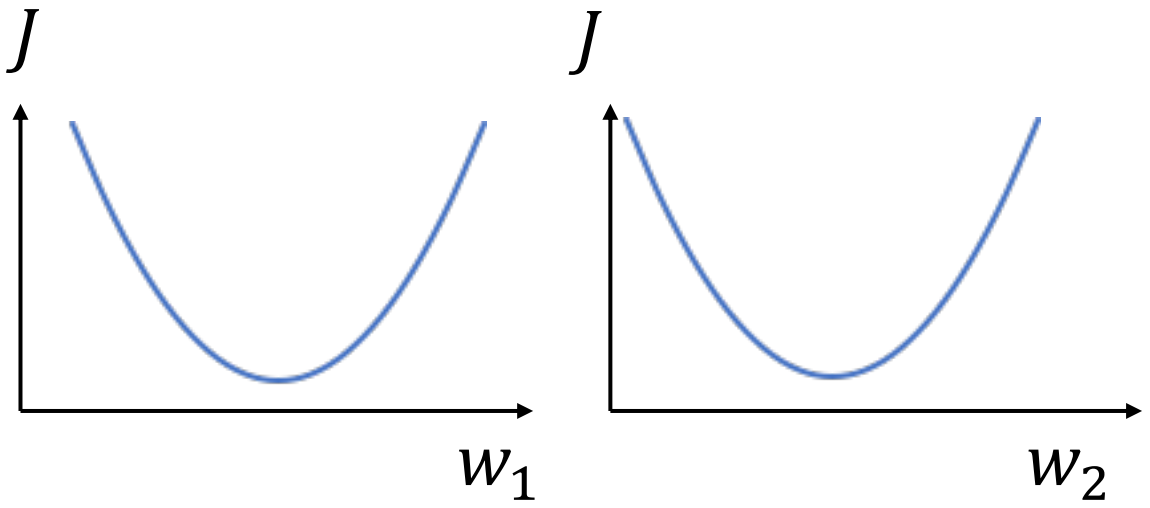
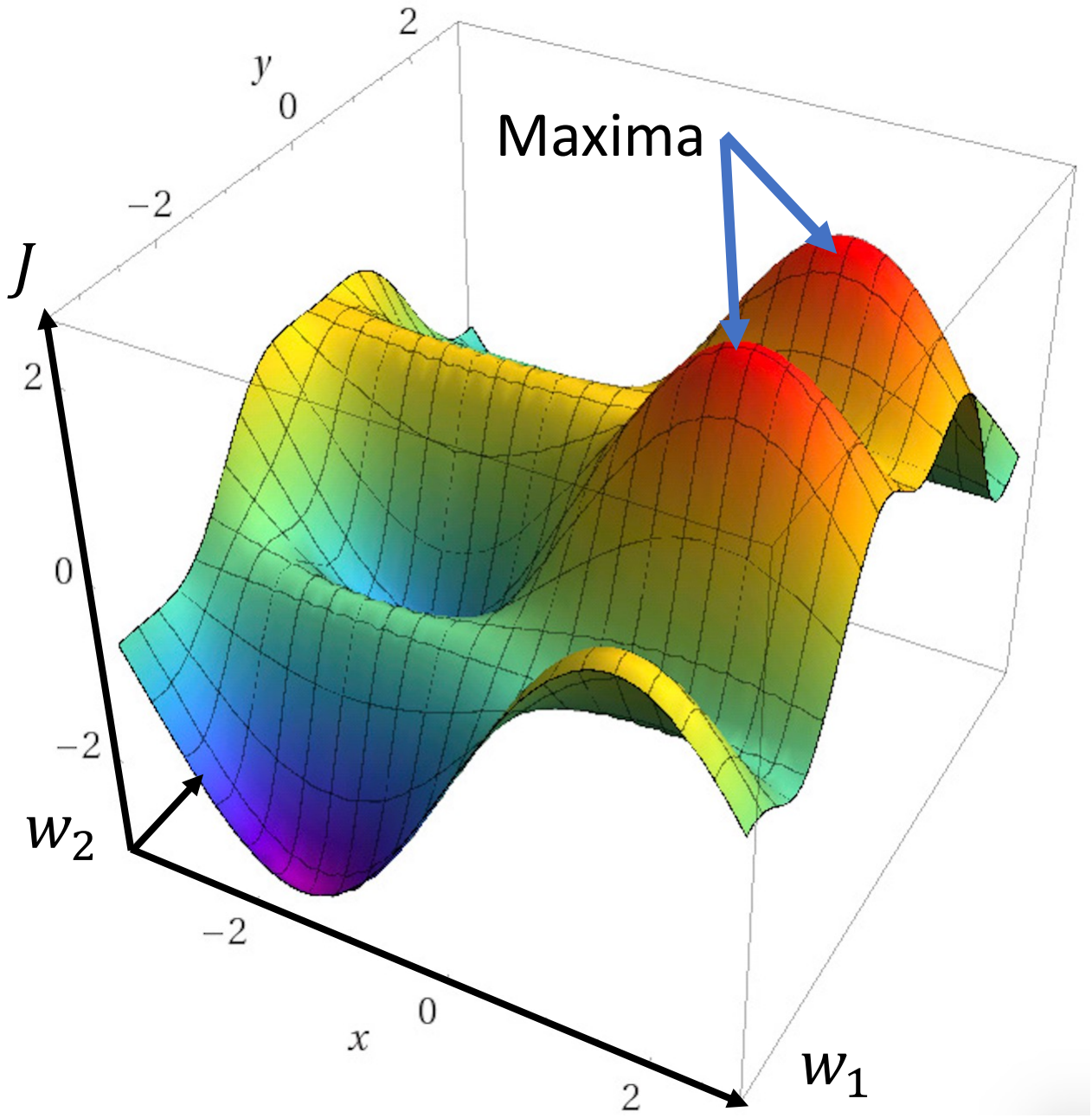


Image Credit: https://www.oreilly.com/radar/the-hard-thing-about-deep-learning/



Maxima

- Concave in all input variables

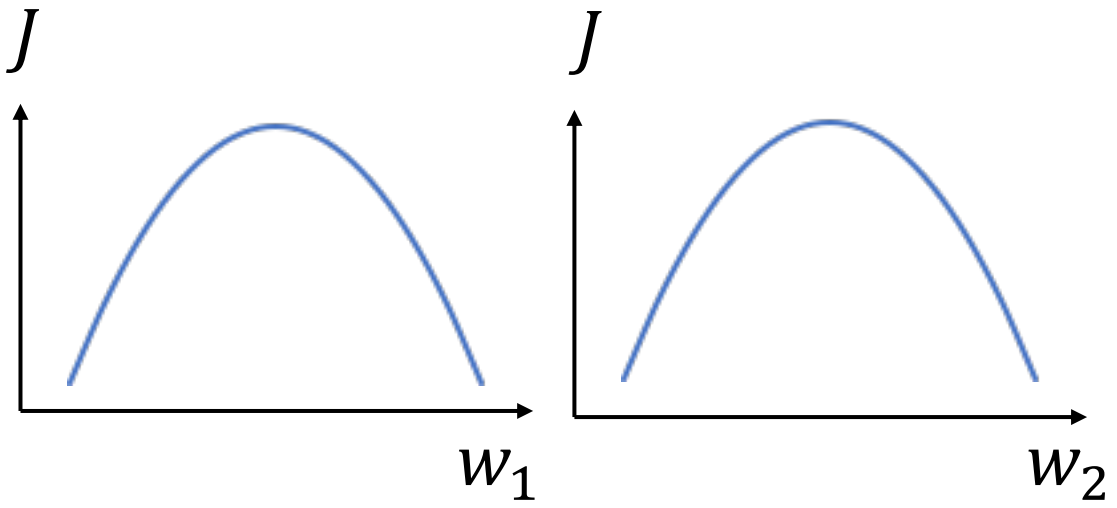
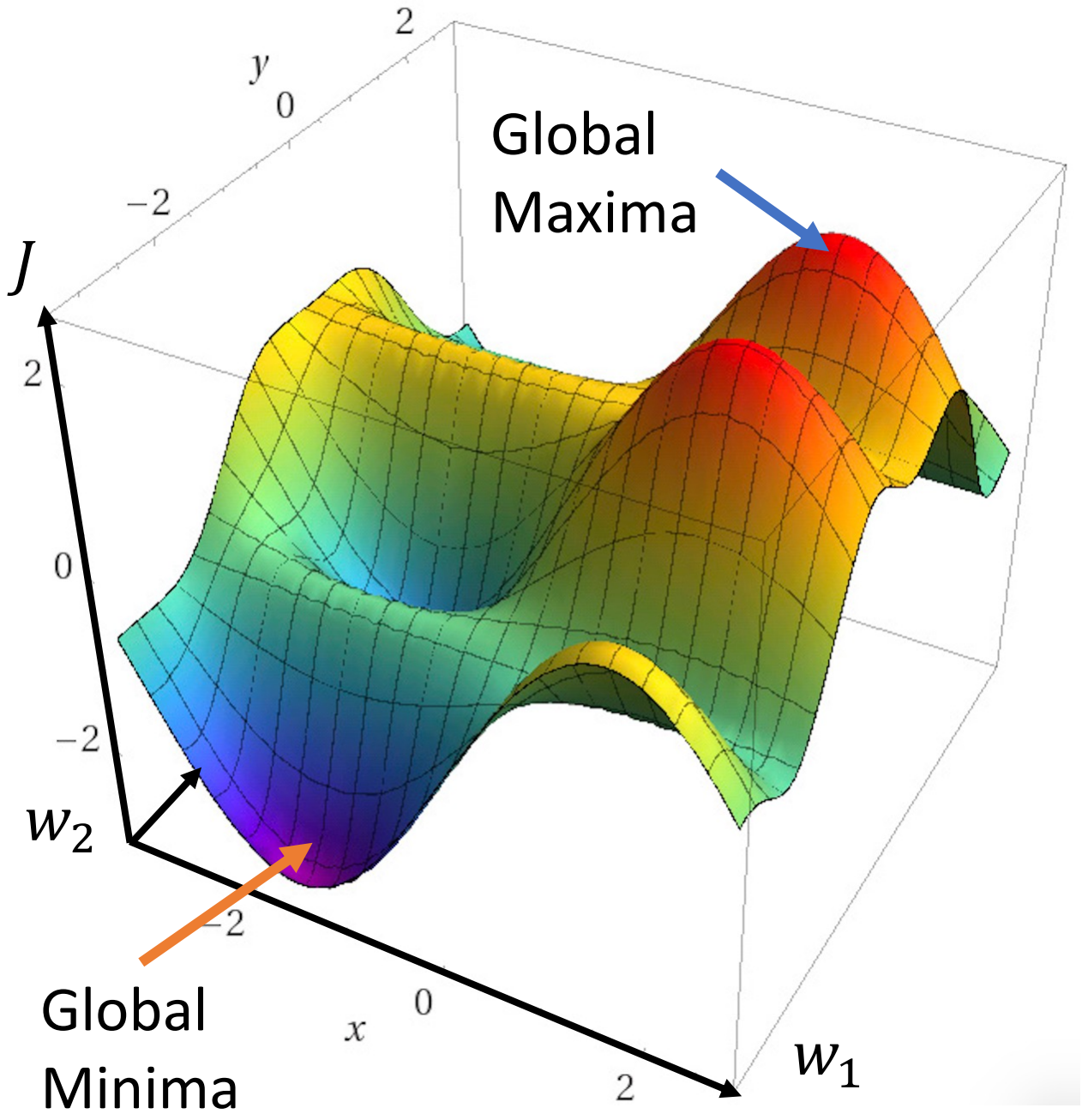


Image Credit: https://www.oreilly.com/radar/the-hard-thing-about-deep-learning/

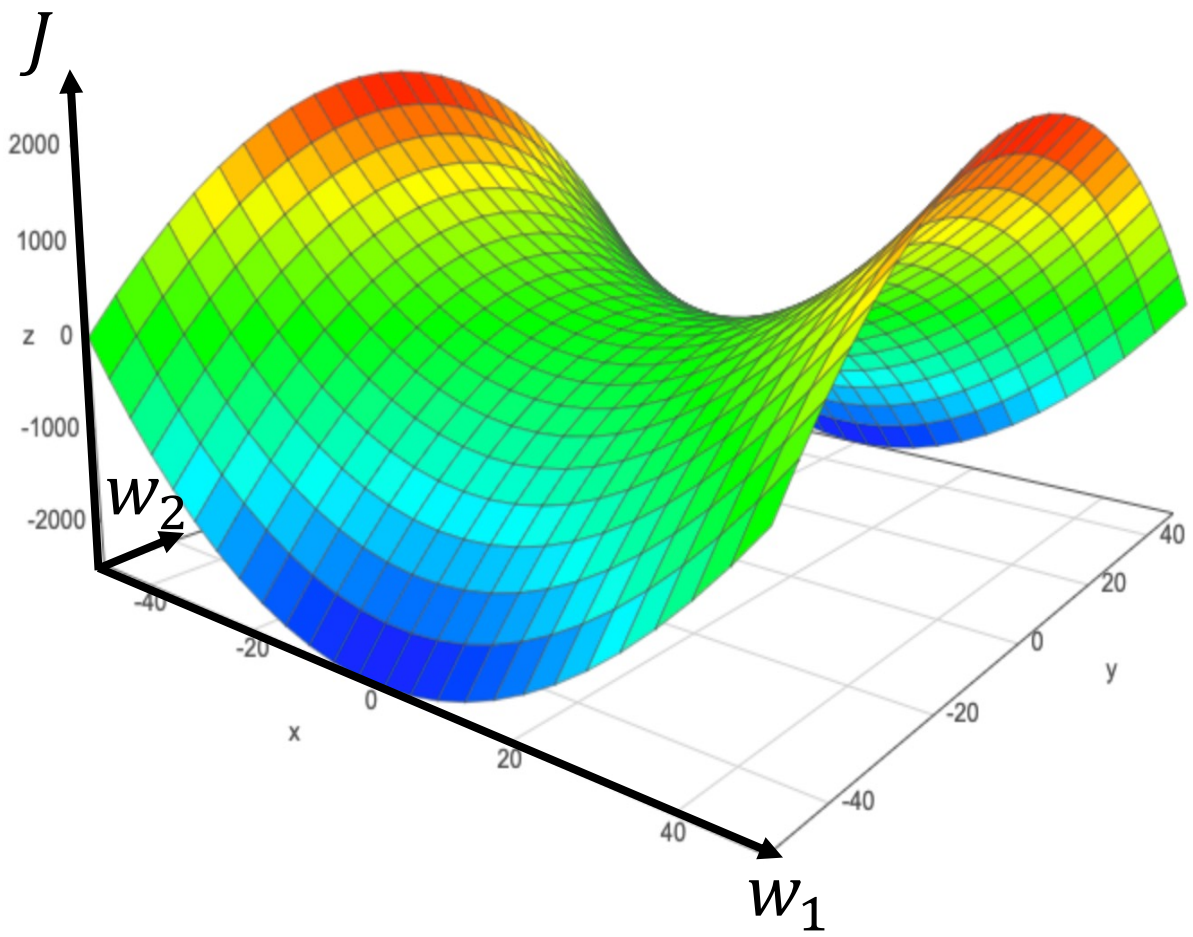


Global Optima

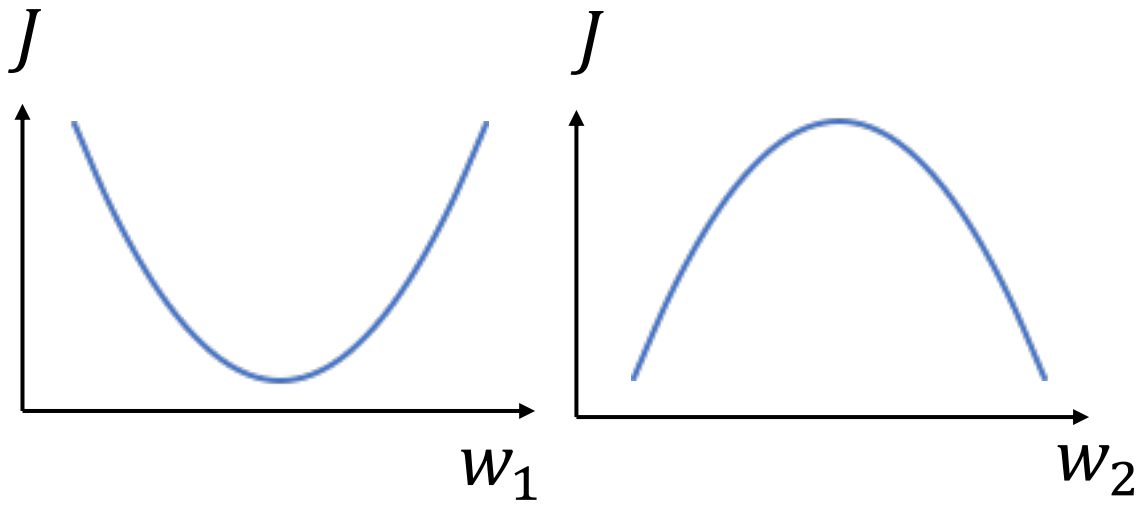
- Optima refers to both maxima and minima
- Global Maxima/Minima refers to the biggest/smallest amongst all your Maxima/Minima
- Local Maxima/Minima refers to all the rest

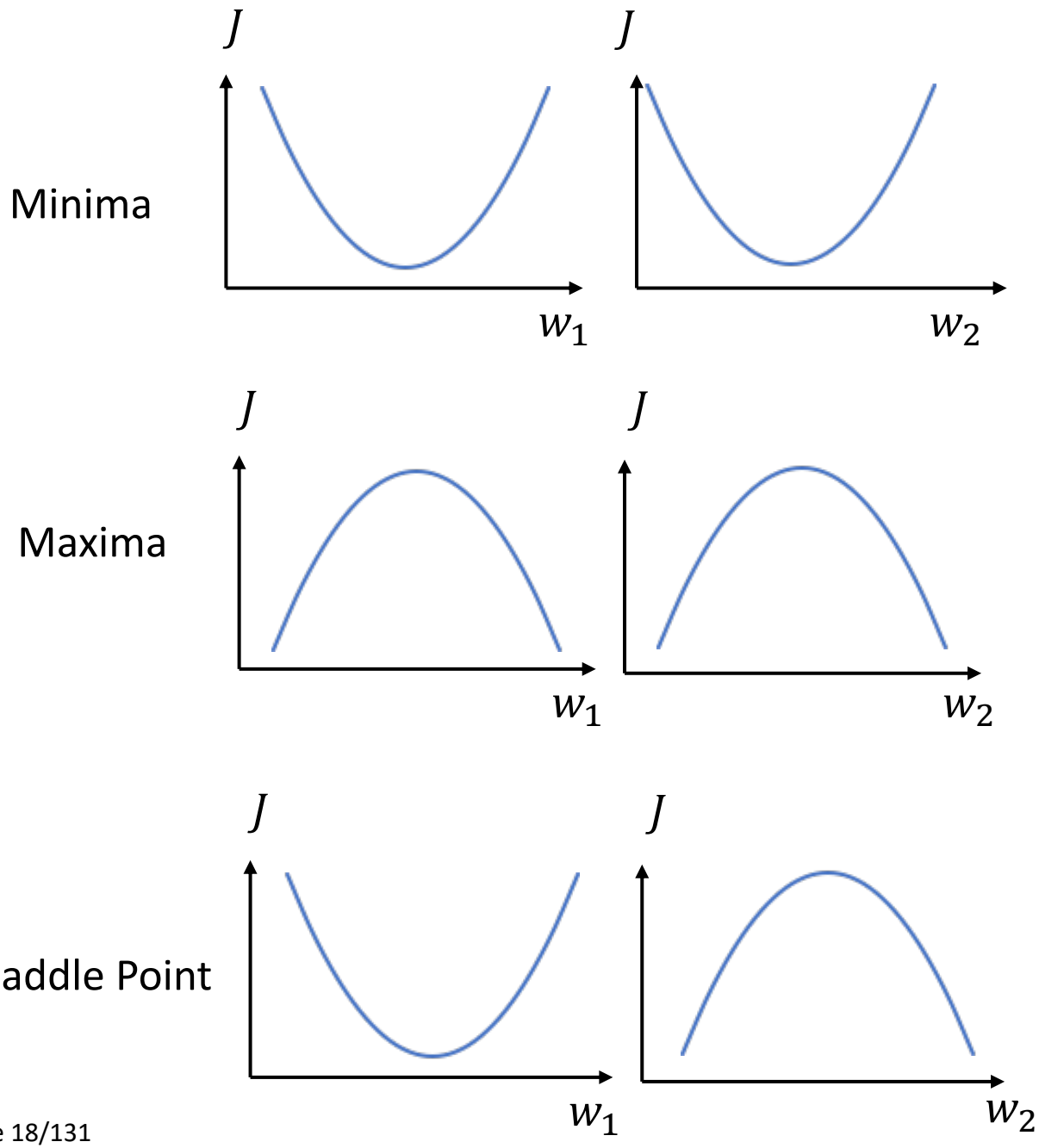
Image Credit: https://www.oreilly.com/radar/the-hard-thing-about-deep-learning/

Saddle Point



- Concave in some inputs
- Convex in others

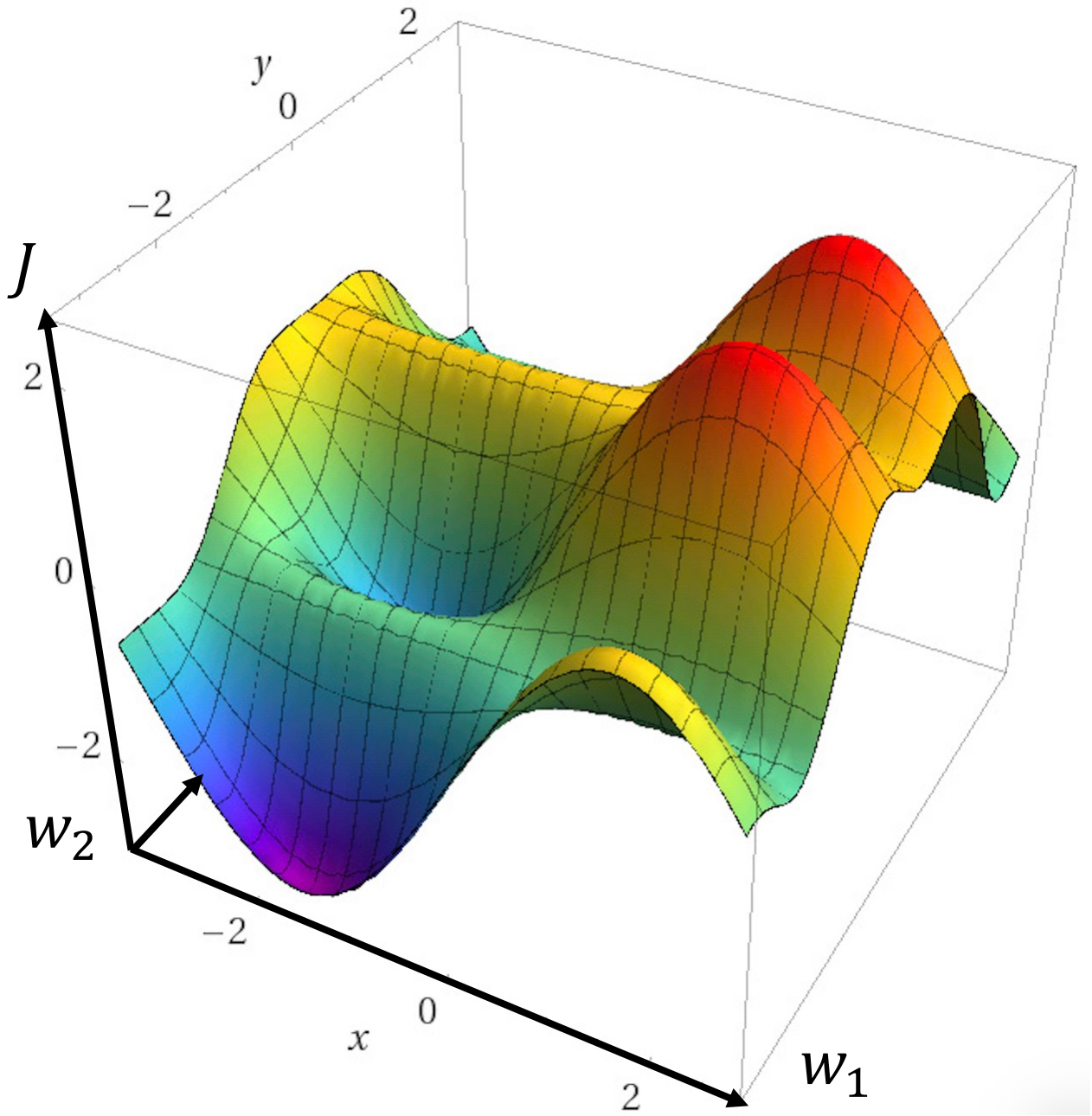




Optima and Saddle Points

Gradients at optima and saddle points are 0

$$\frac{dJ}{dw_i} = 0$$ For all parameters w_i

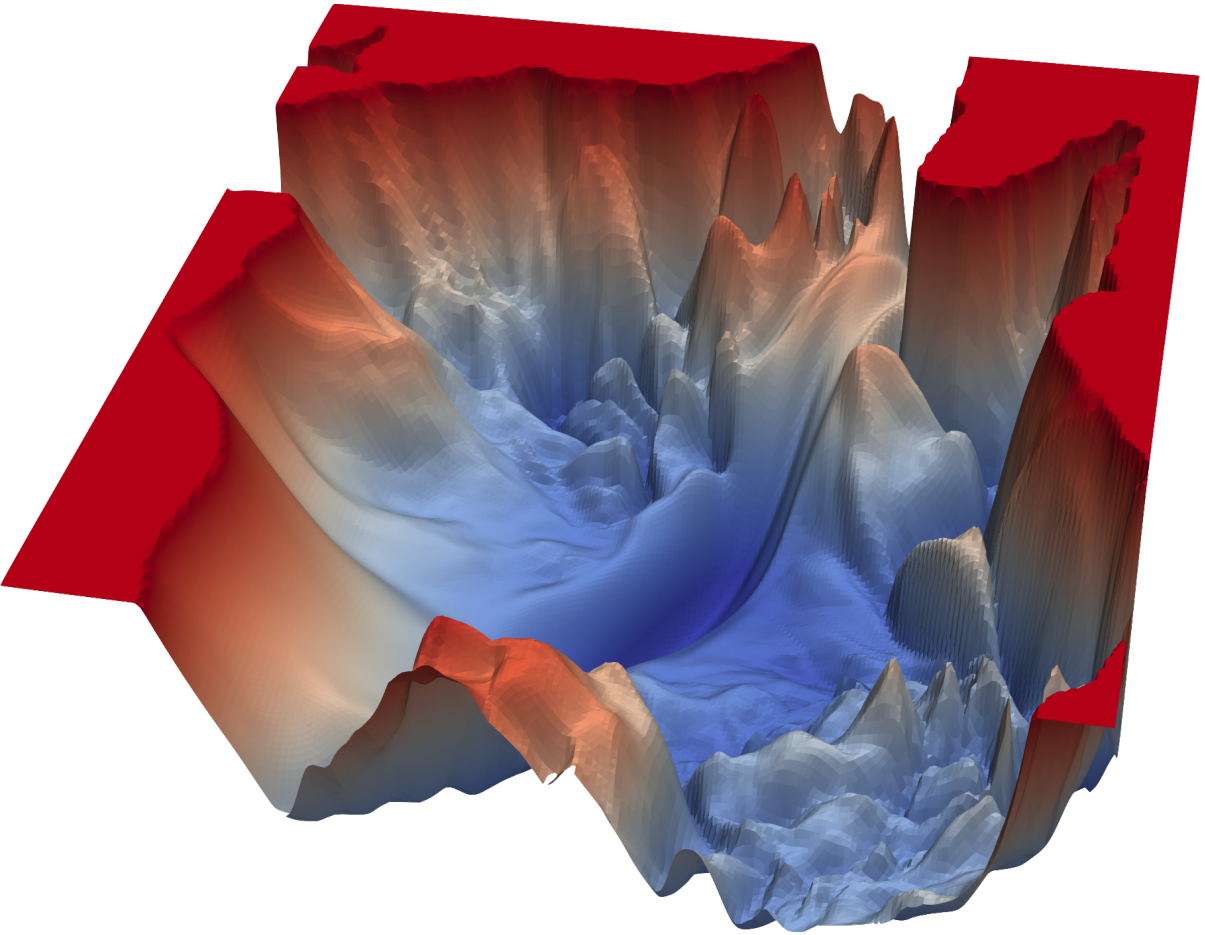


Objective Landscape

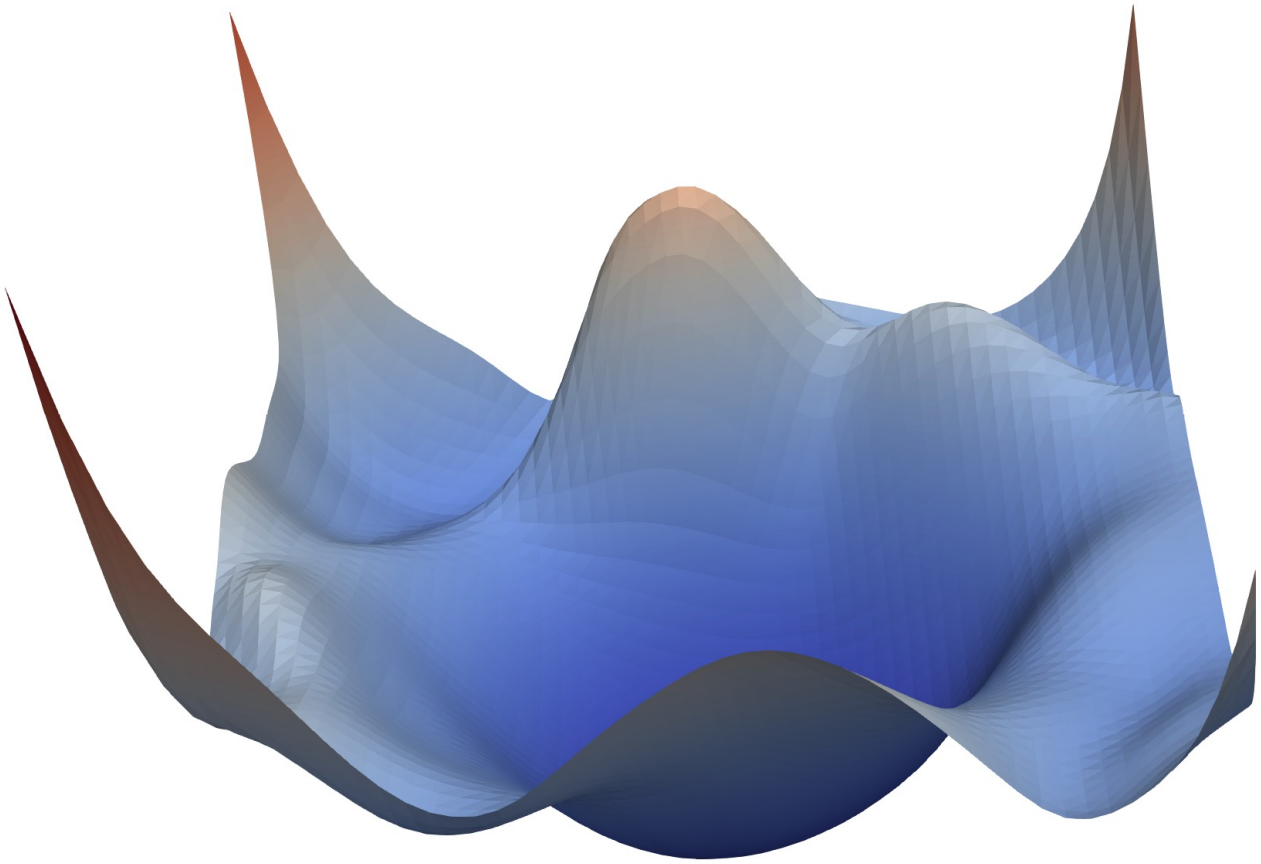
- So far, visualizing two parameter objective landscape
- Modern Deep Neural Networks have millions of parameters!
  → 130 Million for VGGNet
- 130 Million dimensions!
- Each point in this space is one set of parameter values and one potential solution with an associated cost value

Image Credit: https://www.oreilly.com/radar/the-hard-thing-about-deep-learning/

ResNet56 Cost Visualized



Without Skip Connections

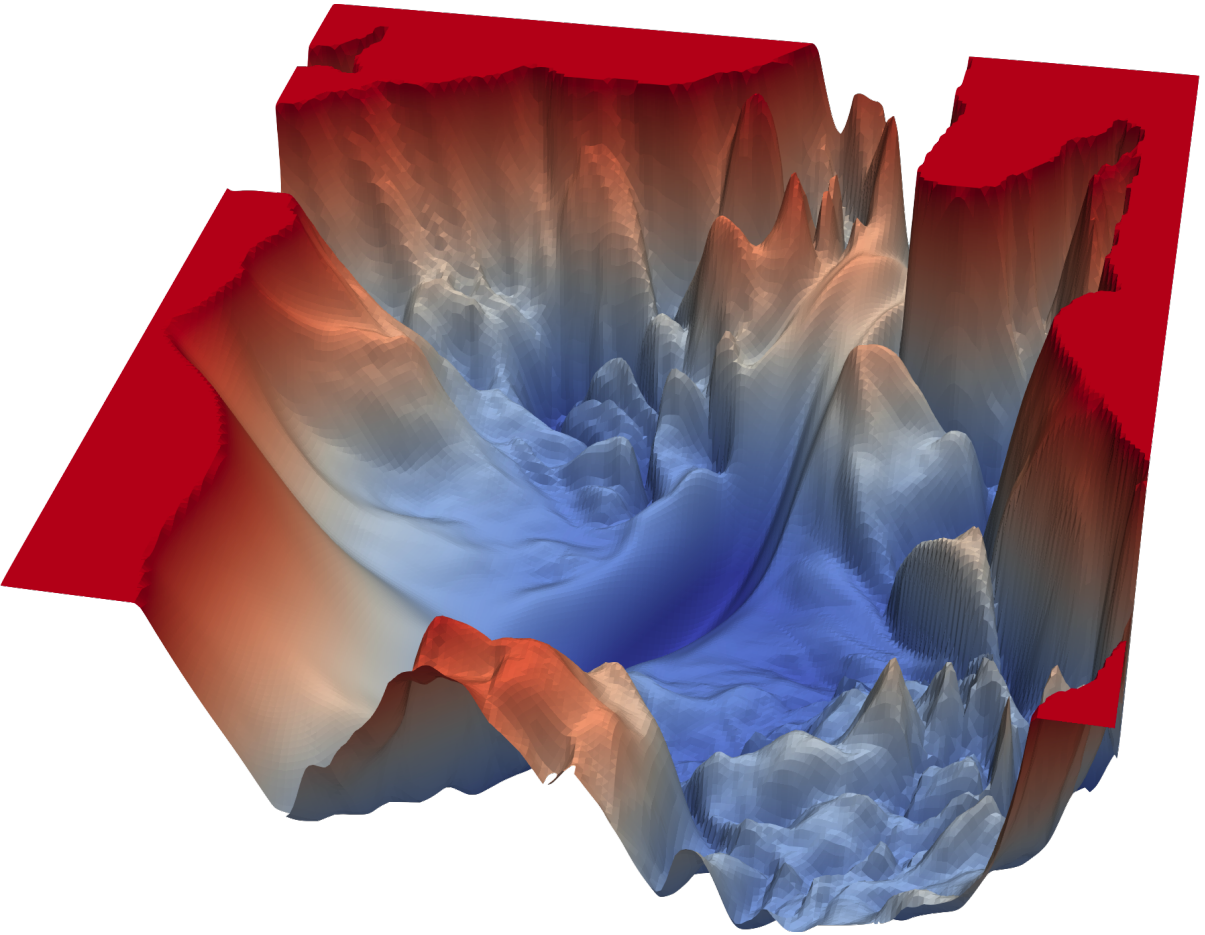


With Skip Connections

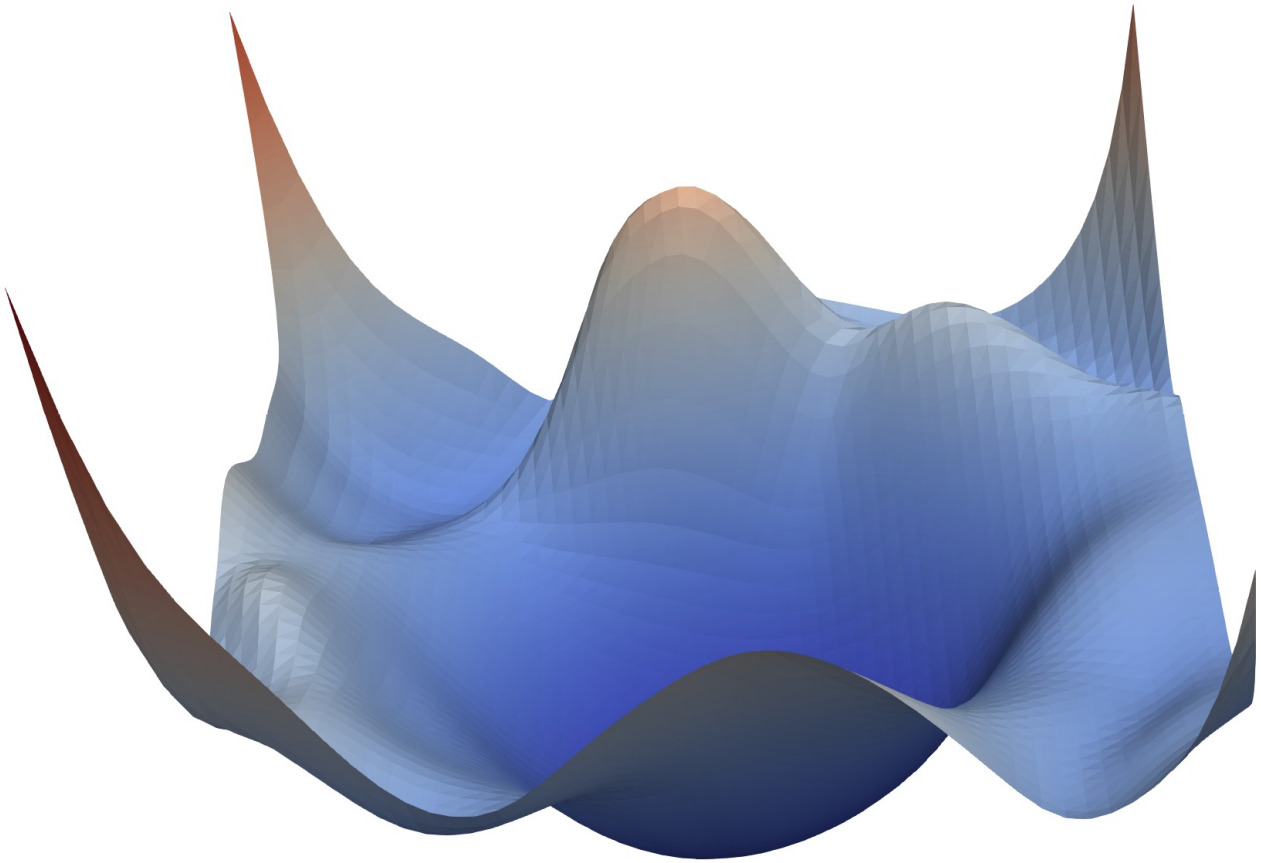
"Visualizing the Loss Landscape of Neural Nets", Li et al., 2017, https://arxiv.org/abs/1712.09913
https://www.cs.umd.edu/~tomg/projects/landscapes/

ResNet56 Cost Visualized

$$J = f(y, x, w, b, \boldsymbol{arch})$$



Without Skip Connections

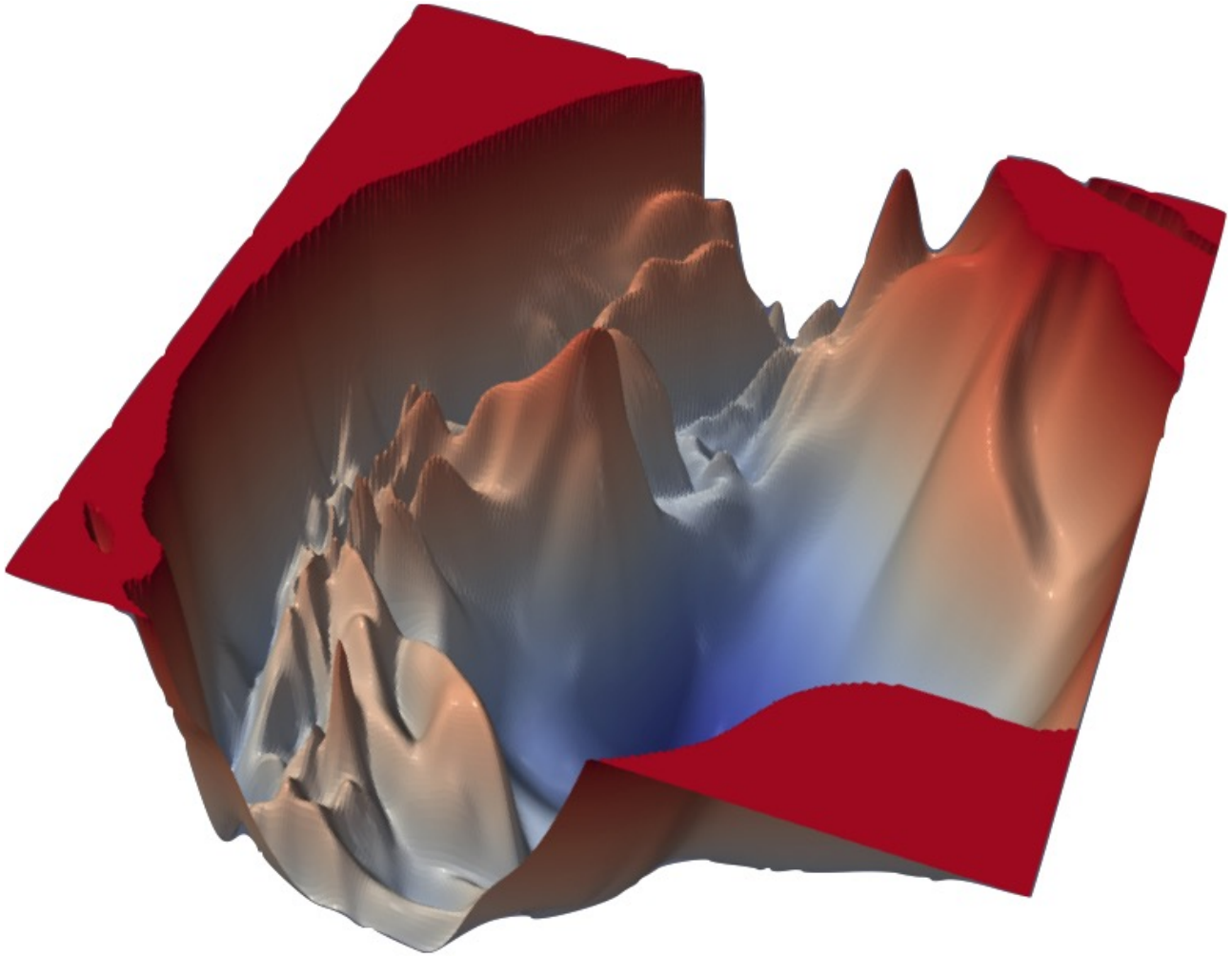


With Skip Connections

"Visualizing the Loss Landscape of Neural Nets", Li et al., 2017, https://arxiv.org/abs/1712.09913
https://www.cs.umd.edu/~tomg/projects/landscapes/

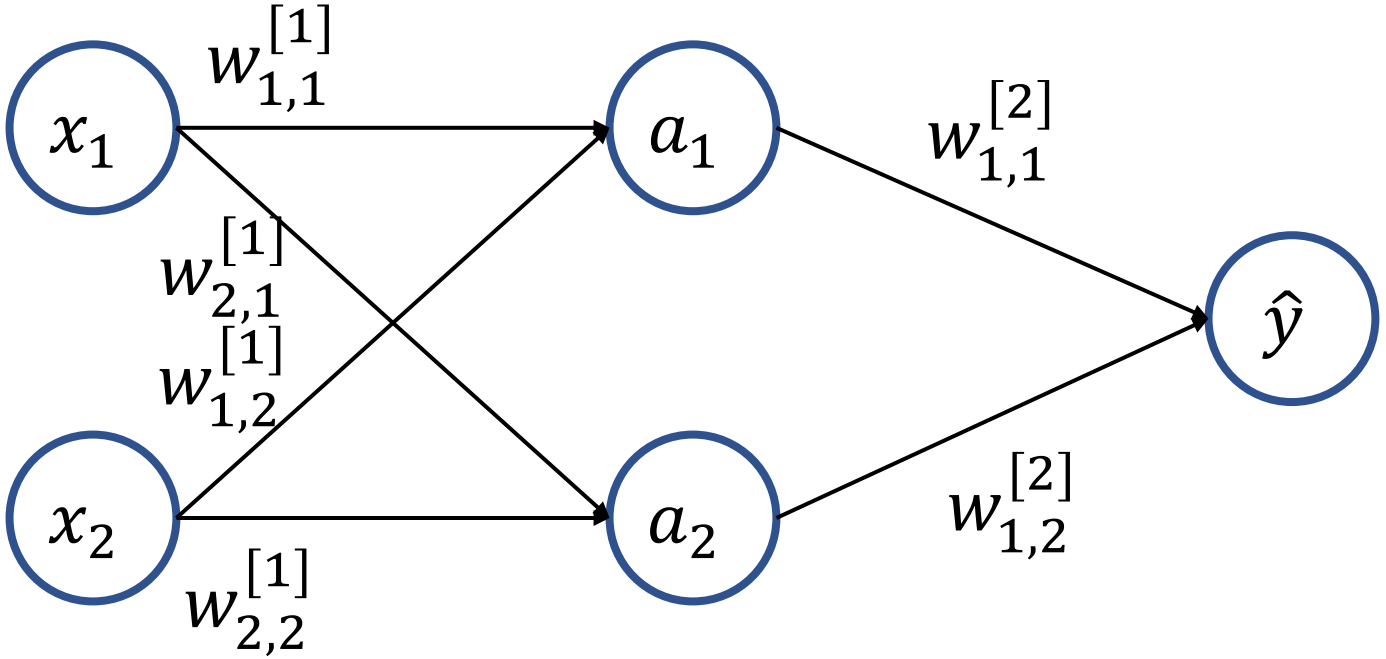
DenseNet Cost Visualized

$$J = f(y, x, w, b, \boldsymbol{arch})$$



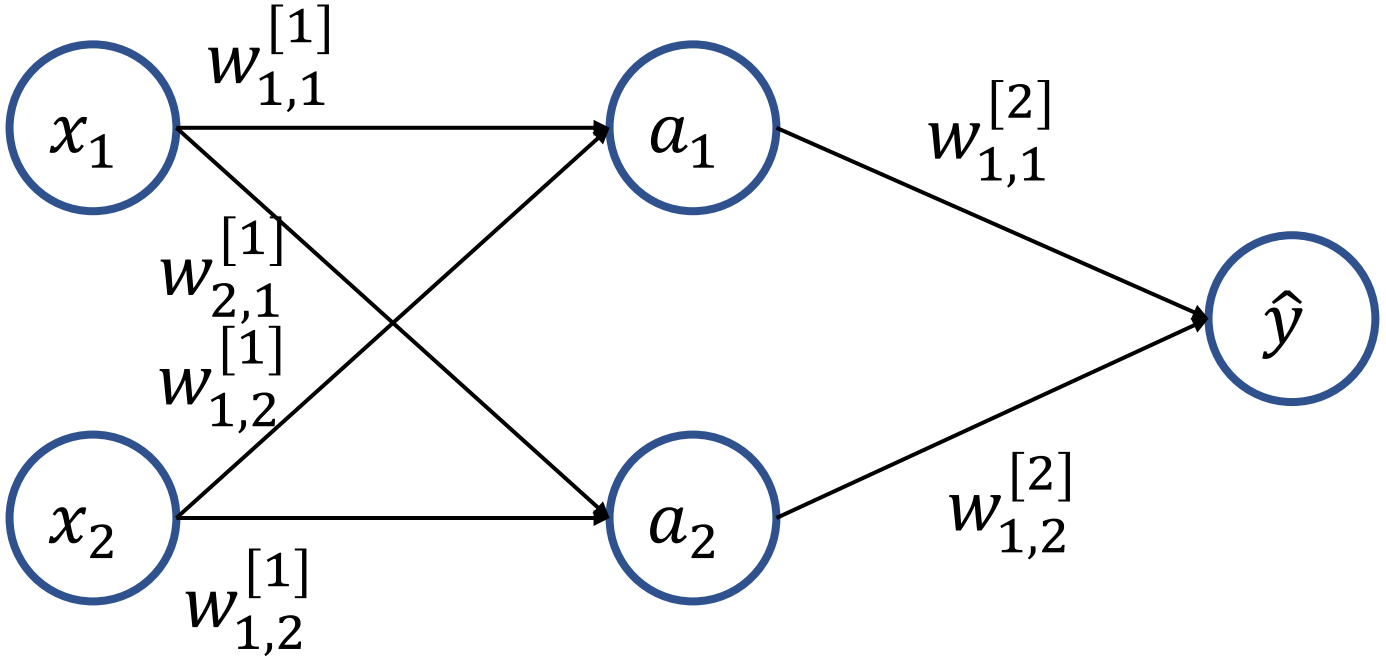
"Visualizing the Loss Landscape of Neural Nets", Li et al., 2017, https://arxiv.org/abs/1712.09913
https://www.cs.umd.edu/~tomg/projects/landscapes/

Deep Learning Cost Function is not Convex
(i.e. there are many equal Global Minima)



$$\hat{y} = g\left(w_{1,1}^{[2]} g\left(w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_1^{[1]}\right) + w_{1,2}^{[2]} g\left(w_{2,1}^{[1]} x_1 + w_{2,2}^{[1]} x_2 + b_2^{[1]}\right) + b_1^{[2]}\right)$$

Deep Learning Cost Function is not Convex
(i.e. there are many equal Global Minima)



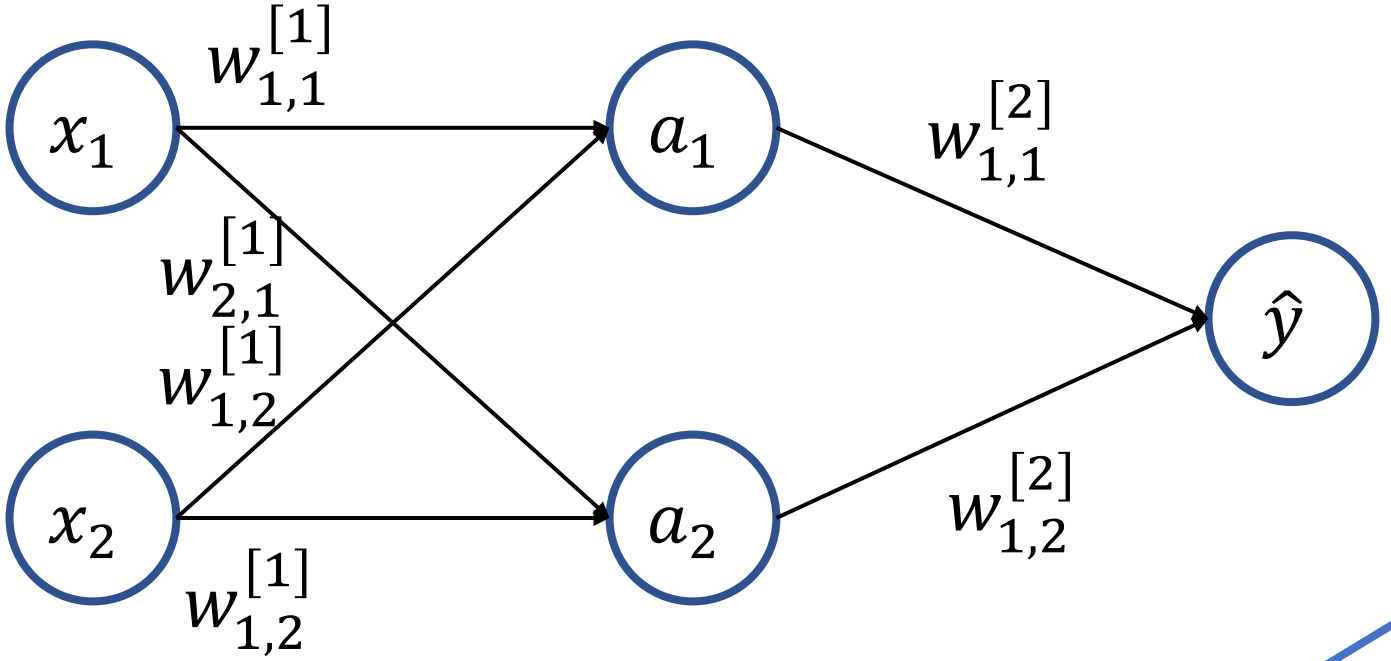
Say this set of parameter values leads to the smallest cost

$$\hat{y} = g\left(w_{1,1}^{[2]} g\left(w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_1^{[1]}\right) + w_{1,2}^{[2]} g\left(w_{2,1}^{[1]} x_1 + w_{2,2}^{[1]} x_2 + b_2^{[1]}\right) + b_1^{[2]}\right)$$

1 3 8 6 2 1 4 7 9

Brad Quinton, Scott Chin

Deep Learning Cost Function is not Convex
(i.e. there are many equal Global Minima)

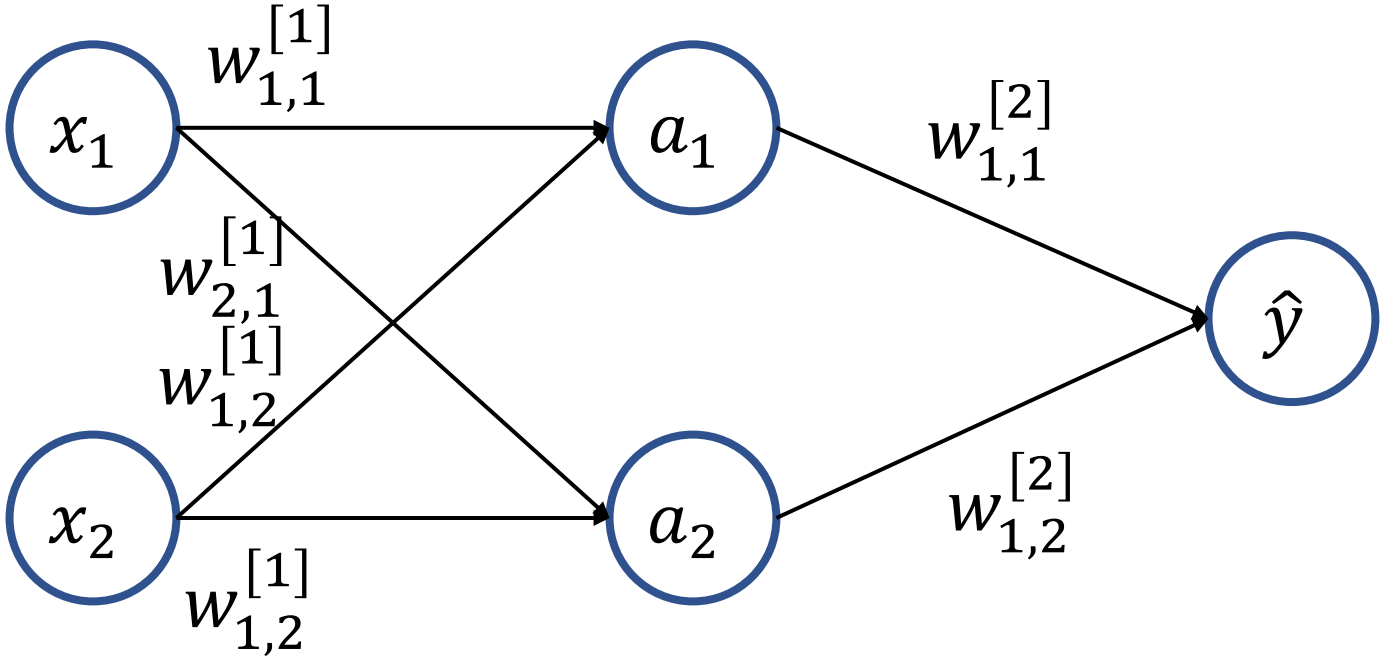


Swapping these two sets of values is mathematically equivalent

$$\hat{y} = g\left(w_{1,1}^{[2]} g\left(w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_1^{[1]}\right) + w_{1,2}^{[2]} g\left(w_{2,1}^{[1]} x_1 + w_{2,2}^{[1]} x_2 + b_2^{[1]}\right) + b_1^{[2]}\right)$$

1	3	8	6	2	1	4	7	9

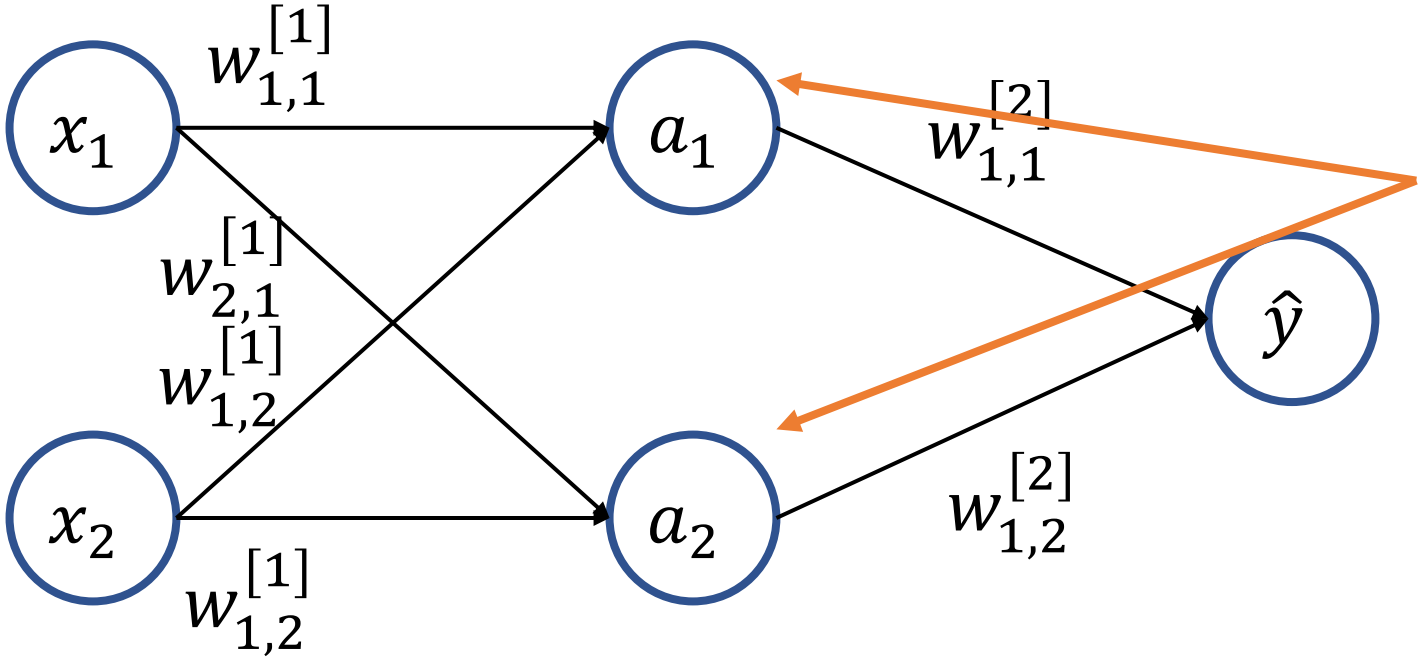
Deep Learning Cost Function is not Convex
(i.e. there are many equal Global Minima)



$$\hat{y} = g\left(w_{1,1}^{[2]} g\left(w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_1^{[1]}\right) + w_{1,2}^{[2]} g\left(w_{2,1}^{[1]} x_1 + w_{2,2}^{[1]} x_2 + b_2^{[1]}\right) + b_1^{[2]}\right)$$

1	3	8	6		2	1	4	7	9
2	1	4	7		1	3	8	6	9

Deep Learning Cost Function is not Convex
(i.e. there are many equal Global Minima)



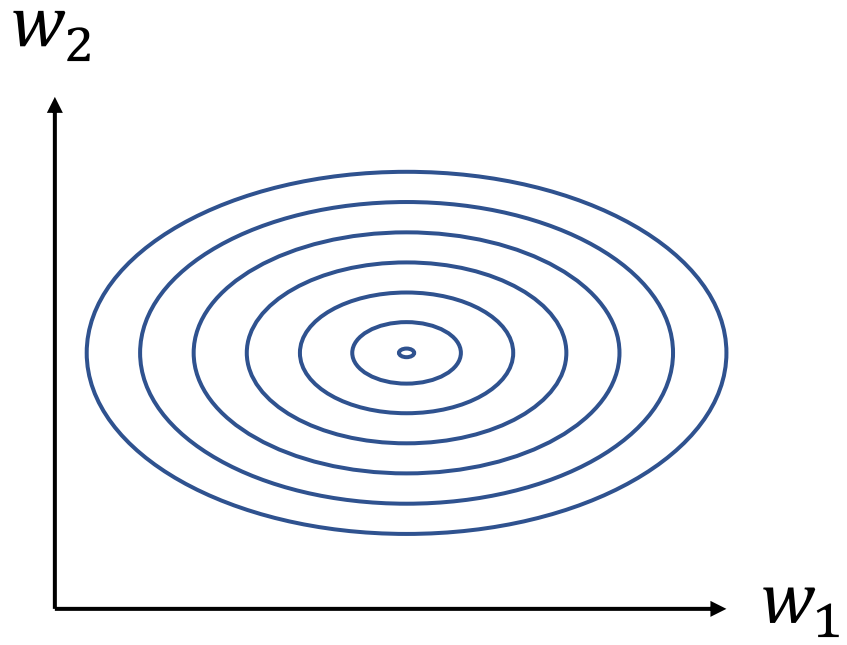
Swapping function of these two hidden units

$$\hat{y} = g\left(w_{1,1}^{[2]} g\left(w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_1^{[1]}\right) + w_{1,2}^{[2]} g\left(w_{2,1}^{[1]} x_1 + w_{2,2}^{[1]} x_2 + b_2^{[1]}\right) + b_1^{[2]}\right)$$

1	3	8	6		2	1	4	7	9
2	1	4	7		1	3	8	6	9

Comment on Dimensionality

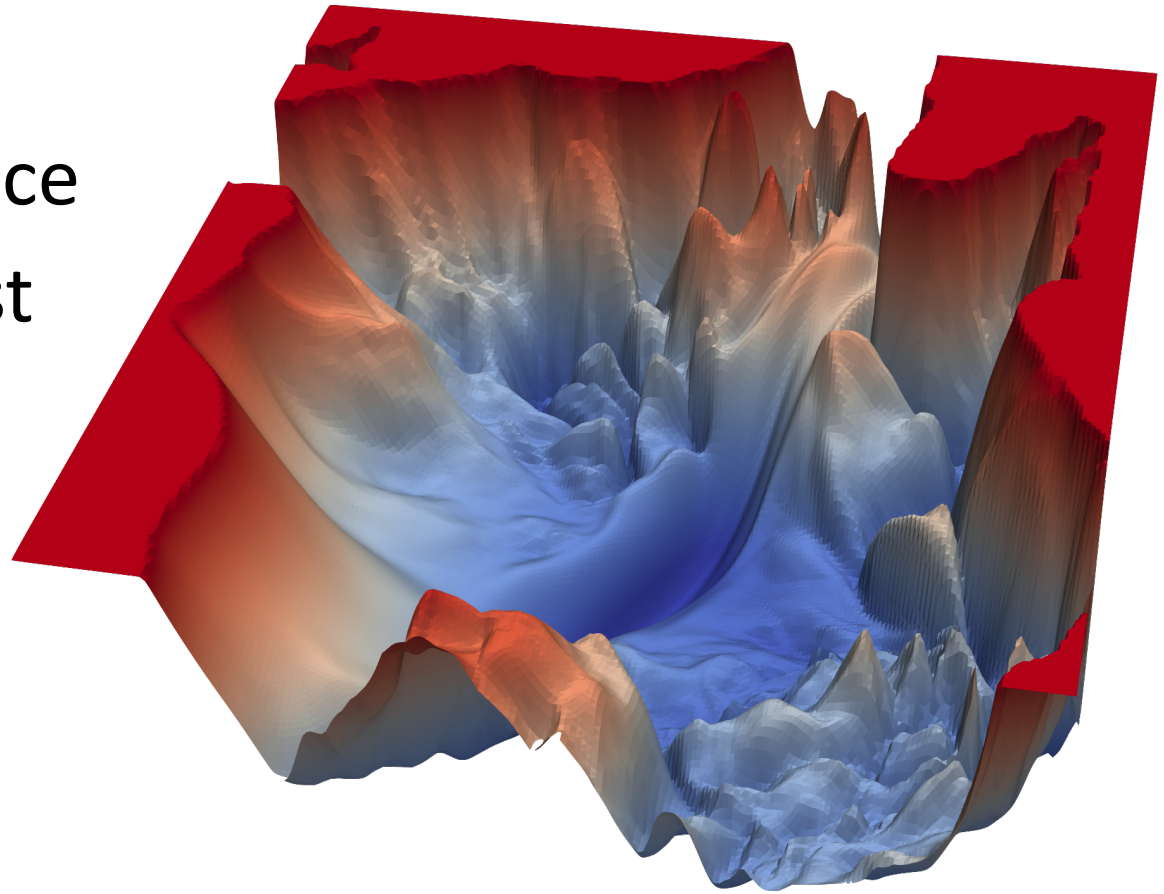
- Solution space is very high-dimensional, hard to visualize
- Will discuss topics in terms of 2D space but the concepts extend to higher dimensions



Gradient Descent (Review)

Intuition:

- Start somewhere in parameter space
- Move in direction with the steepest decrease in Cost.
- Repeat.

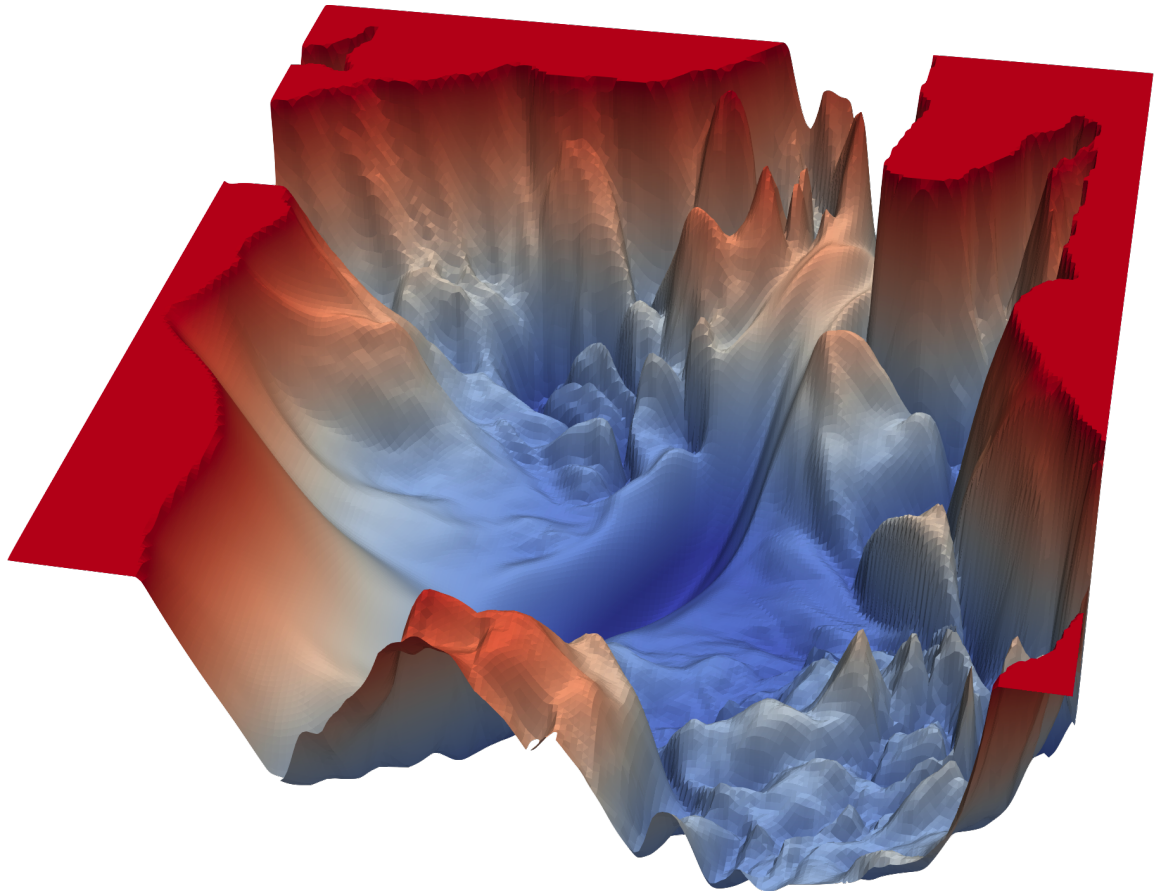


"Visualizing the Loss Landscape of Neural Nets", Li et al., 2017, https://arxiv.org/abs/1712.09913

Gradient Descent (Review)

Analogy:

- You are in a hilly landscape
- You want to find the lowest point in this hilly landscape
- You are blindfolded but you can feel the slope of the ground directly beneath you with your feet
- Take a step in direction with the largest downhill slope
- Repeat.

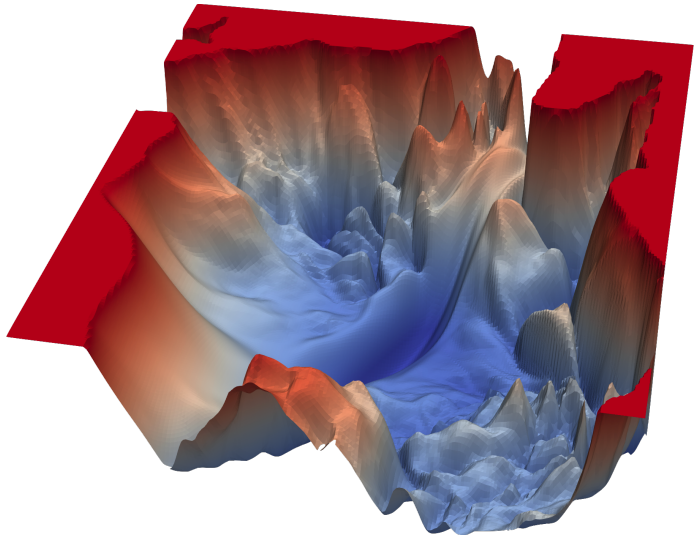


"Visualizing the Loss Landscape of Neural Nets", Li et al., 2017, https://arxiv.org/abs/1712.09913

Gradient Descent (Review)

```
w = initialize()
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  w = w - learning_rate*dJ_dw
```

- Hyperparameters:
  - Parameter Initialization Method
  - Learning Rate
  - Number of iterations

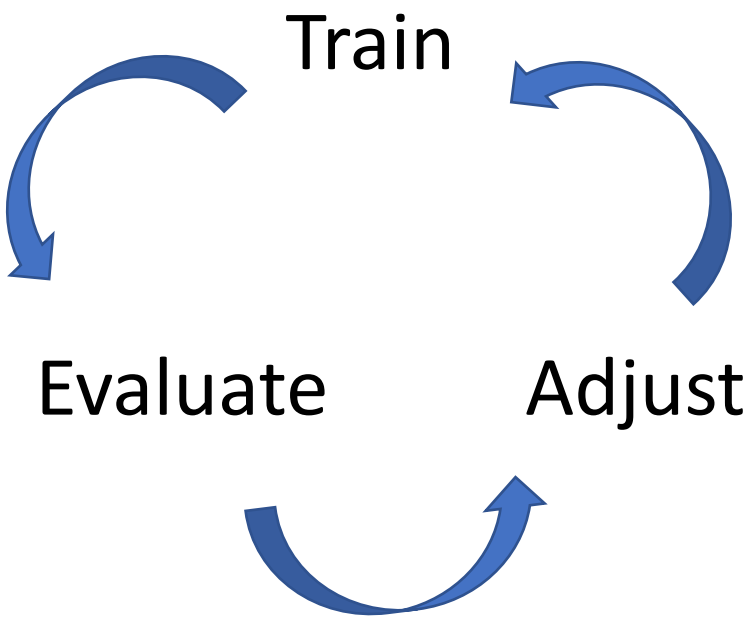


"Visualizing the Loss Landscape of Neural Nets", Li et al., 2017, https://arxiv.org/abs/1712.09913

Why Improve on Gradient Descent?

- Training a good model is a very iterative process.
  - Train model
  - Evaluate
  - Change a hyperparameter
  - Repeat

- Faster you can go through this loop, the more you can tune model
- The longest part is the training



Problem with Gradient Descent

- Cost is a function of all training images
- When training set size get's large (e.g. ILSVRC Imagenet set has 1.2million images), computational requirements make classic gradient descent impractical:

1. Takes too long to compute gradient for one training iteration
2. Requires too much memory to store activations (intermediate volumes) of all samples concurrently in GPU memory

Problem with Gradient Descent

- Cost is a function of all training images
- When training set size get's large (e.g. ILSVRC Imagenet set has 1.2million images), computational requirements make classic gradient descent impractical:

1. Takes too long to compute gradient for one training iteration
2. Requires too much memory to store activations (intermediate volumes) of all samples concurrently in GPU memory

What can we do?

Mini-Batch Gradient Descent

- Use a small subset of your training set (a mini-batch) as an approximation of the overall training set

Mini-Batch Gradient Descent

- Use a small subset of your training set (a mini-batch) as an approximation of the overall training set

Intuition

- View your training set itself as a sampling of some unknown underlying function. It is this function that your model is trying to learn!
- Under this view, a subset of your full training set (i.e. a mini-batch) is still a sampling of the same function
- However, more nuances of the underlying function can be seen in larger samplings

Mini-Batch Gradient Descent

```
w = initialize()
for i in range(num_iterations):
  batch = sample(train_data, batch_size)
  dJ_dw = compute_gradients(batch, cost_func, w)
  w = w - learning_rate*dJ_dw
```

Mini-Batch Gradient Descent

```
w = initialize()
for i in range(num_iterations):
  batch = sample(train_data, batch_size)
  dJ_dw = compute_gradients(batch, cost_func, w)
  w = w - learning_rate*dJ_dw
```

Hyperparameters:

- Parameter Initialization Method
- Learning Rate
- Number of iterations
- Sampling method
- Batch size – 32/64/128/256

Mini-Batch Gradient Descent - Sampling

A common way to sample:

- random shuffle full set
- partition into mini-batches
- iterate across each mini-batch

One full pass through the set is called an epoch

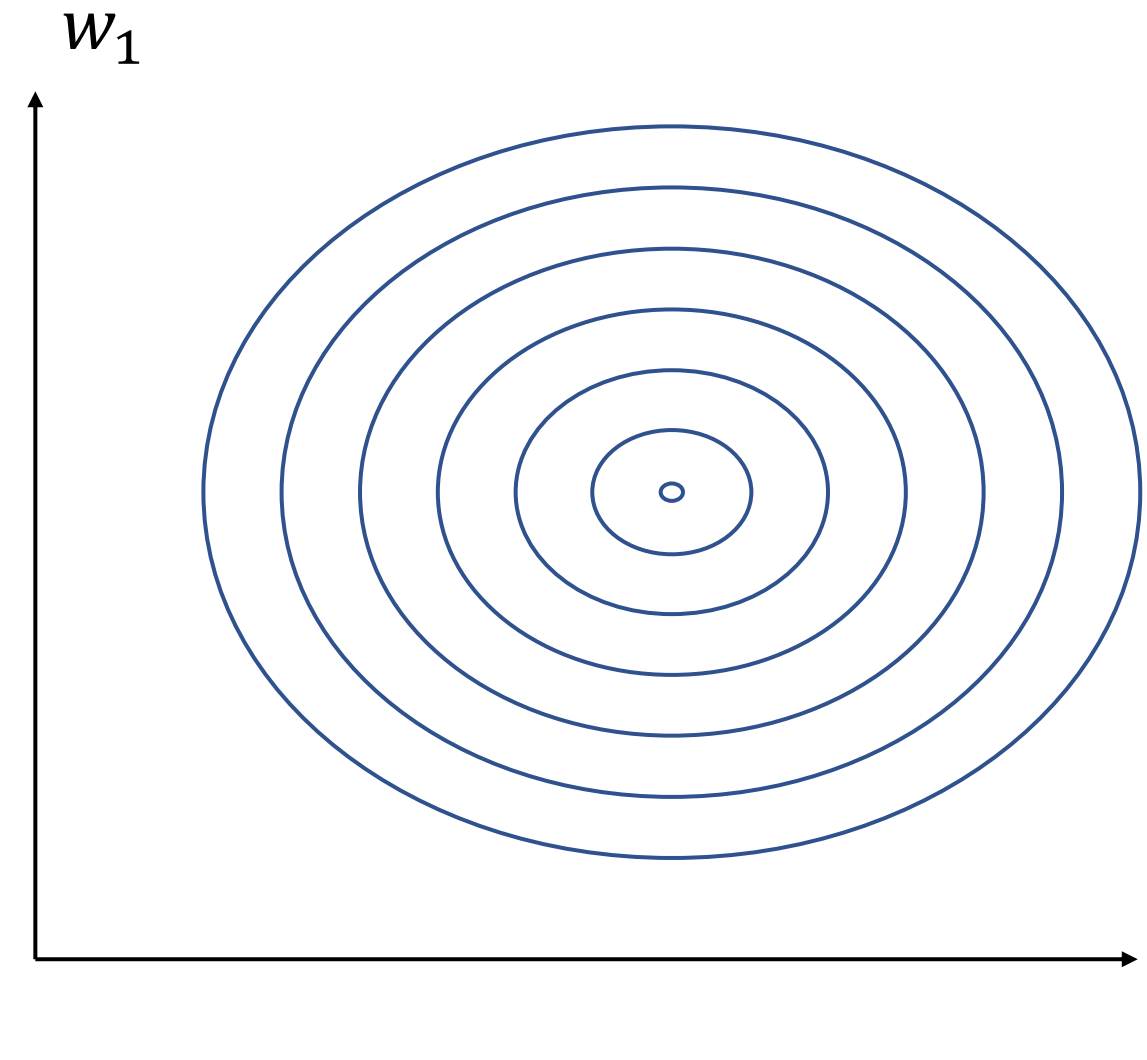
Mini-Batch Gradient Descent - Sampling

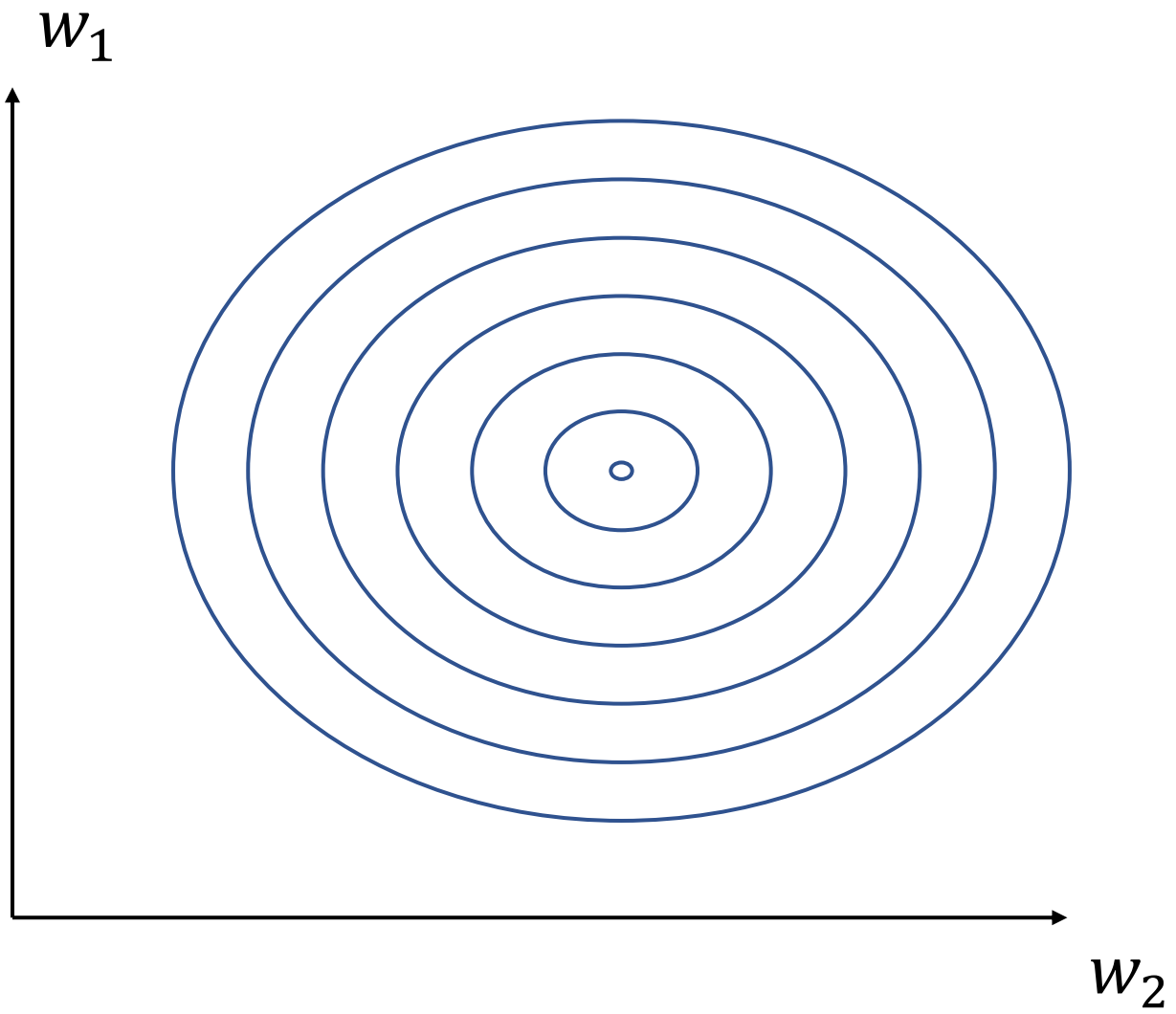
```
w = initialize()
for i in range(num_epochs):
  for batch_i in m/batch_size:
    batch = train_data[batch_i*batch_size:
                       (batch_i+1)*batch_size)]
    dJ_dw = compute_gradients(batch, cost_func, w)
    w = w - learning_rate*dJ_dw
  train_data = random_shuffle()
```

Mini-Batch Gradient Descent - Sampling

- If minibatch size = full set → same as classic gradient descent
- If minibatch size = 1 → each sample is a mini-batch
  - This is called Stochastic Gradient Descent (SGD)
  - Lose benefits from vectorization
- Usually 32/64/128/256
  - Power of 2 because sometimes memory access works out better
- Pick as big as you can and still fit into GPU memory
  - Significant performance hit from memory access if can't fit into memory

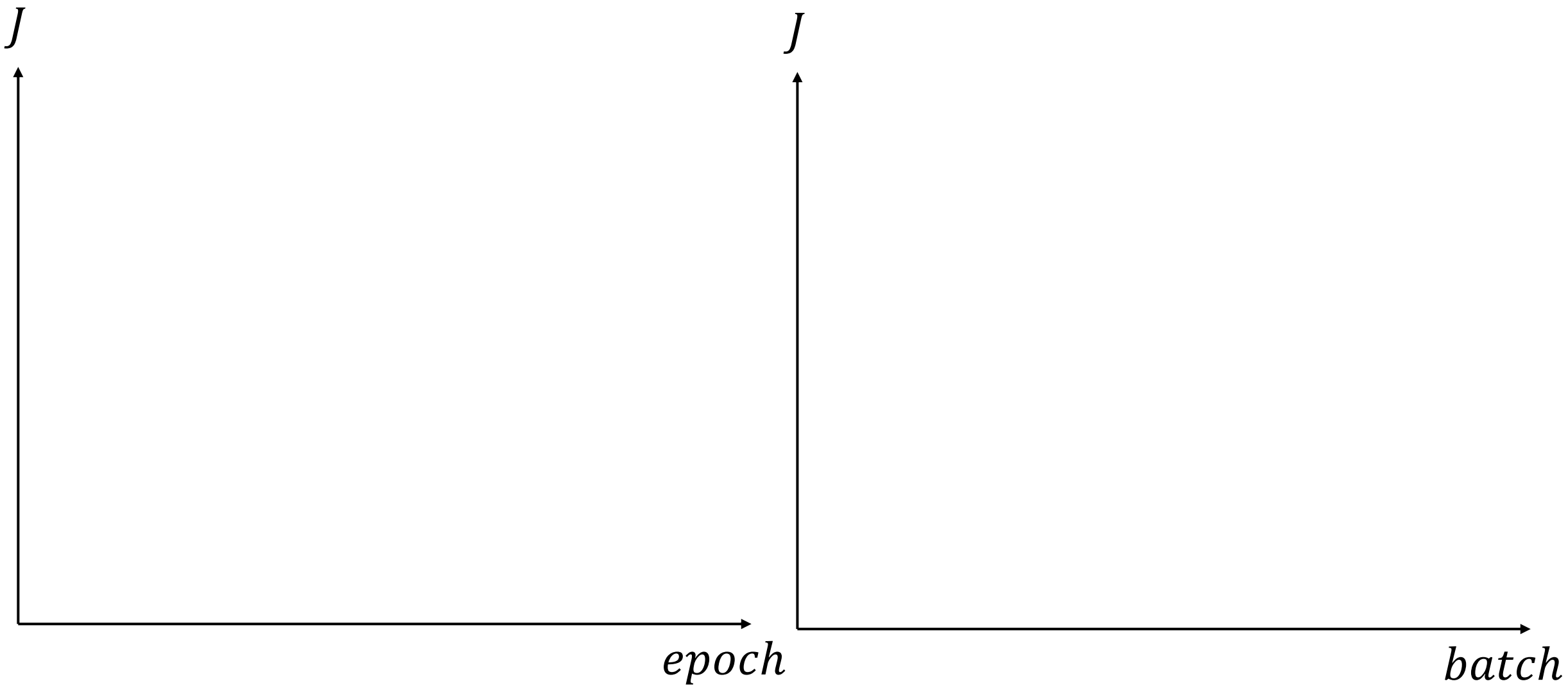
Plain vs Mini-Batch Gradient Descent





More steps per epoch. E.g. MNIST 60k images, batch size=128 → 470 updates per epoch

Plain vs Mini-Batch Gradient Descent



Aside: Comment on Terminology

- Using a mini-batch size of 1 is technically called Stochastic Gradient Descent (SGD)
- Common to see people use SGD to refer to Mini-Batch Gradient Descent (e.g. Keras does this)

Problem with Gradient Descent SOLVED

- Cost is a function of all training images
- When training set size get's large (e.g. ILSVRC Imagenet set has 1.2million images), computational requirements make gradient descent impractical:

1. Takes too long to compute gradient for one training iteration
2. Requires too much memory to store activations (intermediate volumes) of all samples concurrently in GPU memory

What can we do?

Still some Problems

Problems in both classic and mini-batch gradient descent

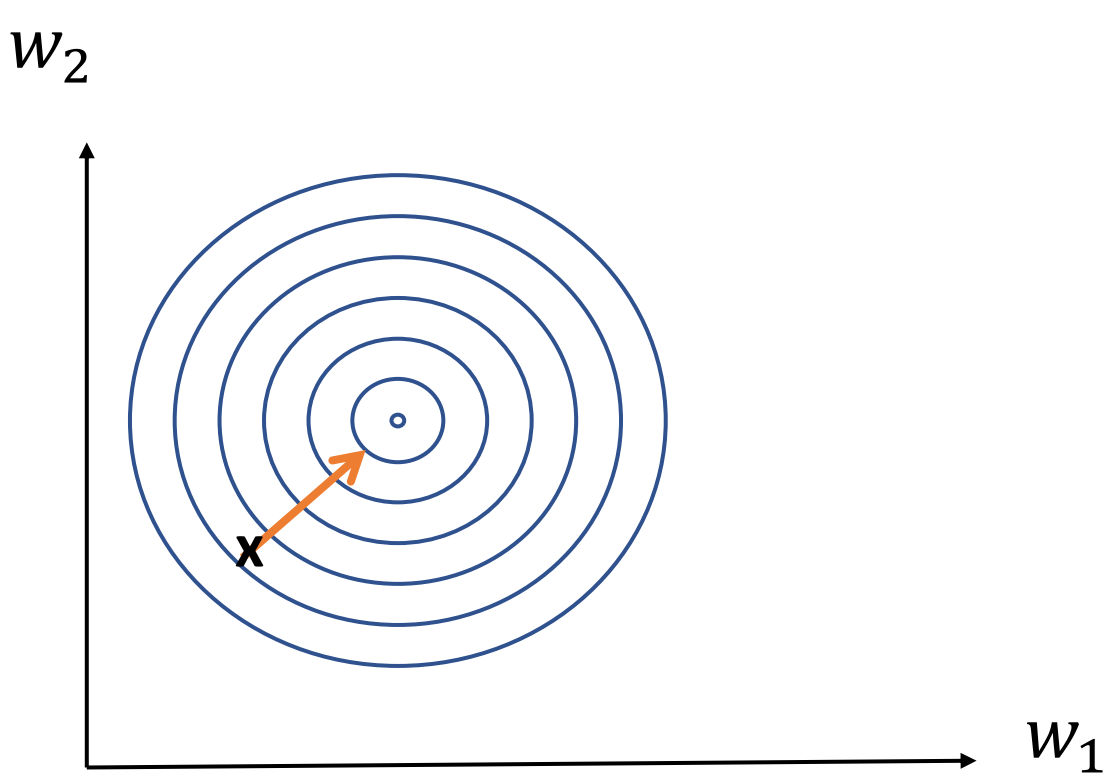
1. Different dimensions (parameters) may change at different rates
2. Local optima and saddle points

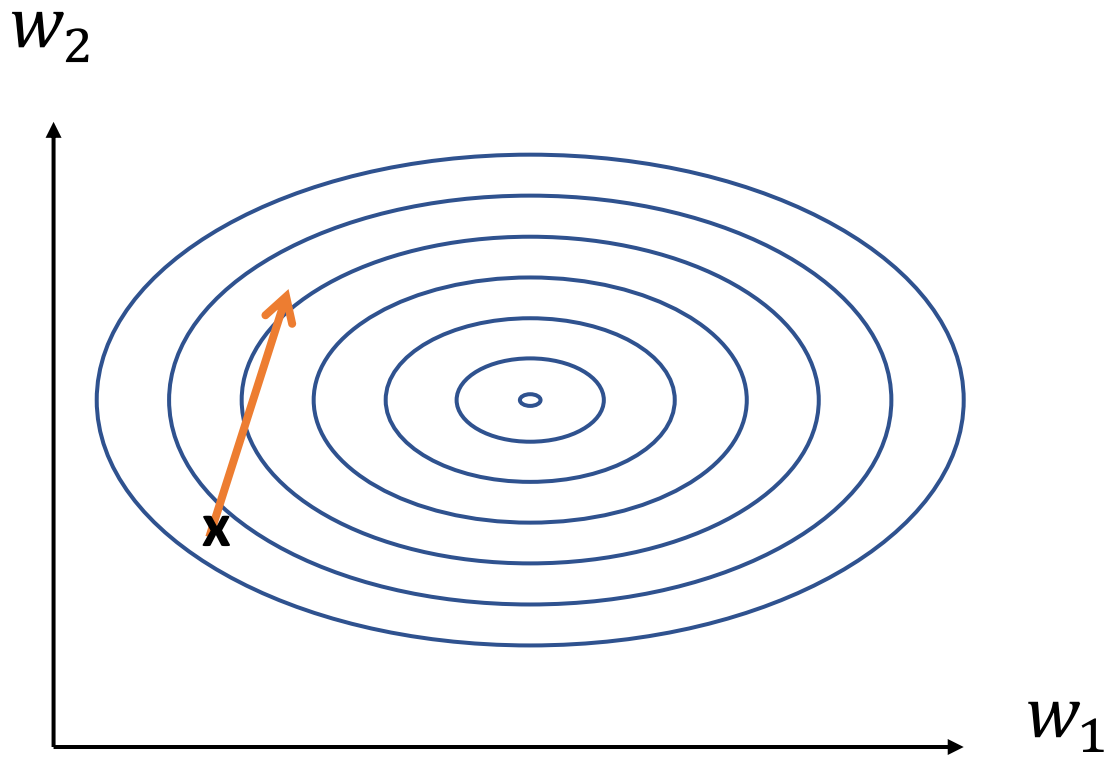
Problem only in mini-batch gradient descent

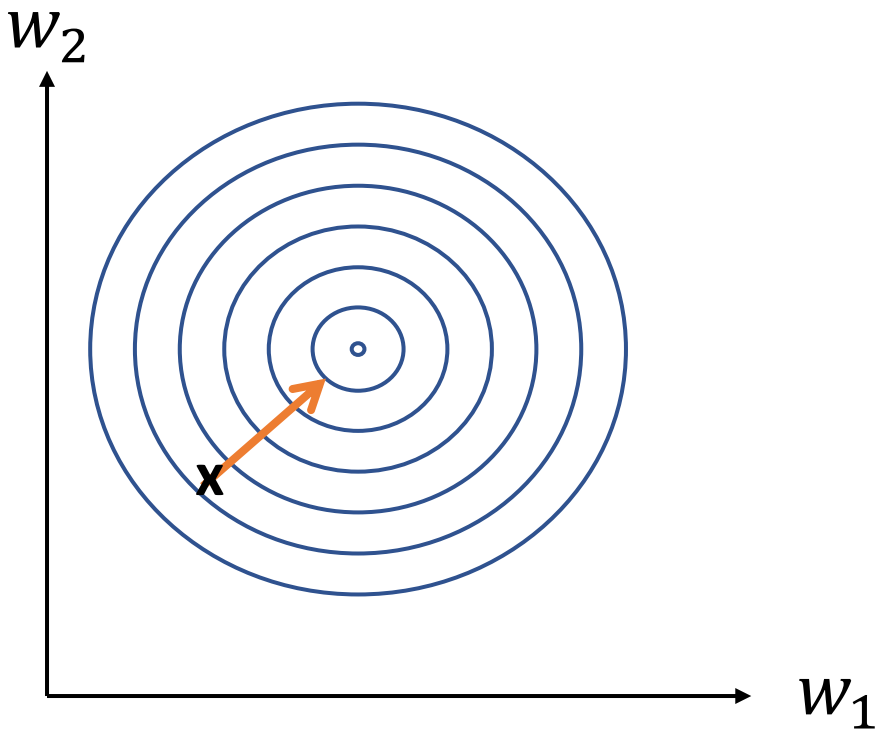
3. Meandering nature of Mini-Batch Gradient Descent

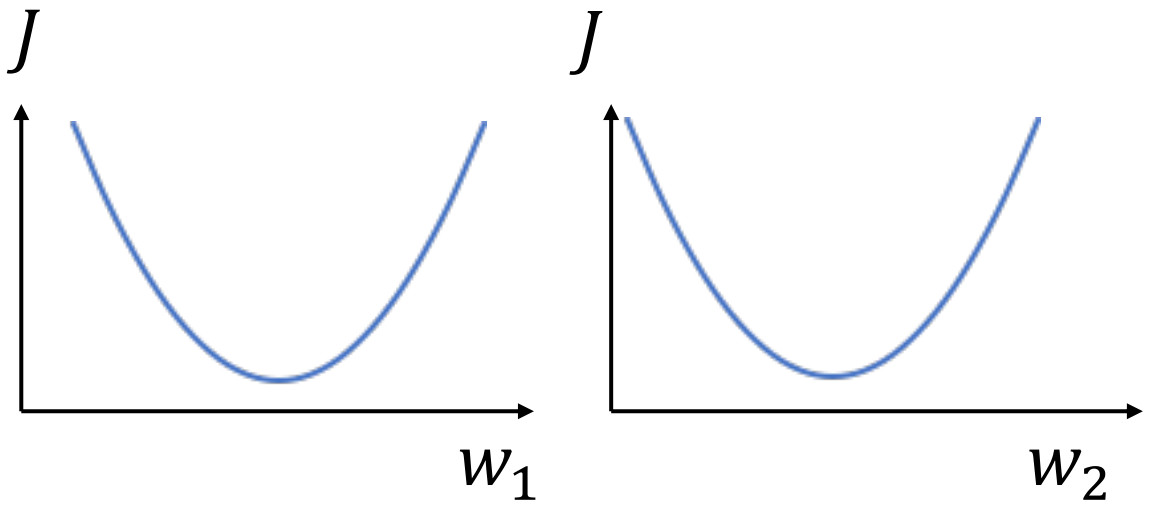
1. Different dimensions change at different rates

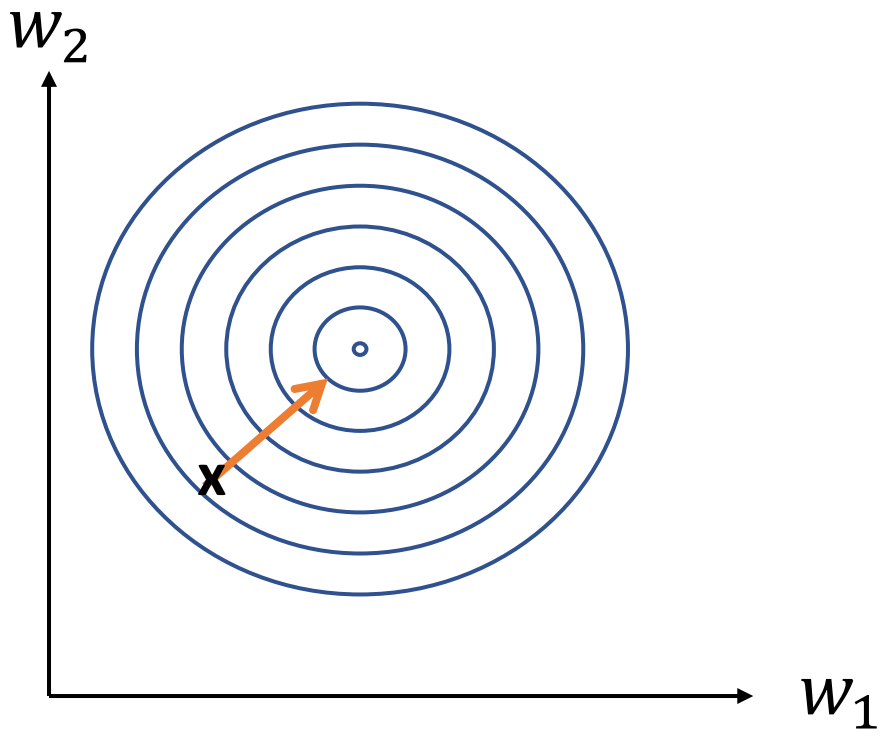
- Direction of steepest descent isn't directly to minimum unless it is a circle

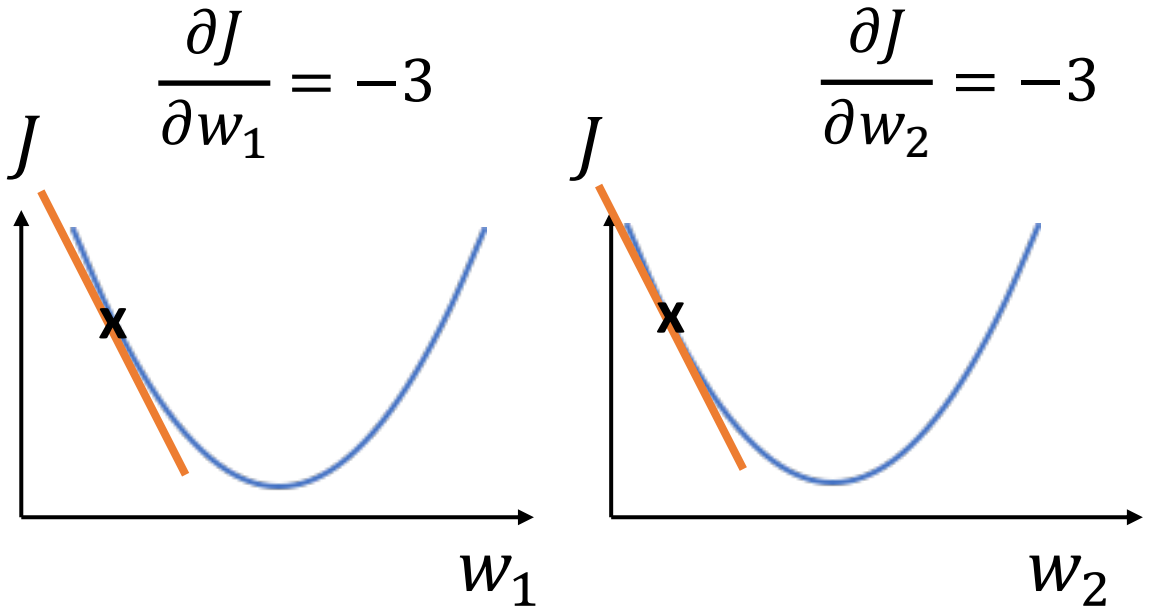




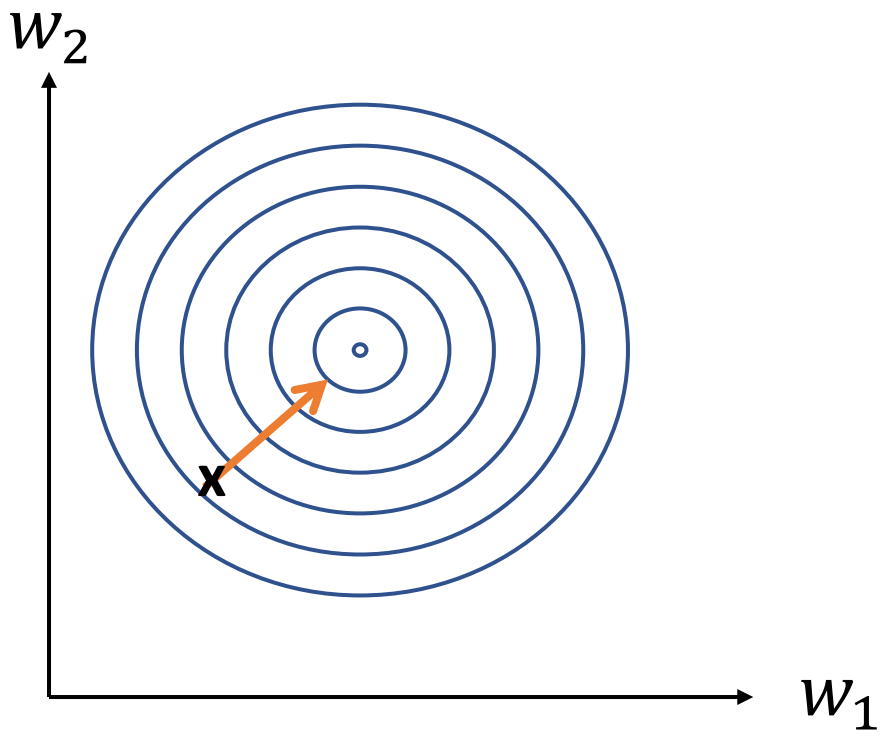

w_2
x
w_1

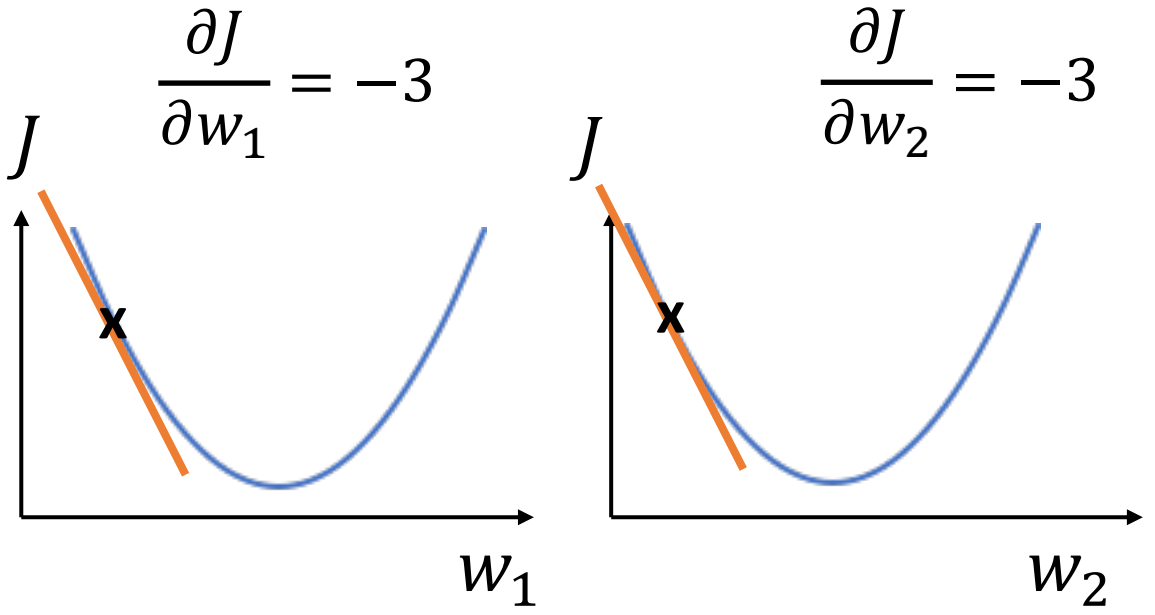

J
w_1
J
w_2





$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial w_2}$$

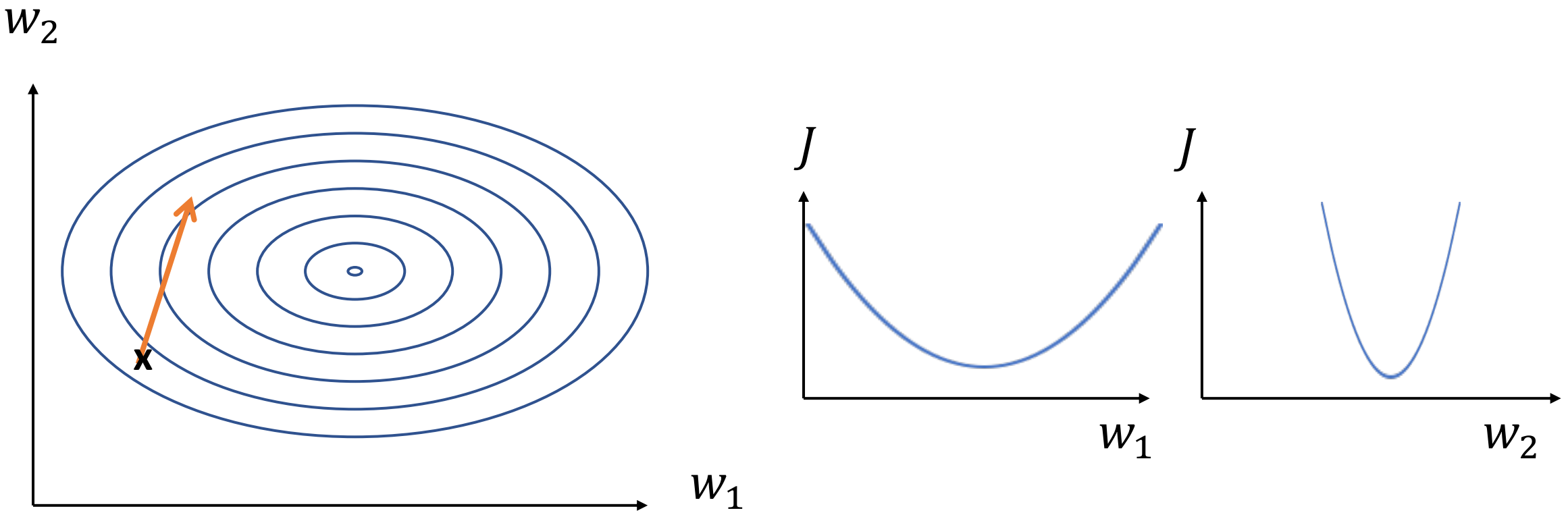


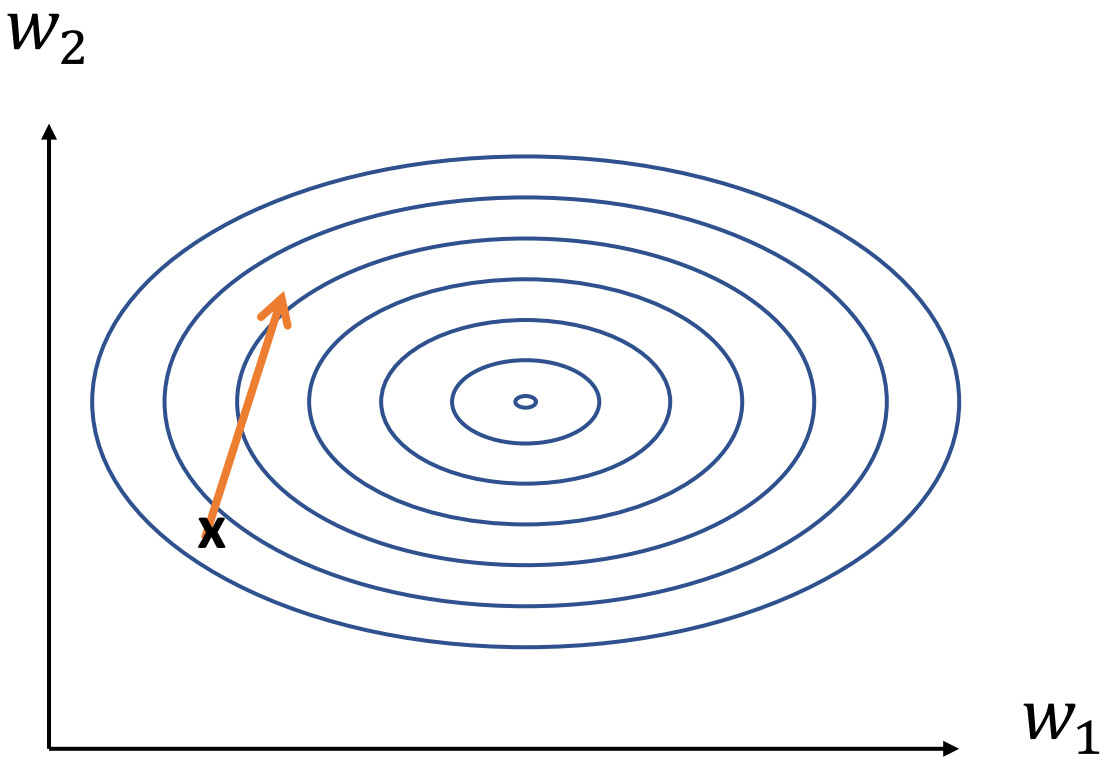


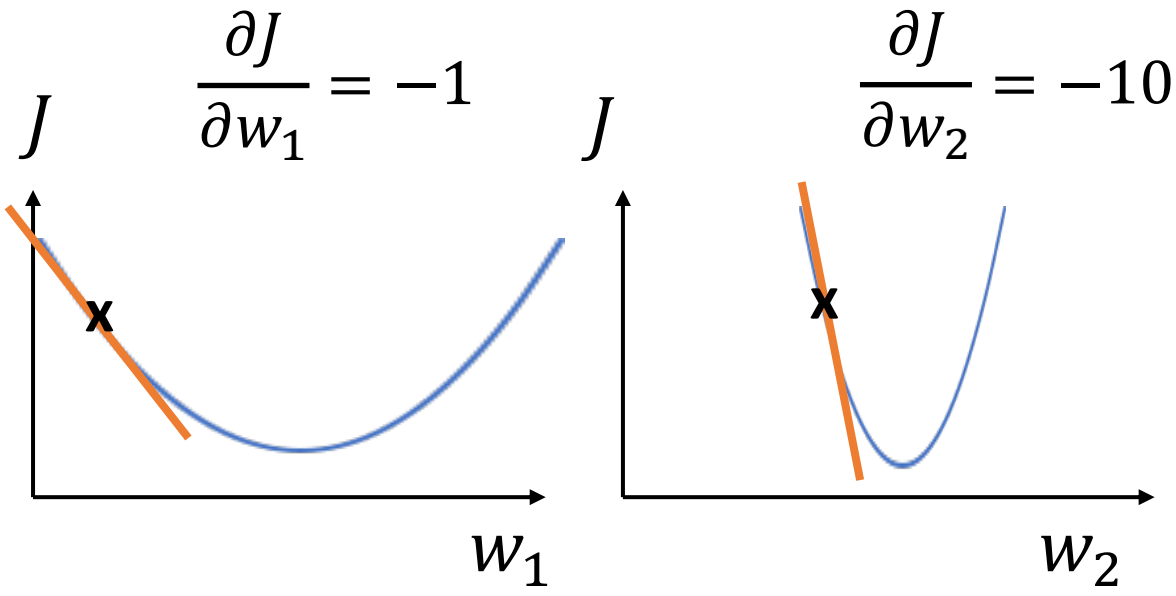
$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial w_2}$$

$$w_1 = w_1 - \alpha(-3)$$

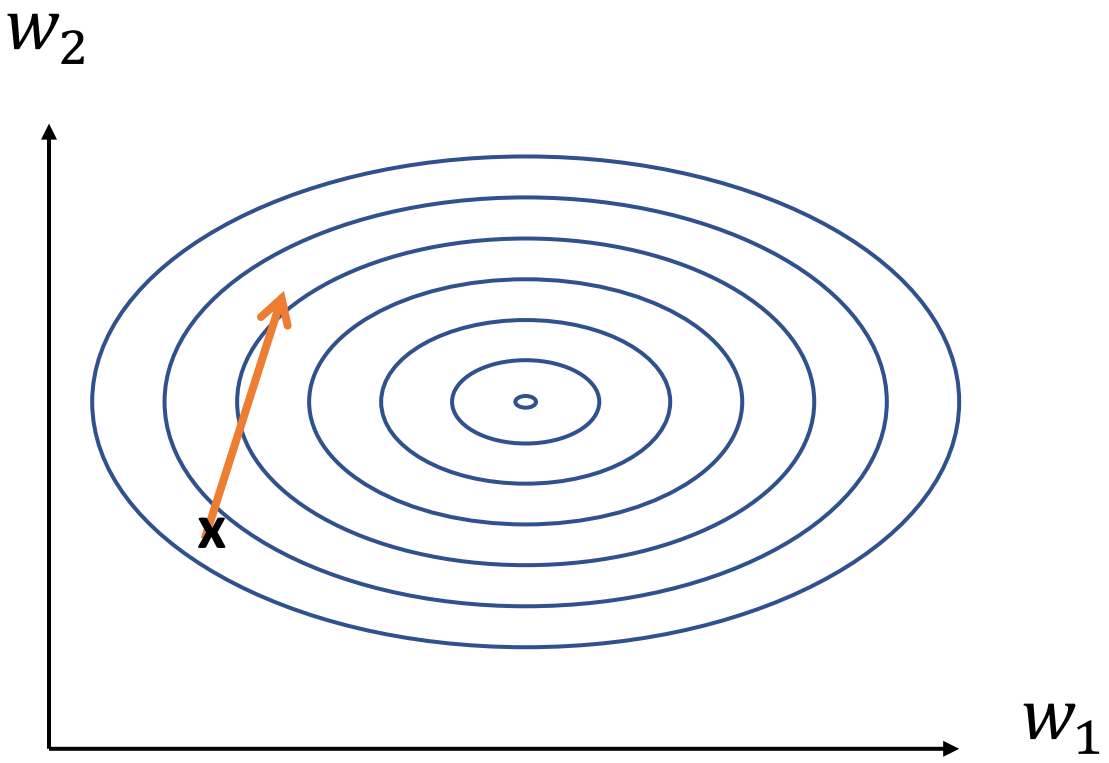
$$w_2 = w_2 - \alpha(-3)$$

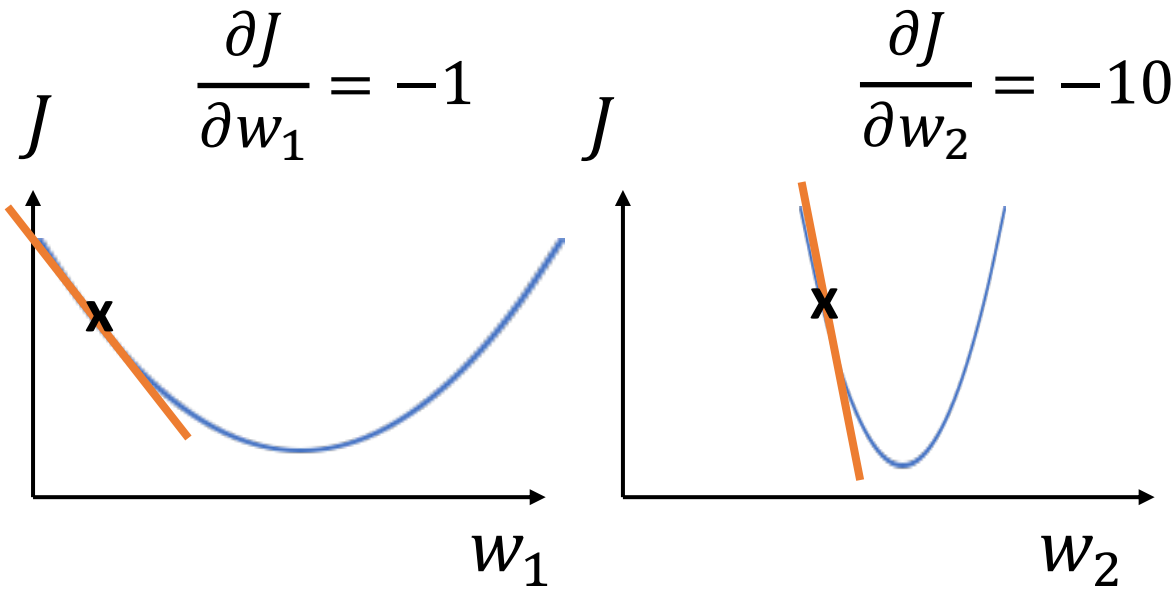

w_2
x
w_1
J
w_1
J
w_2





$$\frac{\partial J}{\partial w_1} \ll \frac{\partial J}{\partial w_2}$$





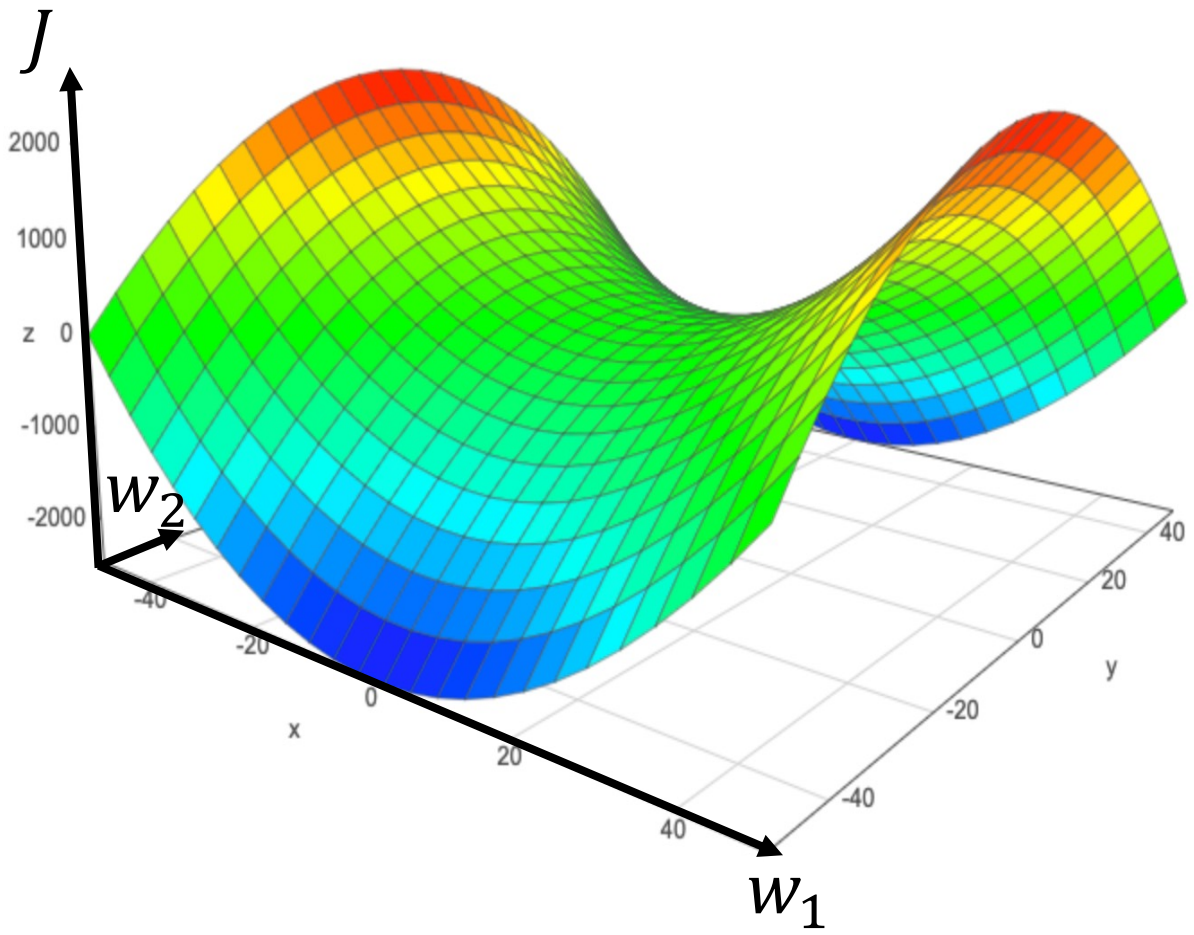
$$\frac{\partial J}{\partial w_1} \ll \frac{\partial J}{\partial w_2}$$

Larger steps at steeper areas, and smaller steps at shallower areas

$$w_1 = w_1 - \alpha(-1)$$

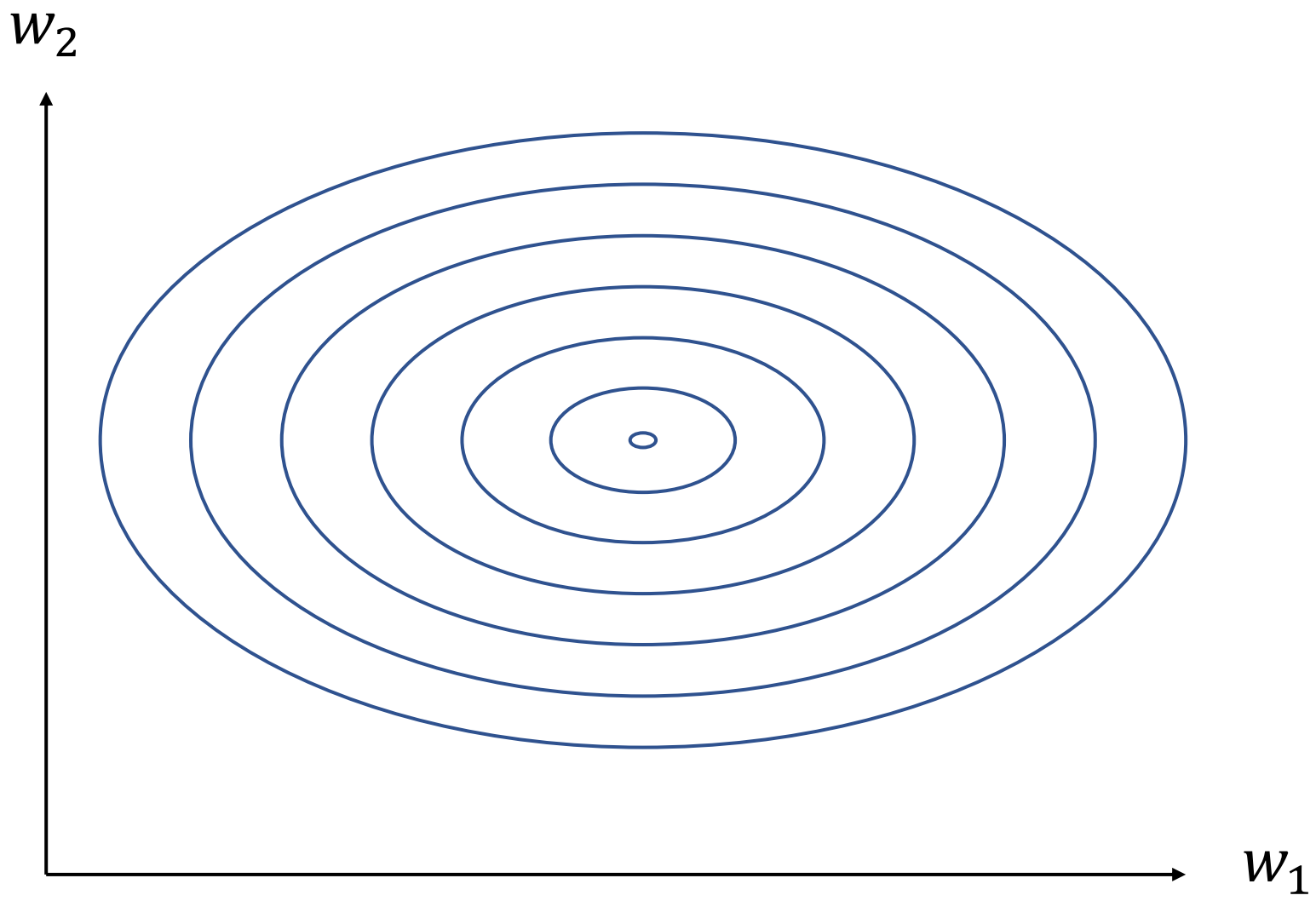
$$w_2 = w_2 - \alpha(-10)$$

2. Local Optima and Saddle Points



- Gradient is 0 at local optima and saddle points
- Saddle points are unstable but simply no gradient info
- Gradient Descent will stop updating parameters

3. Meandering Mini-Batch Gradients



Gradient Descent with Momentum

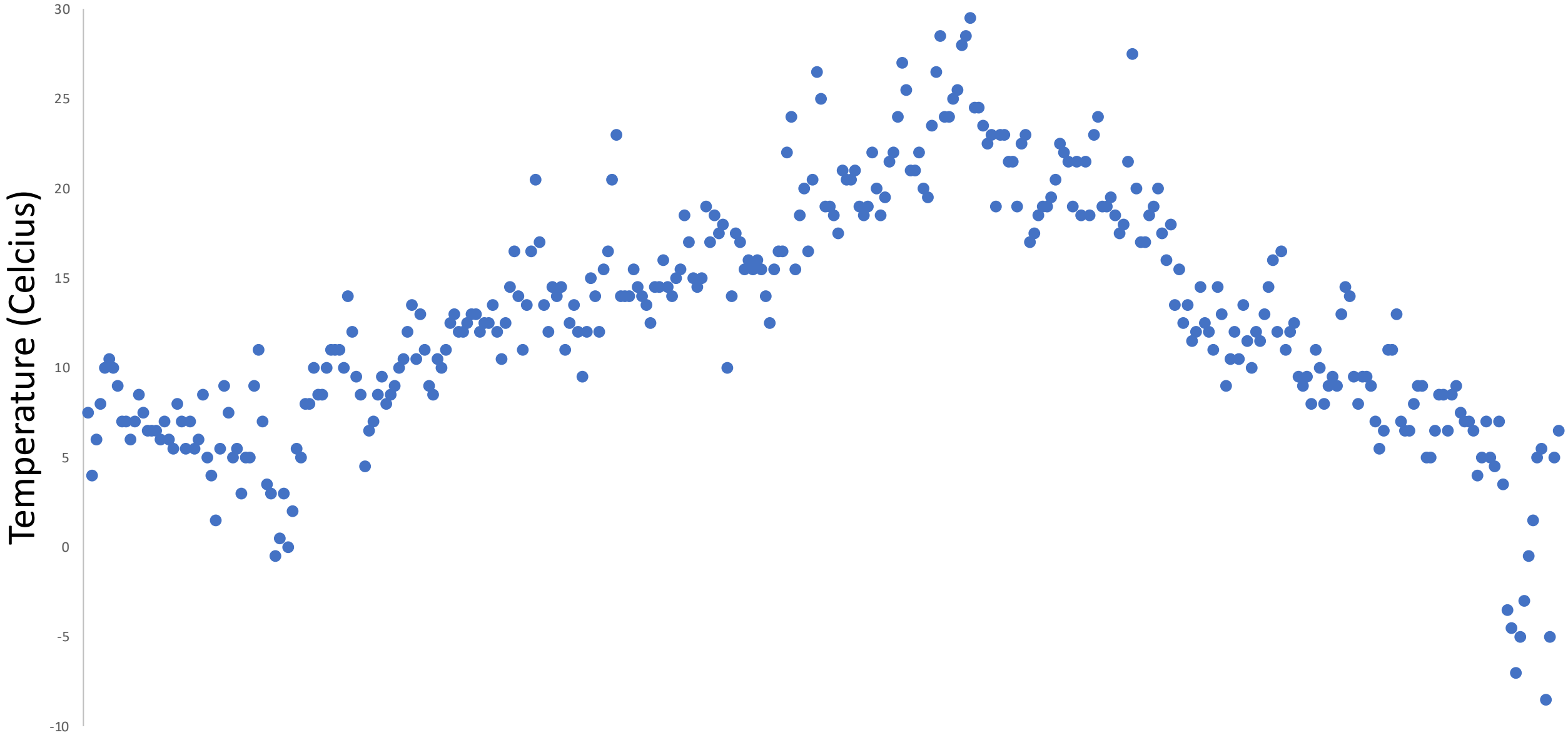
"On the Importance of Initialization and Momentum in Deep Learning", Sutskever, Martens, Dahl, Hinton, 2013, https://www.cs.toronto.edu/~fritz/absps/momentum.pdf

Exponentially Weighted Averages

- A form of moving average
- Can be used to smooth out short-term fluctuations and highlight longer-term trends
- For example, often used in financial data, weather data, etc.

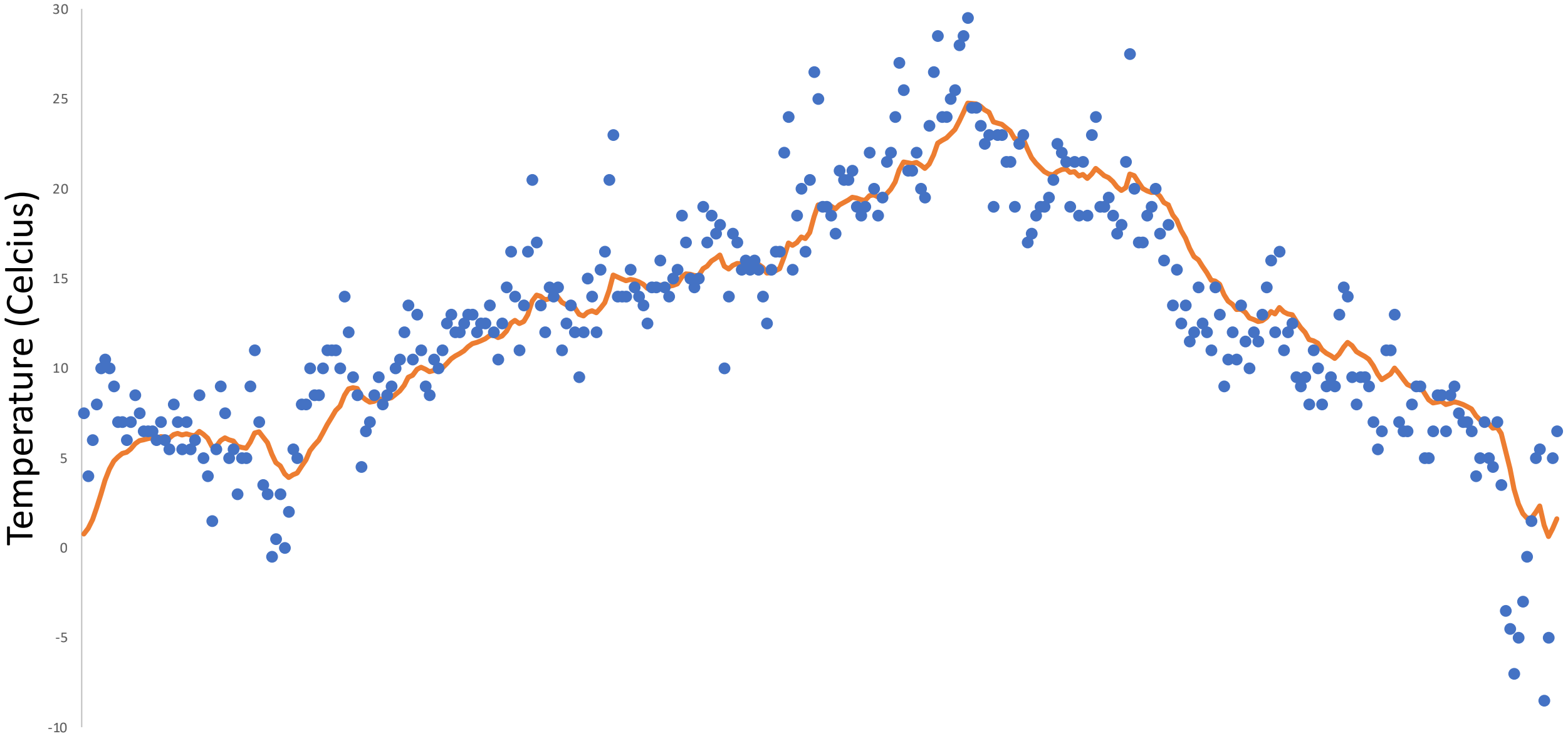
$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot x_t$$

UBC Daily Temperature (1990)



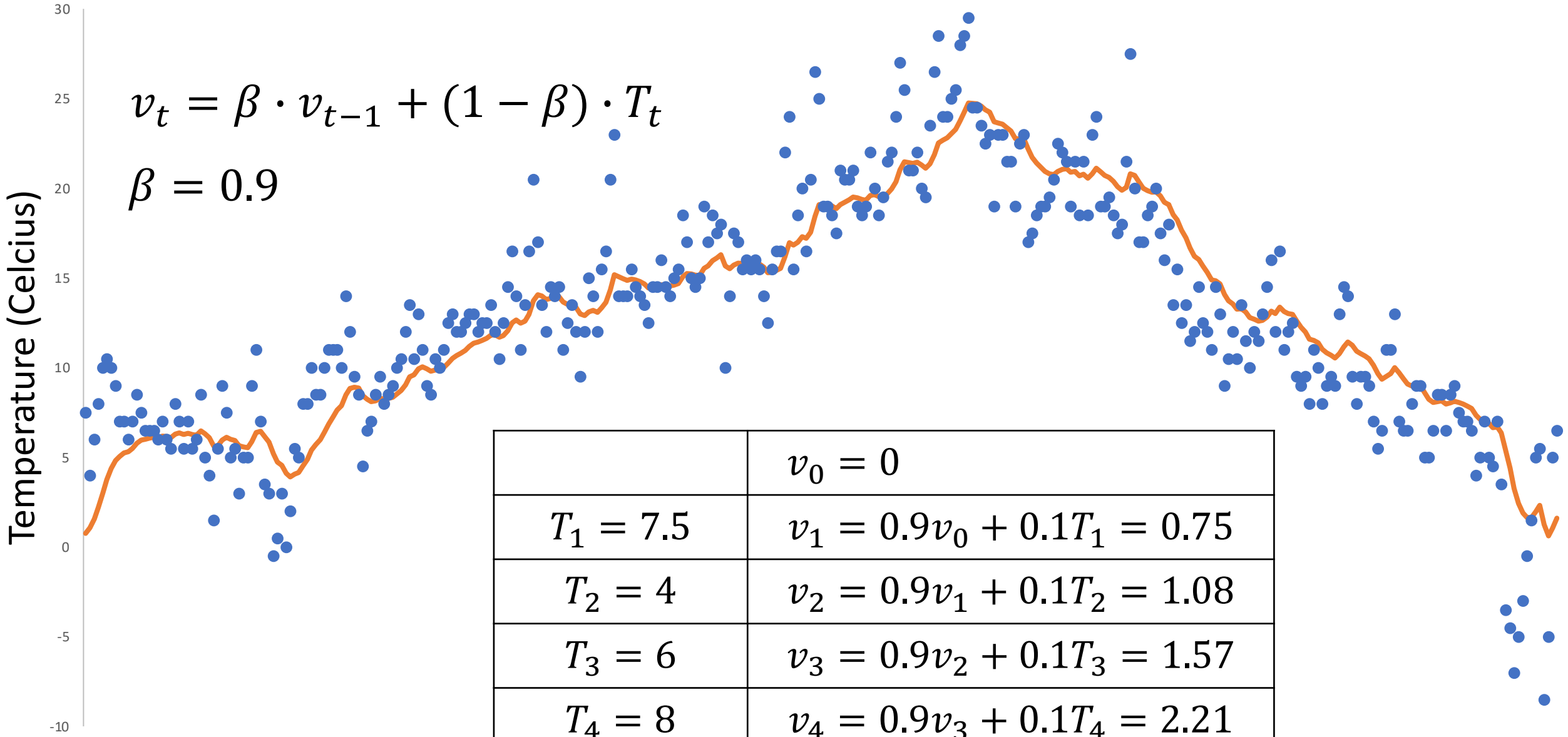
Brad Quinton, Scott Chin

UBC Daily Temperature (1990)



Brad Quinton, Scott Chin

UBC Daily Temperature (1990)



$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot T_t$$

$$\beta = 0.9$$

	$v_0 = 0$
$T_1 = 7.5$	$v_1 = 0.9v_0 + 0.1T_1 = 0.75$
$T_2 = 4$	$v_2 = 0.9v_1 + 0.1T_2 = 1.08$
$T_3 = 6$	$v_3 = 0.9v_2 + 0.1T_3 = 1.57$
$T_4 = 8$	$v_4 = 0.9v_3 + 0.1T_4 = 2.21$

Brad Quinton, Scott Chin

$$\dots$$
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$\dots$$

$$\dots$$
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$\dots$$

$$v_{103} = 0.1T_{103} + 0.9v_{102}$$

$$\dots$$
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$\dots$$

$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$v_{103} = 0.1T_{103} + 0.9(0.1T_{102} + 0.9v_{101})$$

$$\dots$$
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$\dots$$

$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$v_{103} = 0.1T_{103} + 0.9(0.1T_{102} + 0.9v_{101})$$
$$v_{103} = 0.1T_{103} + 0.9\big(0.1T_{102} + 0.9(0.1T_{101} + 0.9v_{100})\big)$$

$$\dots$$
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$\dots$$

$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$v_{103} = 0.1T_{103} + 0.9(0.1T_{102} + 0.9v_{101})$$
$$v_{103} = 0.1T_{103} + 0.9\big(0.1T_{102} + 0.9(0.1T_{101} + 0.9v_{100})\big)$$
$$v_{103} = 0.1T_{103} + 0.9\Big(0.1T_{102} + 0.9\big(0.1T_{101} + 0.9(0.1T_{100} + 0.9v_{99})\big)\Big)$$

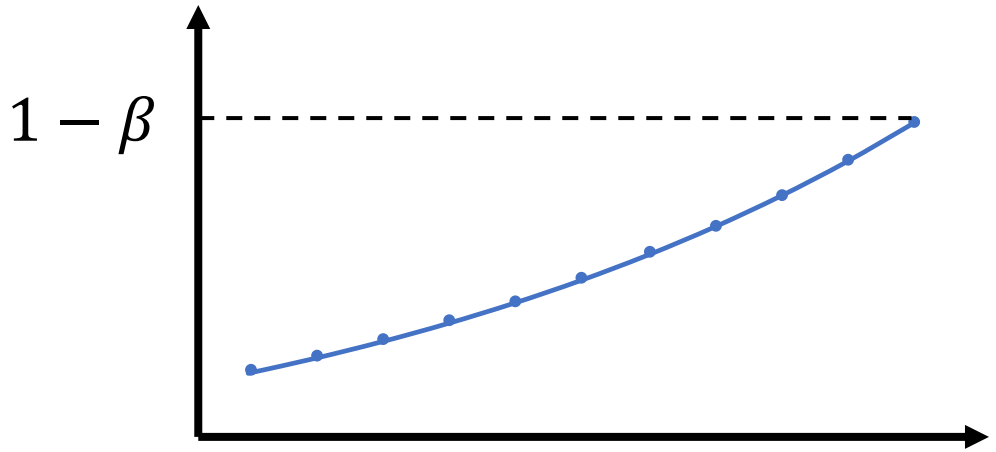
$$\dots$$
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$\dots$$

$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$v_{103} = 0.1T_{103} + 0.9(0.1T_{102} + 0.9v_{101})$$
$$v_{103} = 0.1T_{103} + 0.9\big(0.1T_{102} + 0.9(0.1T_{101} + 0.9v_{100})\big)$$
$$v_{103} = 0.1T_{103} + 0.9\Big(0.1T_{102} + 0.9\big(0.1T_{101} + 0.9(0.1T_{100} + 0.9v_{99})\big)\Big)$$
$$v_{103} = 0.1T_{103} + 0.1 \cdot 0.9T_{102} + 0.1 \cdot 0.9^2T_{101} + 0.1 \cdot 0.9^3T_{100} + 0.1 \cdot 0.9^4T_{99} +$$

$$\dots$$

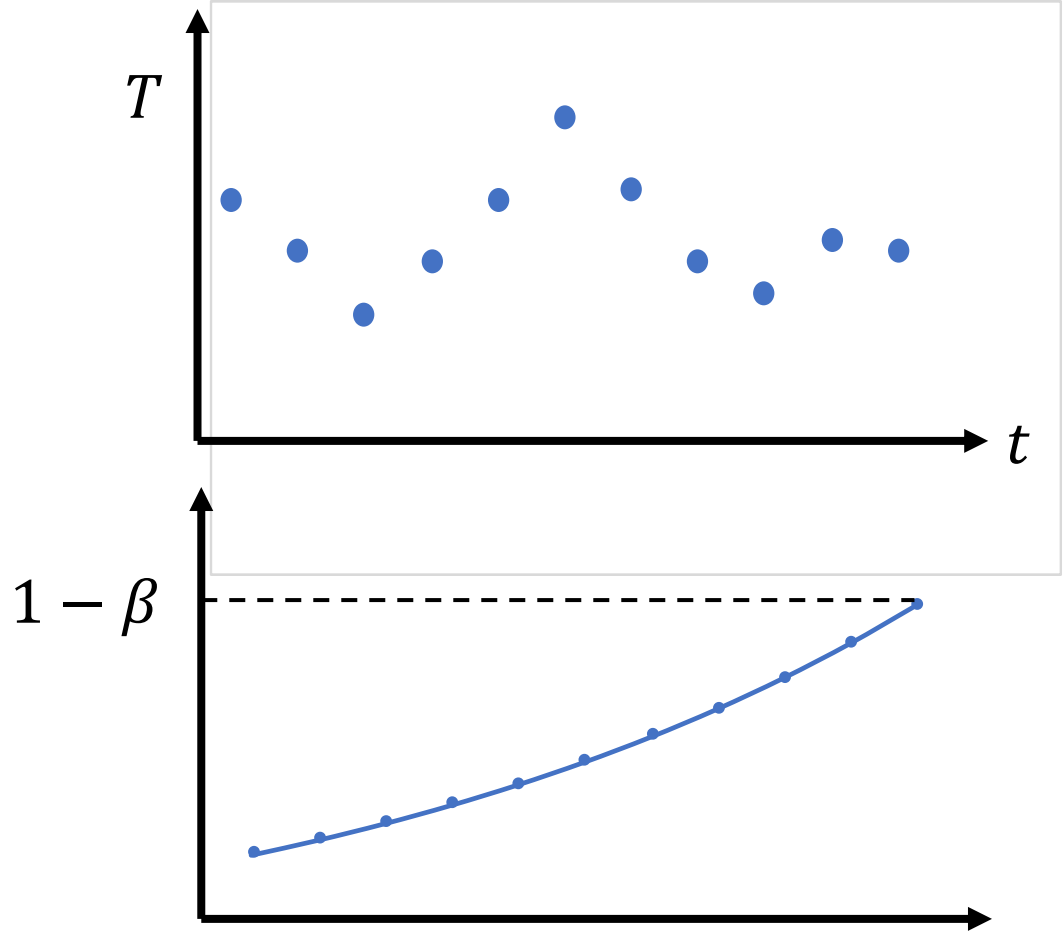
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$

$$\dots$$



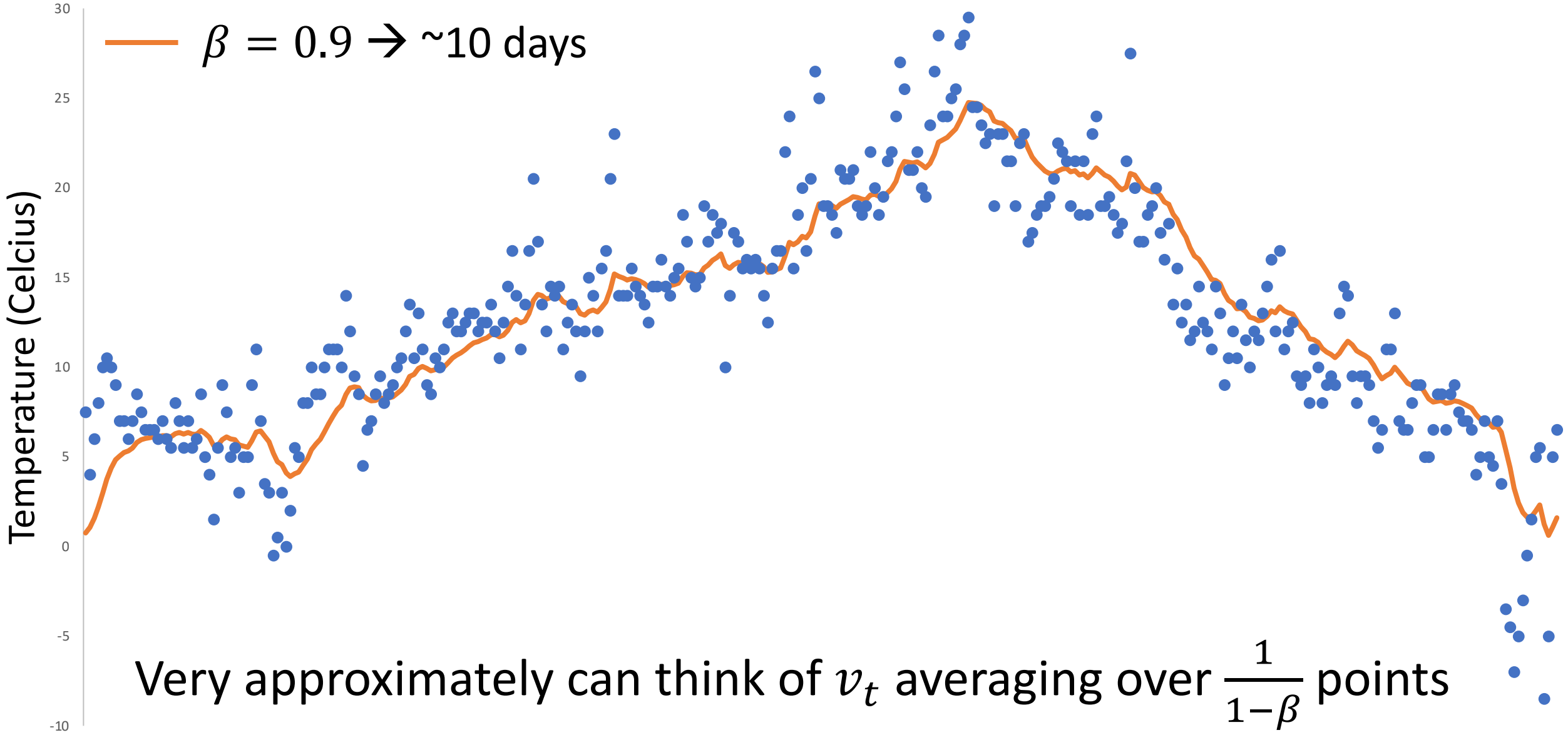
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$v_{103} = 0.1T_{103} + 0.9(0.1T_{102} + 0.9v_{101})$$
$$v_{103} = 0.1T_{103} + 0.9\big(0.1T_{102} + 0.9(0.1T_{101} + 0.9v_{100})\big)$$
$$v_{103} = 0.1T_{103} + 0.9\Big(0.1T_{102} + 0.9\big(0.1T_{101} + 0.9(0.1T_{100} + 0.9v_{99})\big)\Big)$$
$$v_{103} = 0.1T_{103} + 0.1 \cdot 0.9T_{102} + 0.1 \cdot 0.9^2T_{101} + 0.1 \cdot 0.9^3T_{100} + 0.1 \cdot 0.9^4T_{99} +$$

$$\dots$$
$$v_{100} = 0.1T_{100} + 0.9v_{99}$$
$$v_{101} = 0.1T_{101} + 0.9v_{100}$$
$$v_{102} = 0.1T_{102} + 0.9v_{101}$$
$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$\dots$$

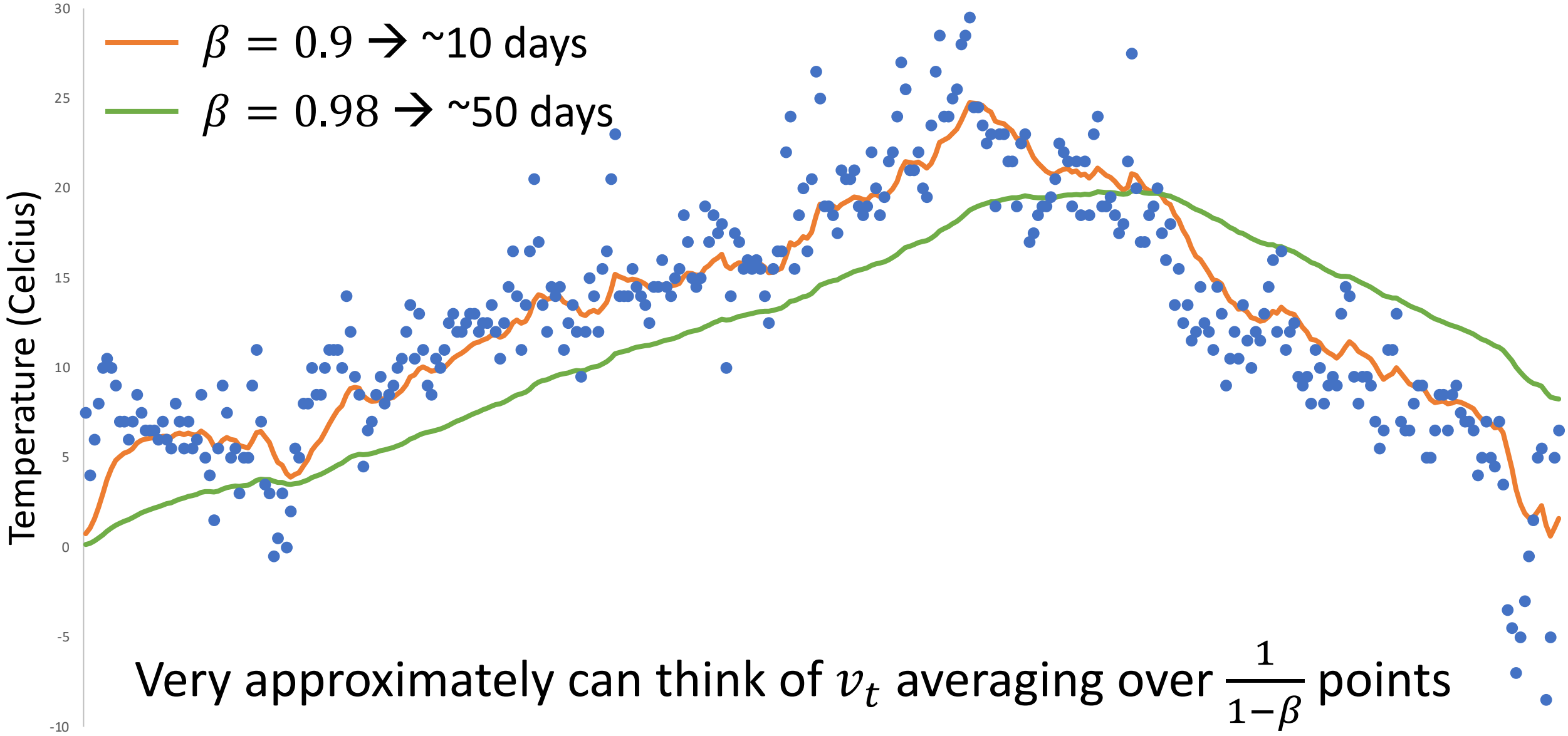


$$v_{103} = 0.1T_{103} + 0.9v_{102}$$
$$v_{103} = 0.1T_{103} + 0.9(0.1T_{102} + 0.9v_{101})$$
$$v_{103} = 0.1T_{103} + 0.9\big(0.1T_{102} + 0.9(0.1T_{101} + 0.9v_{100})\big)$$
$$v_{103} = 0.1T_{103} + 0.9\Big(0.1T_{102} + 0.9\big(0.1T_{101} + 0.9(0.1T_{100} + 0.9v_{99})\big)\Big)$$
$$v_{103} = 0.1T_{103} + 0.1 \cdot 0.9T_{102} + 0.1 \cdot 0.9^2T_{101} + 0.1 \cdot 0.9^3T_{100} + 0.1 \cdot 0.9^4T_{99} +$$

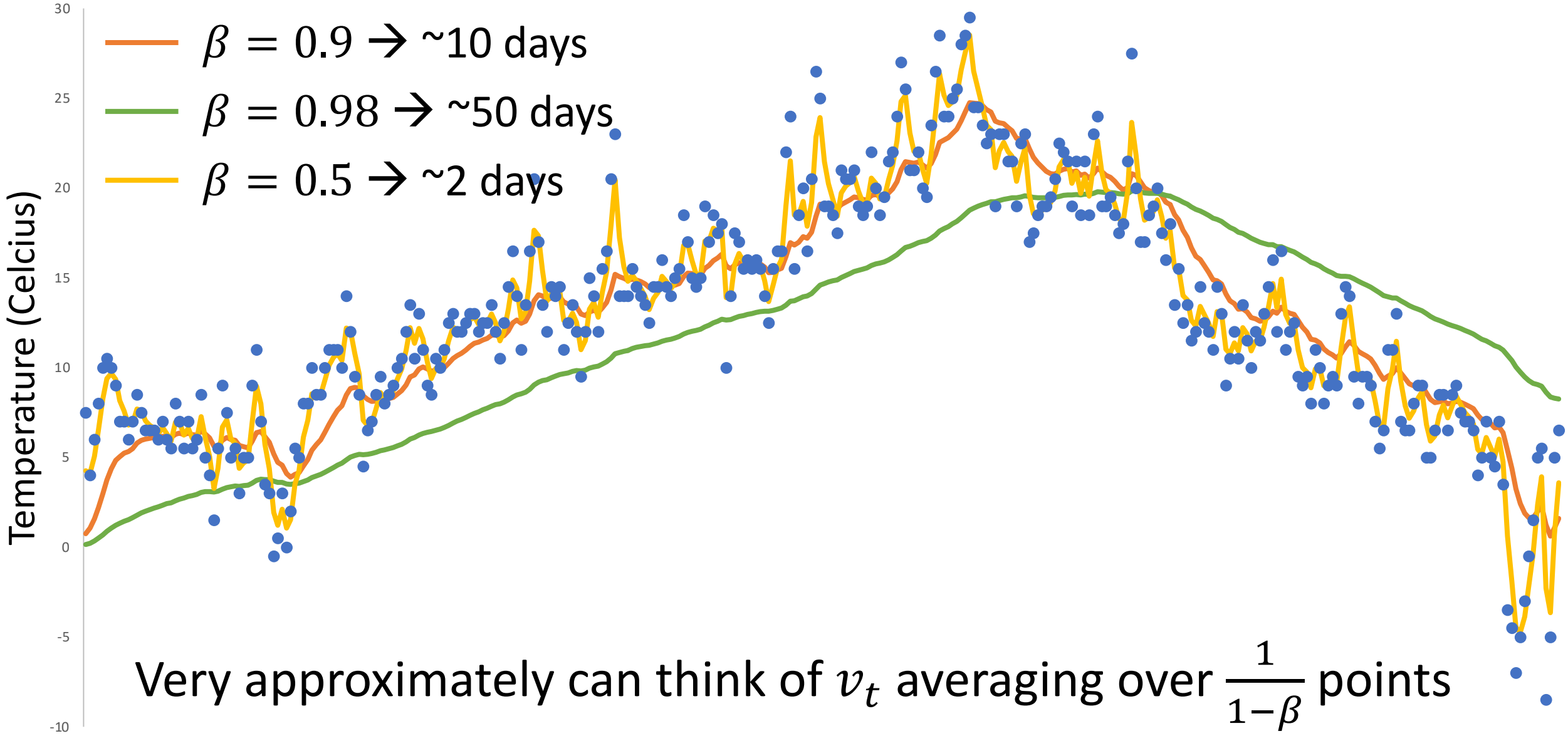
UBC Daily Temperature (1990)



UBC Daily Temperature (1990)



UBC Daily Temperature (1990)



Very approximately can think of v_t averaging over $\frac{1}{1-\beta}$ points

Recap: Problems with Gradient Descent

1. Different rates of change in different parameter dimensions
2. Local minima and saddle points
3. Meandering gradients with mini-batches

Recap: Problems with Gradient Descent

1. Different rates of change in different parameter dimensions
2. Local minima and saddle points
3. Meandering gradients with mini-batches

How can Exponentially Weighted Averages help?

Recap: Problems with Gradient Descent

1. Different rates of change in different parameter dimensions
2. Local minima and saddle points
3. Meandering gradients with mini-batches

How can Exponentially Weighted Averages help?

Instead of using the gradients directly to update the parameters, use an exponentially weighted average of past gradients

Gradient Descent with Momentum

```
w = initialize()
v = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v = beta*v + dJ_dw
  w = w - learning_rate * v
```

Note: I've removed the (1- β) term to simplify our following discussion. But I will show afterwards that this formulation is equivalent with a bit of variable transformation

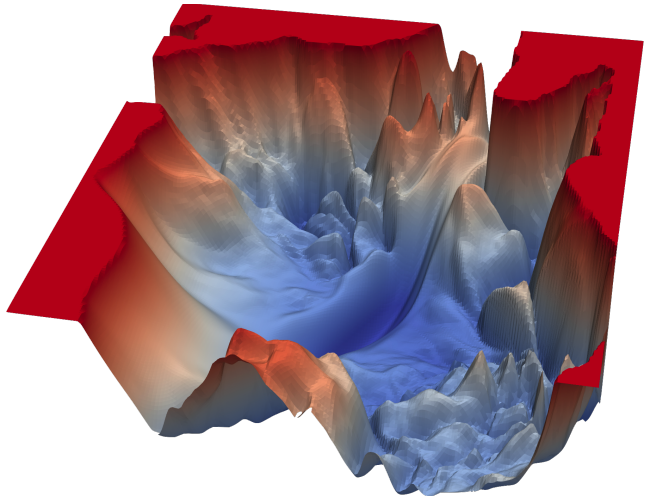
Gradient Descent with Momentum

```
w = initialize()
v = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v = beta*v + dJ_dw
  w = w - learning_rate * v
```

Note: Keep a running average for each parameter separately

Momentum Analogy (Ball Rolling down a hill)

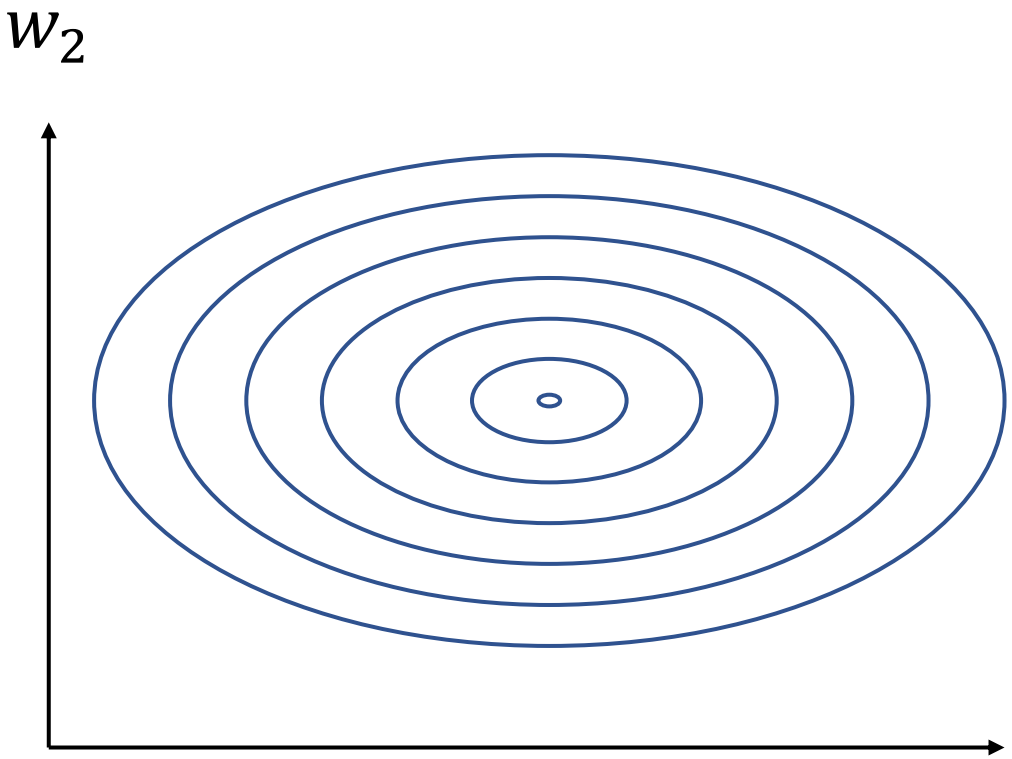
$$v_t = \beta \cdot v_{t-1} + \frac{\partial J}{\partial w}$$

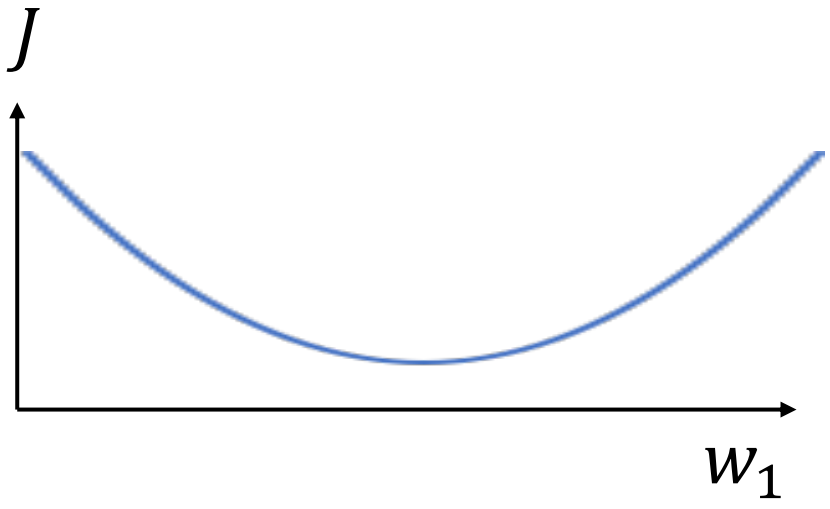


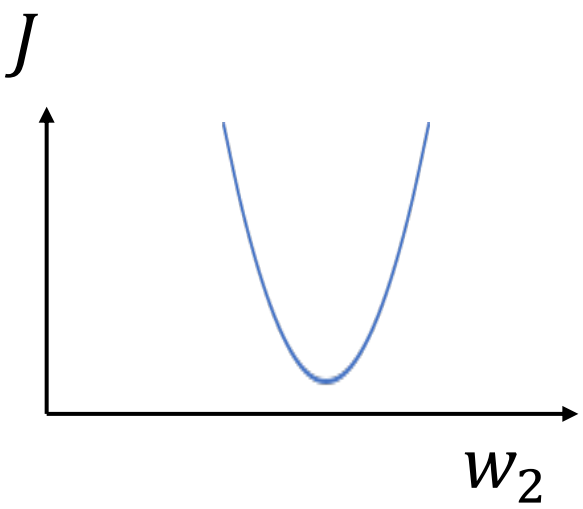
- Cost is potential energy
- $\frac{\partial J}{\partial w}$ is acceleration due to gravity
- Historical portion of exponentially weighted average is velocity (momentum) accumulated due the gradients (acceleration) over previous iterations/mini-batches
- Beta is friction (decay factor) that slows down velocities

Problem 1: Gradients in different dimensions

- Consistent gradient will build up velocity from accumulated acceleration
- Inconsistent gradients will cancel out (e.g. opposing accelerations)







$$v_t = \beta \cdot v_{t-1} + \frac{\partial J}{\partial w}$$

Problem 2 : Local minima and saddle points

$$v_t = \beta \cdot v_{t-1} + \frac{\partial J}{\partial w}$$

- At saddle points, gradient is 0, but historical component (momentum) won't be
- At local minima, velocity can help get back out of some local minima

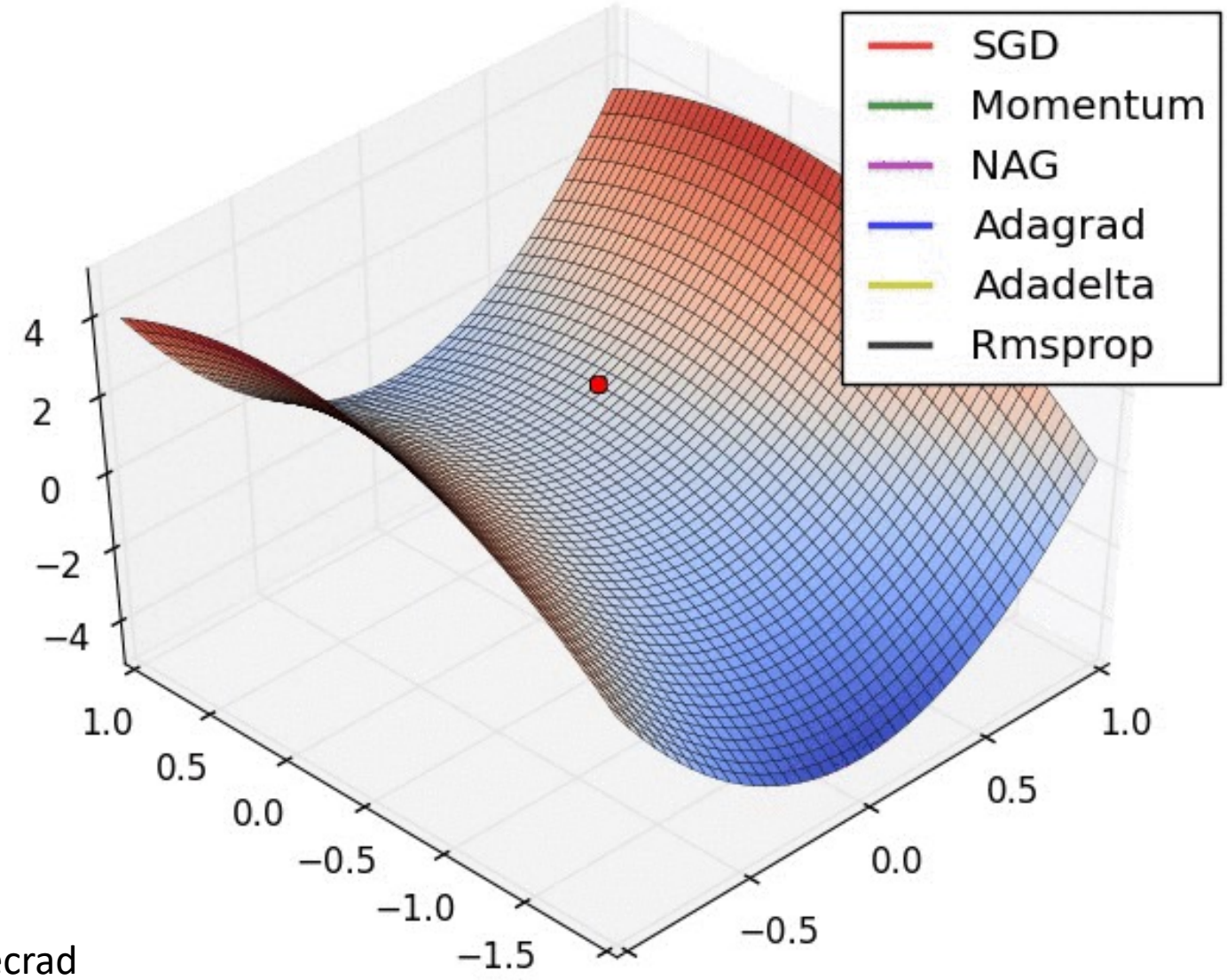
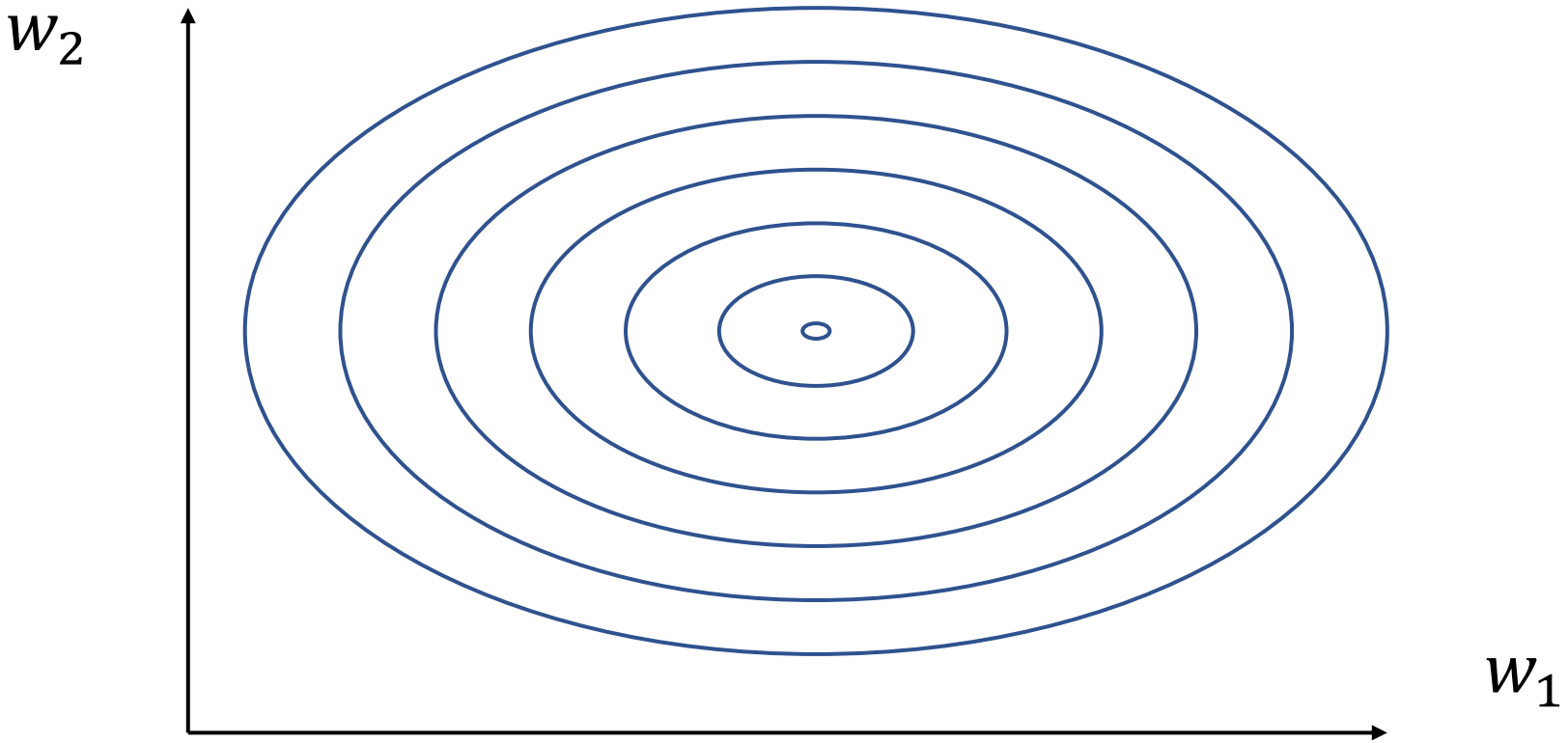


Image Source: Alec Radford https://twitter.com/alecrad

Problem 3: Meandering mini-batch gradients

- Just like we discussed for the first problem, the moving average will create a ”smoothing” effect



So what about the $(1-\beta)$ term?

```
v = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v = b*v + (1-b)*dJ_dw
  w = w - learning_rate * v
```

```
v = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v = b*v + dJ_dw
  w = w - learning_rate * v
```

So what about the $(1-\beta)$ term?

```
v = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v = b*v + (1-b)*dJ_dw
  w = w - learning_rate * v
```

```
v = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v = b*v + dJ_dw
  w = w - learning_rate * v
```

So what about the $(1-\beta)$ term?

```
v = b*v + (1-b)*dJ_dw
w = w - learning_rate * v
```

```
v = b*v + dJ_dw
w = w - learning_rate * v
```

So what about the $(1-\beta)$ term?

$$v_2 = 0.1T_2 + 0.9\big(0.1T_1 + 0.9(0.1T_0)\big)$$

```
v=(1-b)*dJ_dw2+(1-b)*b*dJ_dw1+(1-b)*b*b*dJ_dw0
w = w - learning_rate * v
```

```
v = dJ_dw2 + b*b*dJ_dw1 + b*b*dJ_dw0
w = w - learning_rate * v
```

For demonstration, write out 3 terms of v

So what about the $(1-\beta)$ term?

Factor out (1-b)

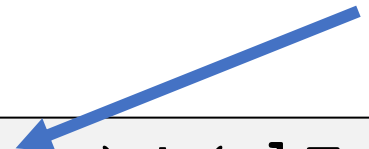
```
v=(1-b)*dJ_dw2+(1-b)*b*dJ_dw1+(1-b)*b*b*dJ_dw0
w = w - learning_rate * v
```

```
v = dJ_dw2 + b*b*dJ_dw1 + b*b*dJ_dw0
w = w - learning_rate * v
```

For demonstration, write out 3 terms of v

So what about the $(1-\beta)$ term?

Factor out (1-b)



```
v=(1-b)*(dJ_dw2 + b*dJ_dw1 + b*b*dJ_dw0)
w = w - learning_rate * v
```

```
v = dJ_dw2 + b*d*dJ_dw1 + b*b*dJ_dw0
w = w - learning_rate * v
```

So what about the $(1-\beta)$ term?

Move it down here

```
v = dJ_dw2 + b*dJ_dw1 + b*b*dJ_dw0
w = w - learning_rate * (1-b) * v
```

```
v = dJ_dw2 + b*d*dJ_dw1 + b*b*dJ_dw0
w = w - learning_rate * v
```

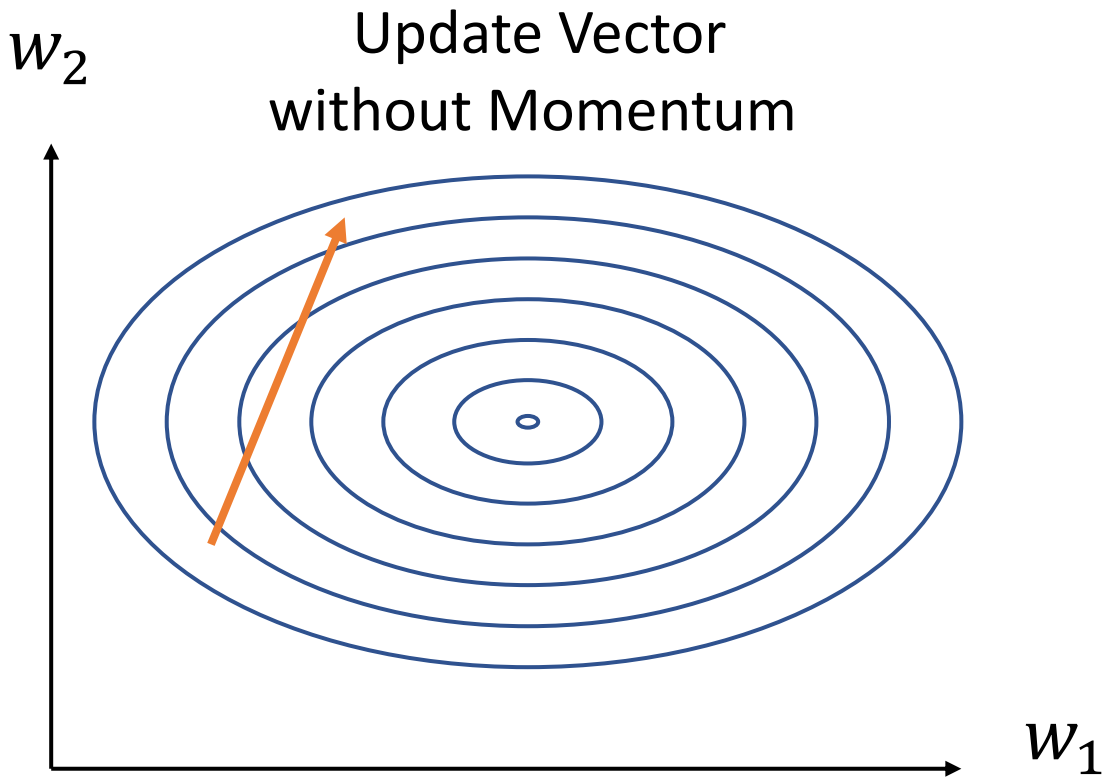
So what about the $(1-\beta)$ term?

Both formulations equivalent with appropriate change to learning rates

```
v = dJ_dw2 + b*dJ_dw1 + b*b*dJ_dw0
w = w - learning_rate * (1-b) * v
```

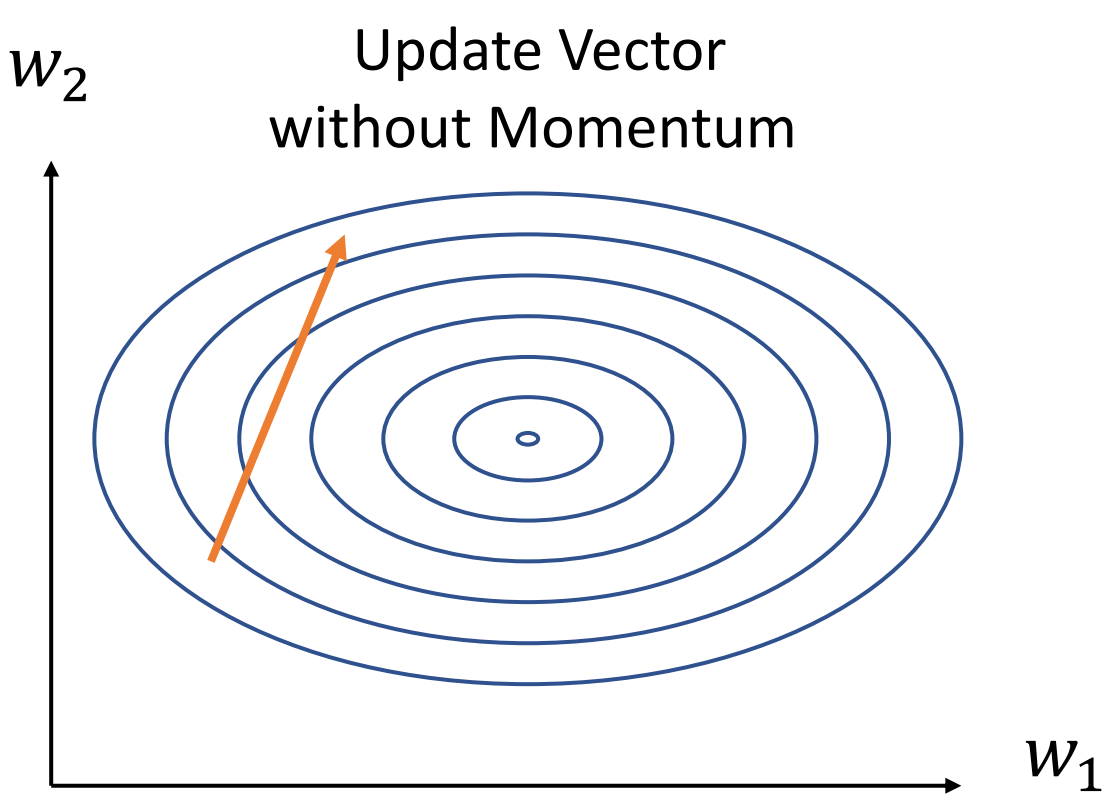
```
v = dJ_dw2 + b*d*dJ_dw1 + b*b*dJ_dw0
w = w - learning_rate * v
```

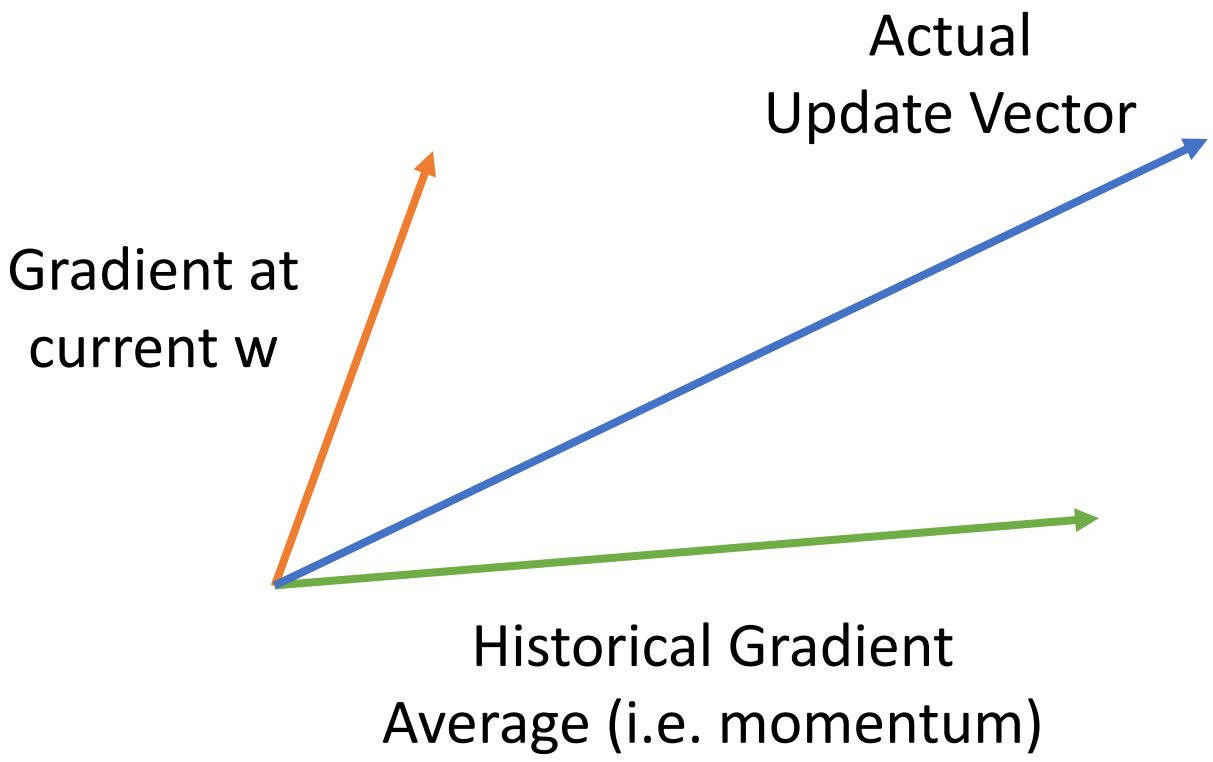
Momentum so far



Momentum so far

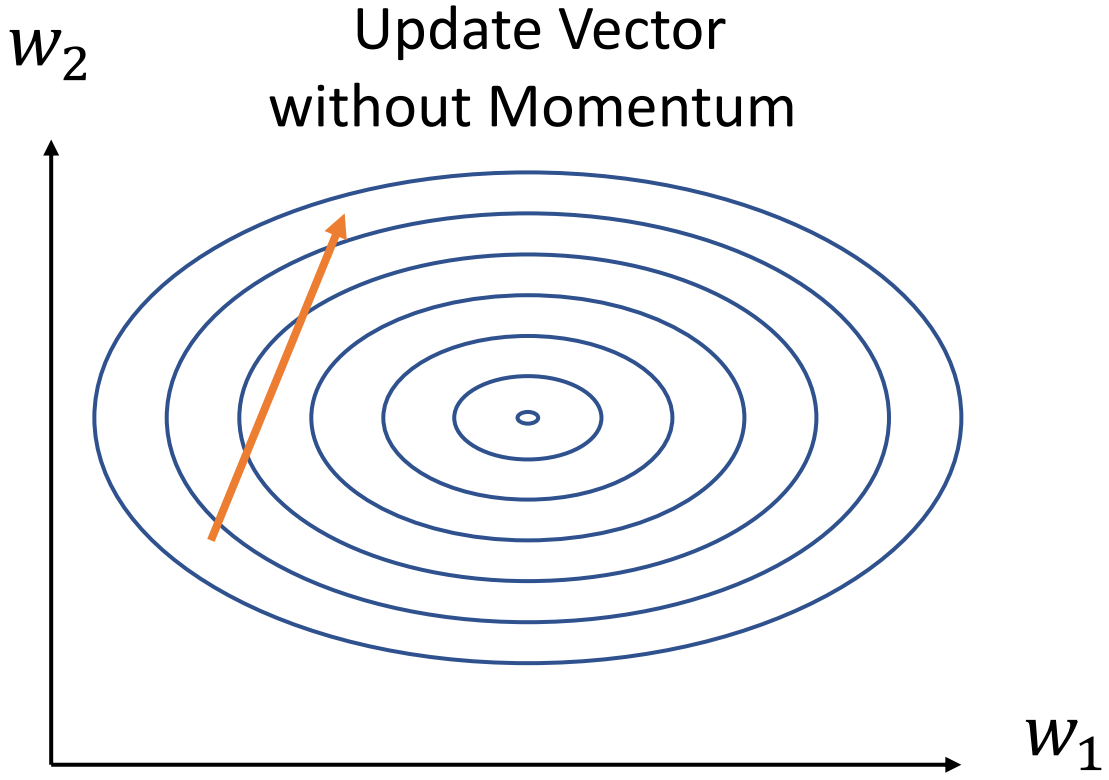
$$v_t = \beta \cdot v_{t-1} + \frac{\partial J}{\partial w}$$

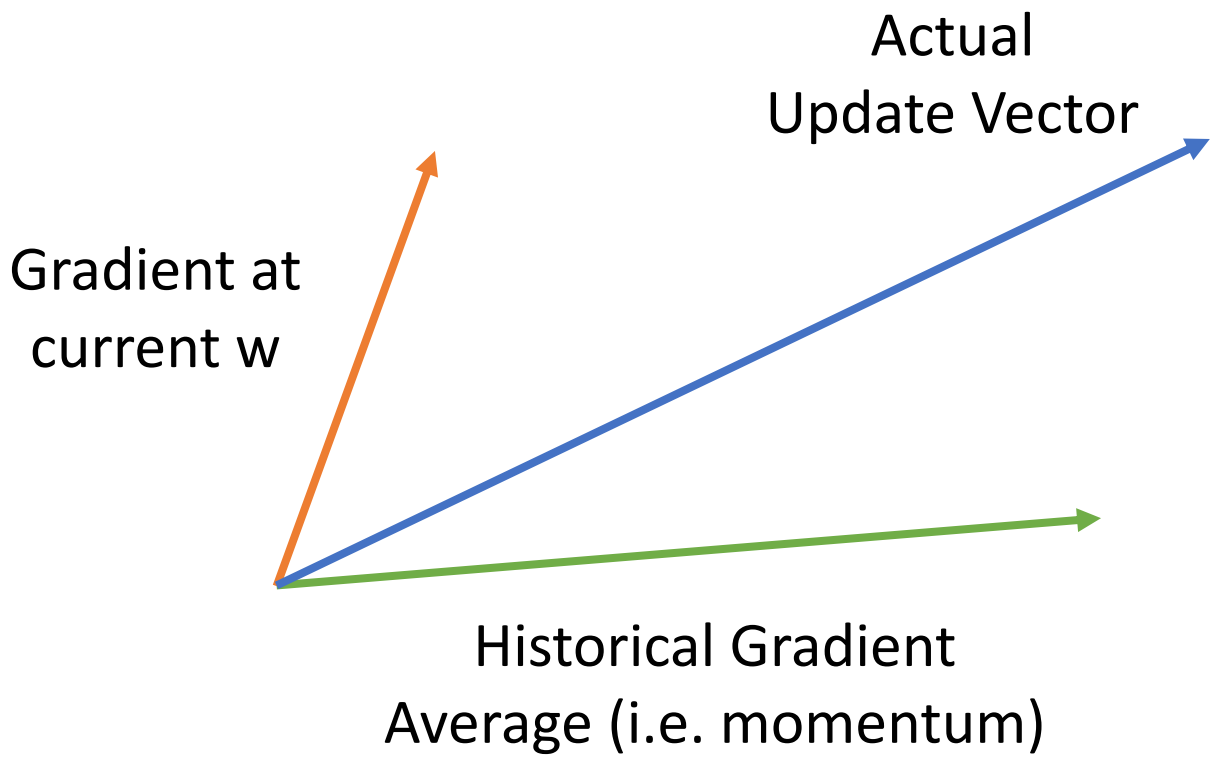




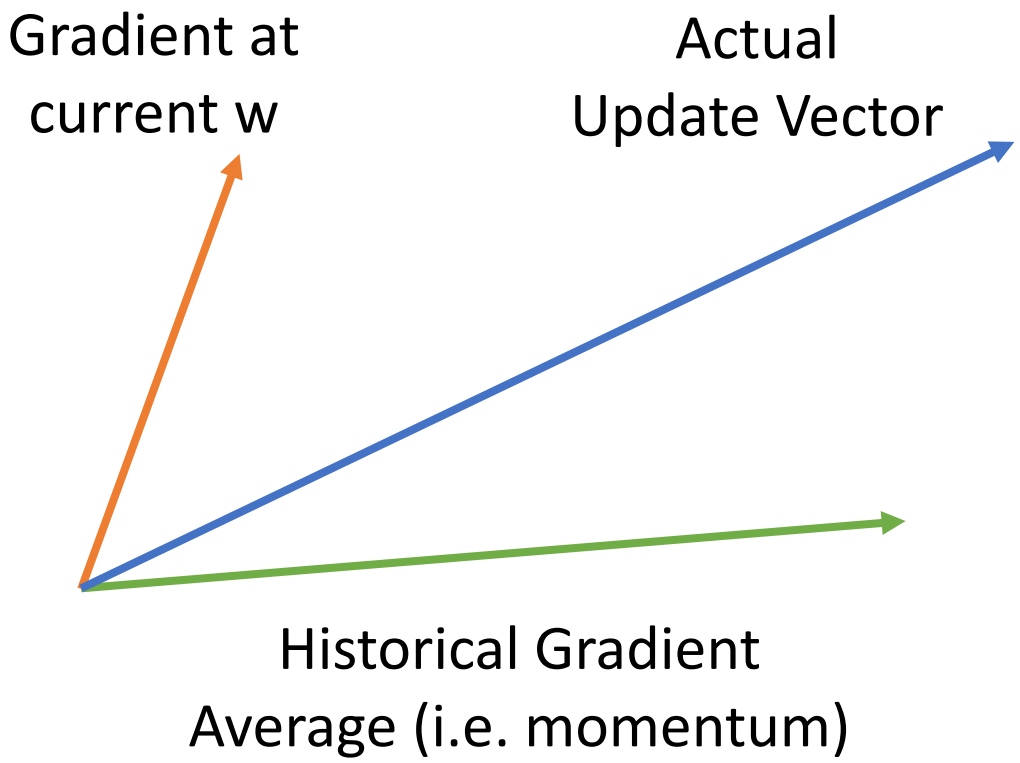
Momentum so far

$$v_t = \beta \cdot v_{t-1} + \frac{\partial J}{\partial w}$$

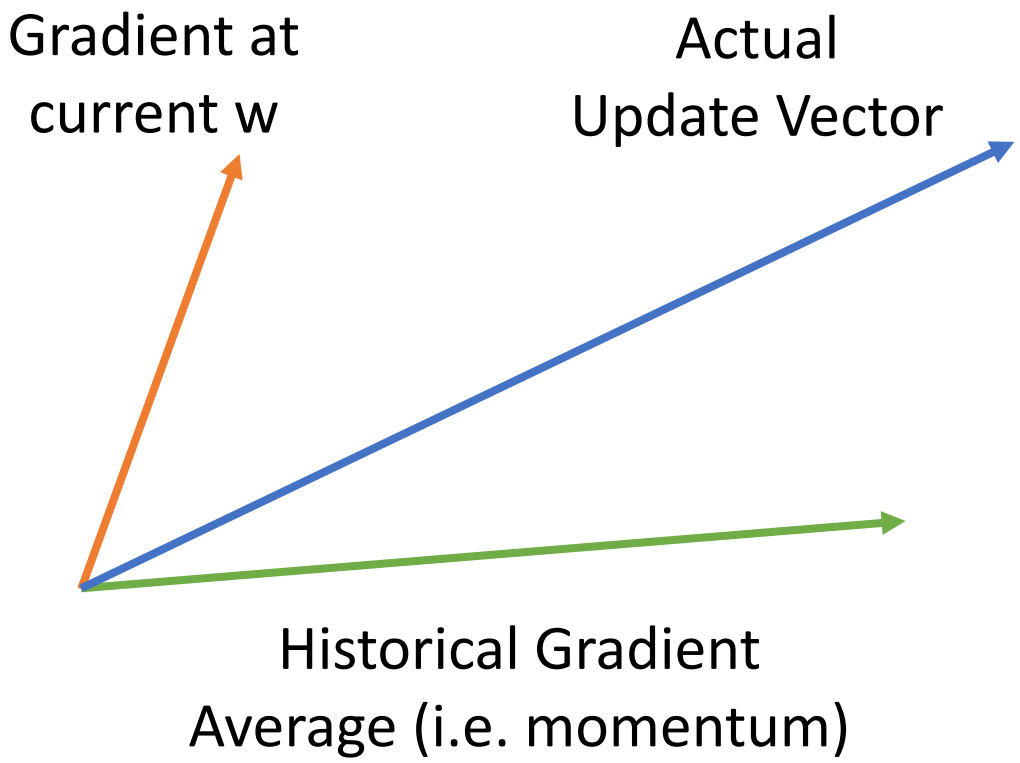




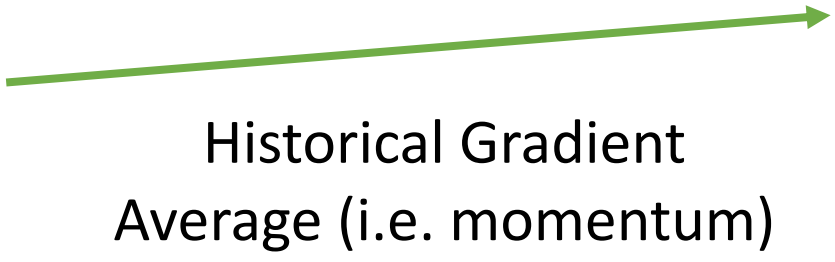
Classic Momentum



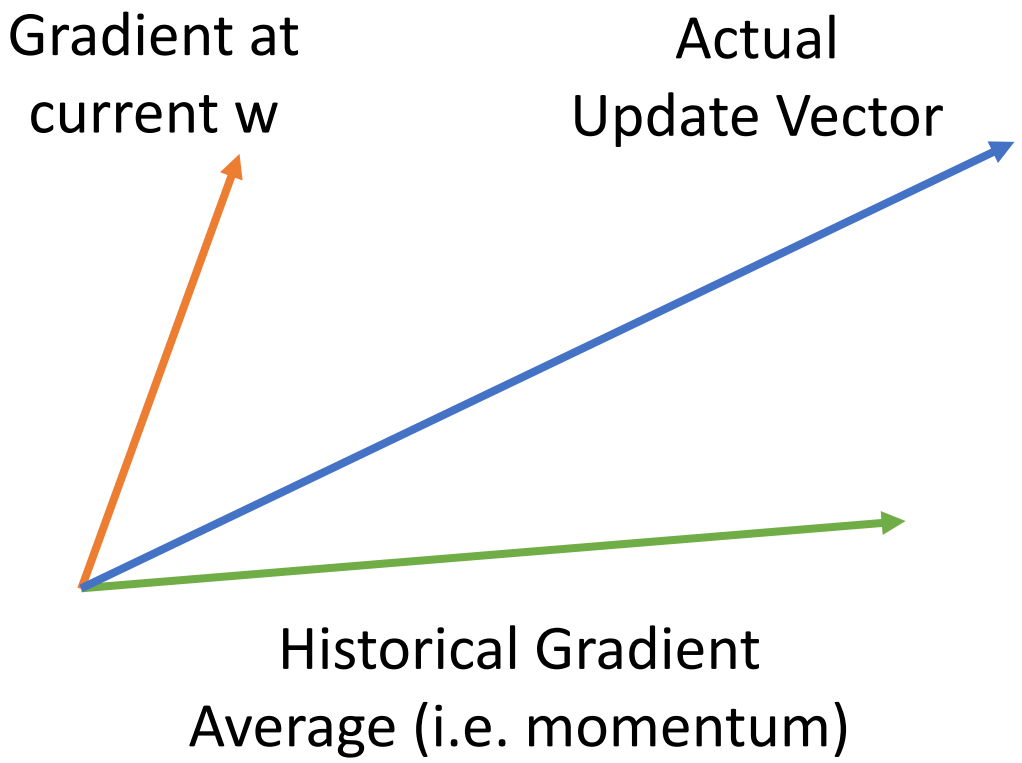
Classic Momentum



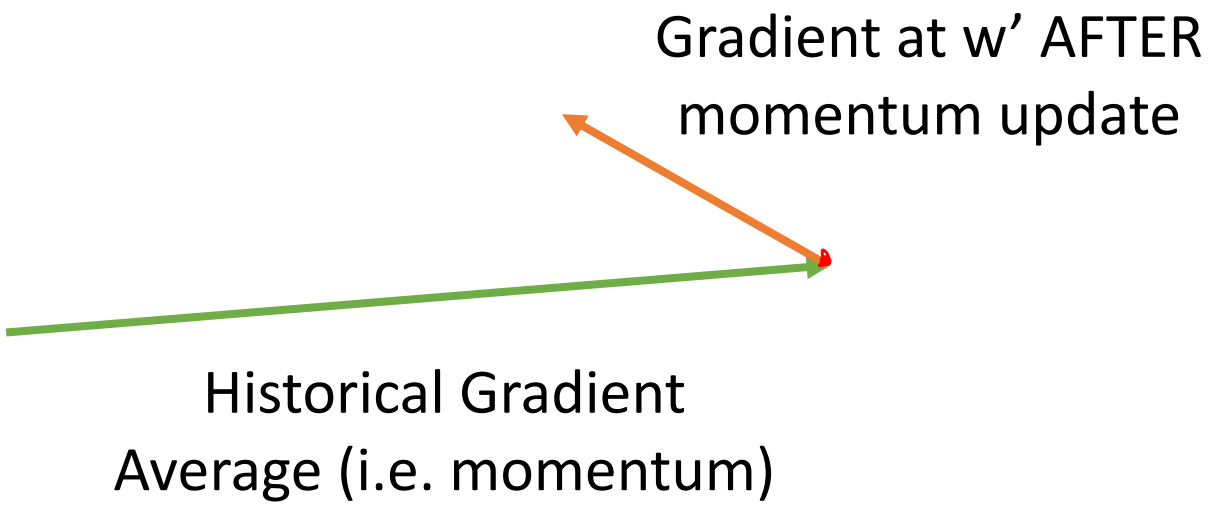
Nesterov Momentum



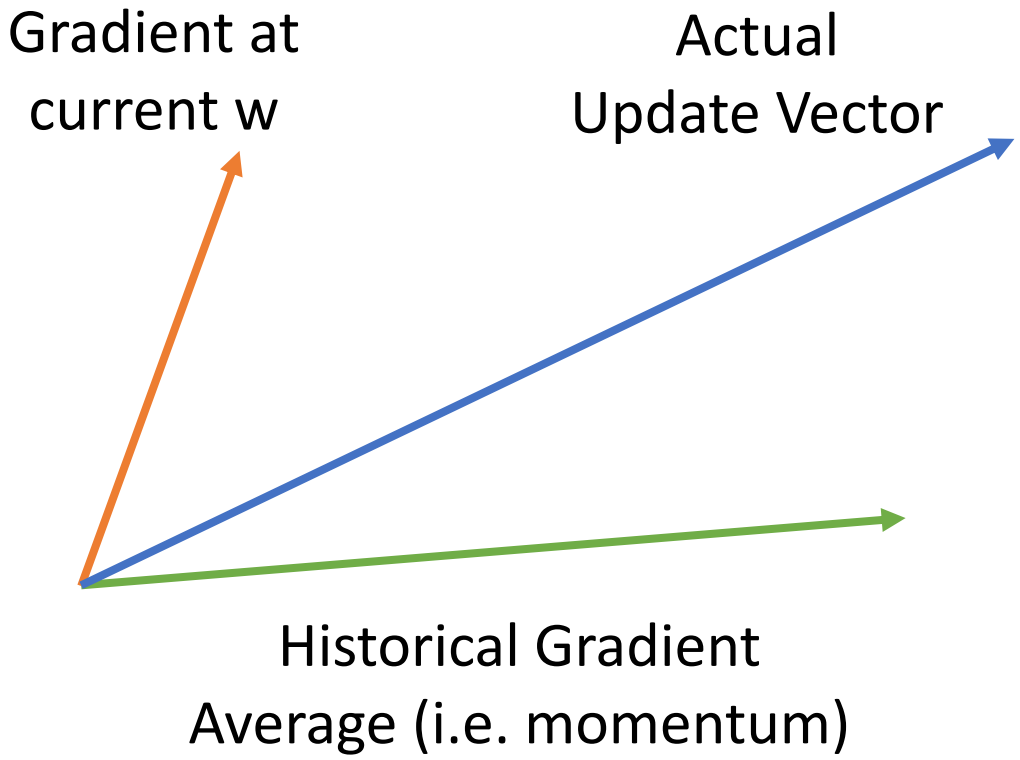
Classic Momentum



Nesterov Momentum

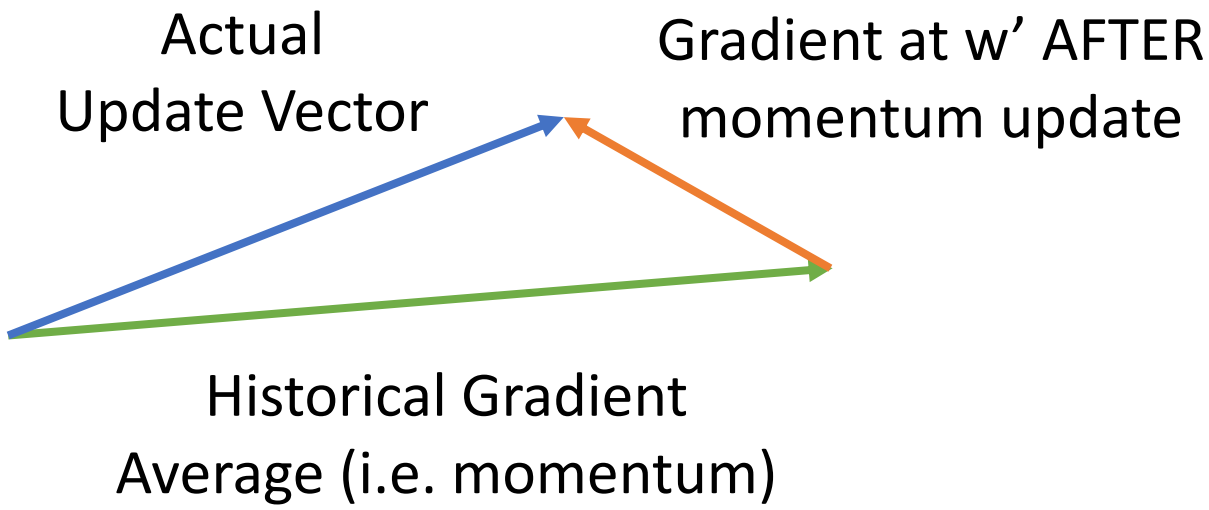


Classic Momentum



Nesterov Momentum

Difference is in where you compute the gradient



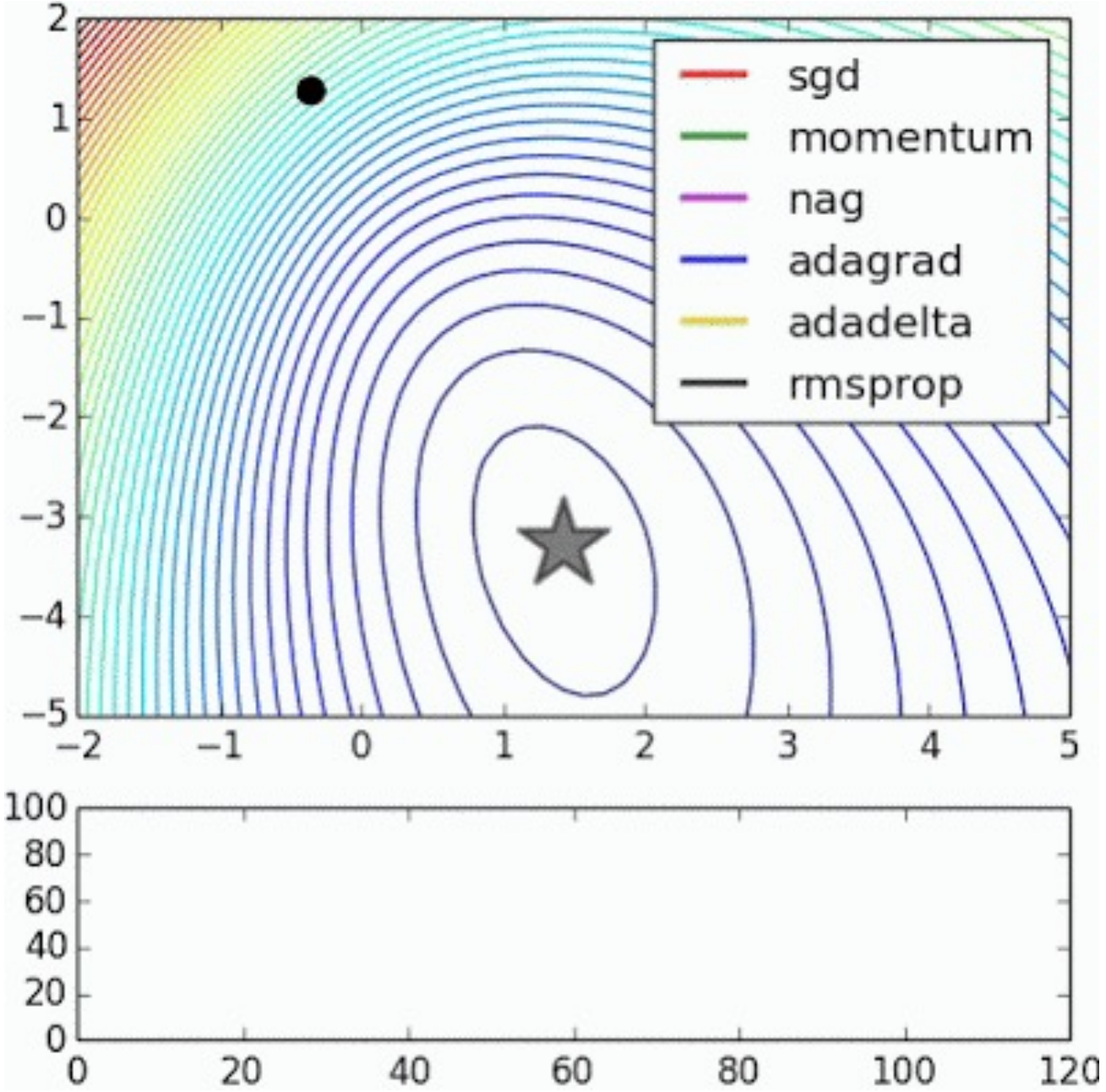


Image Source: Alec Radford https://twitter.com/alecrad
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.htmls

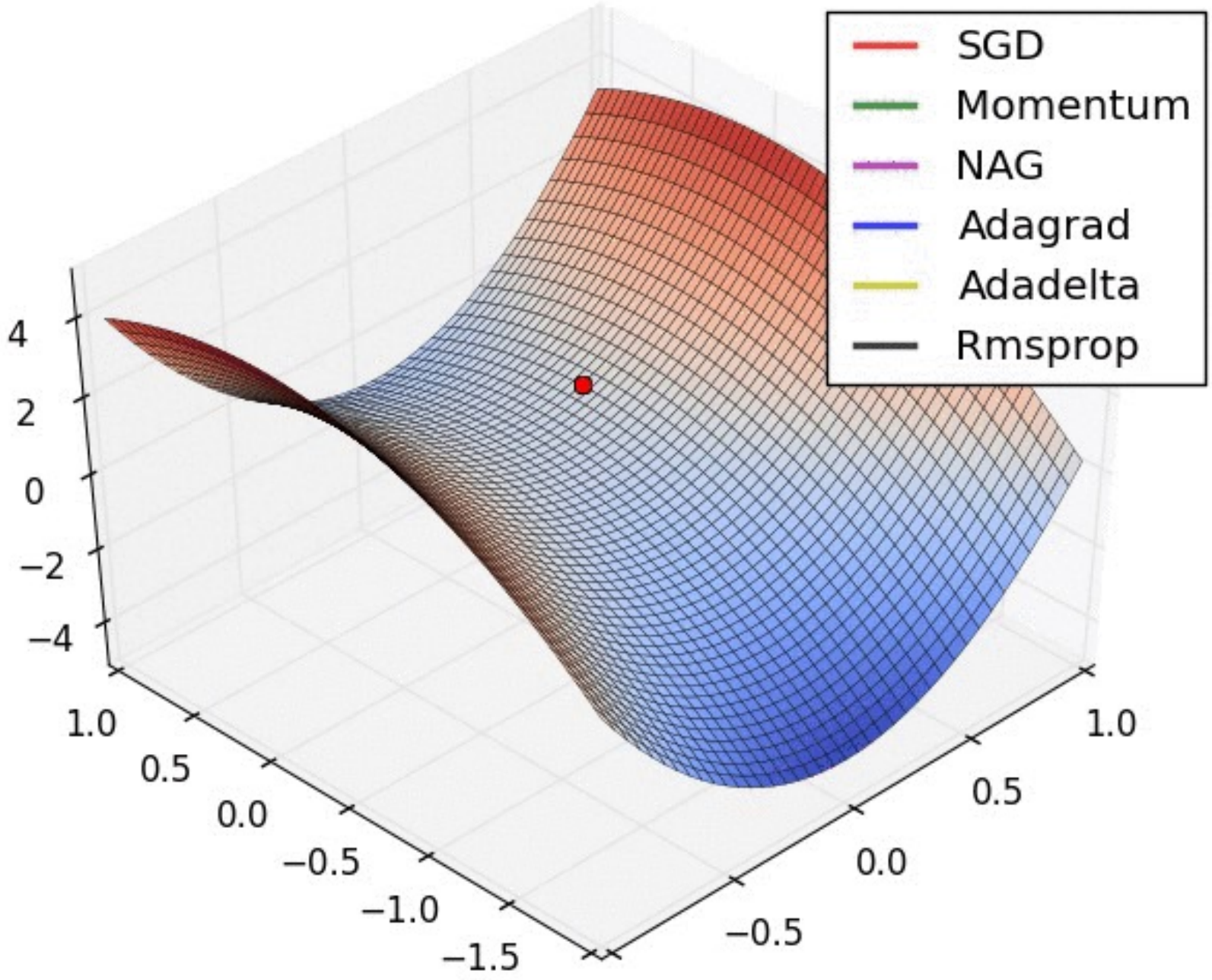


Image Source: Alec Radford https://twitter.com/alecrad
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

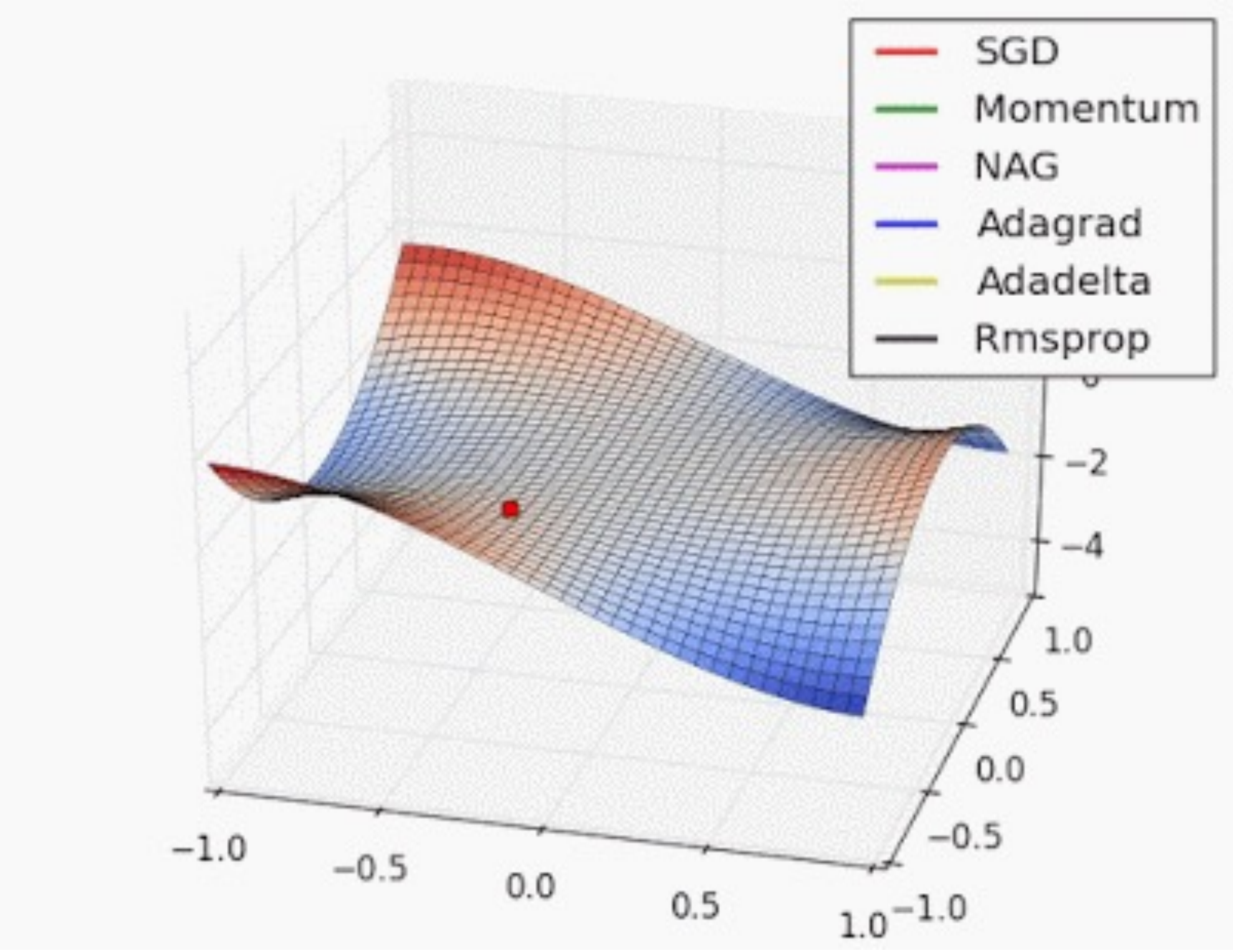
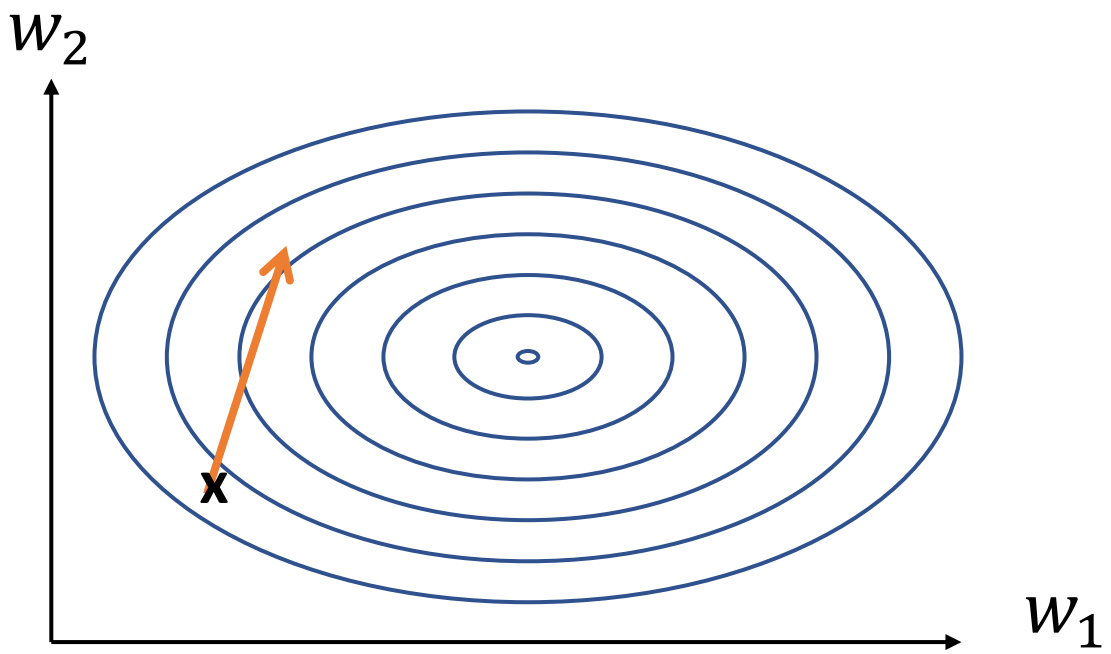
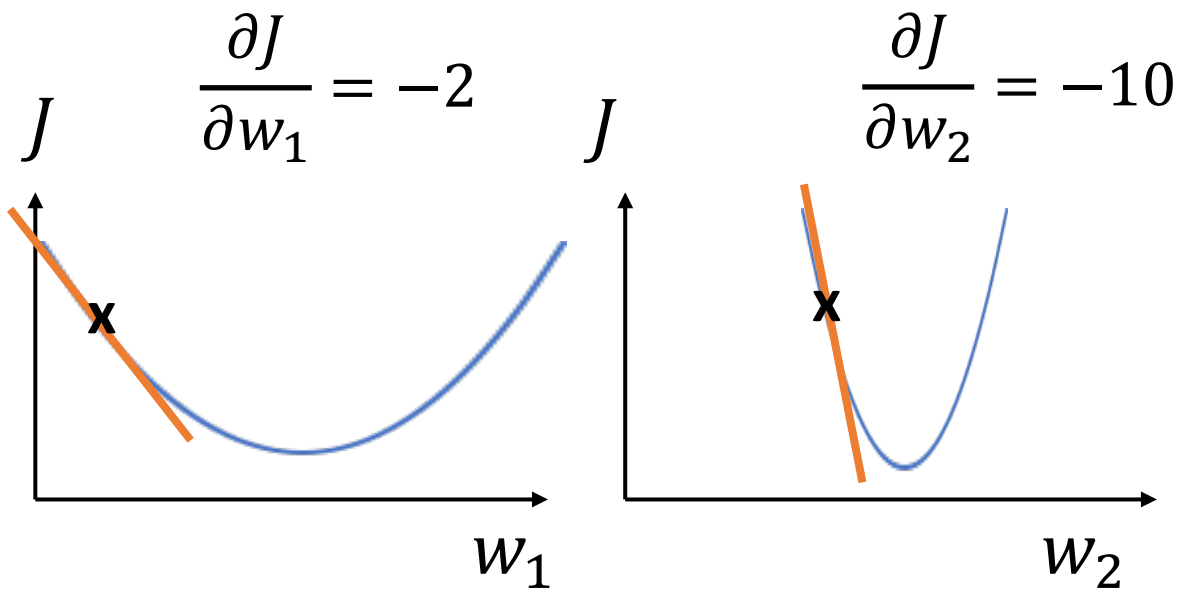


Image Source: Alec Radford https://twitter.com/alecrad
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Per-Parameter Adaptive Learning Rates

Recall when dimensions change at different rates





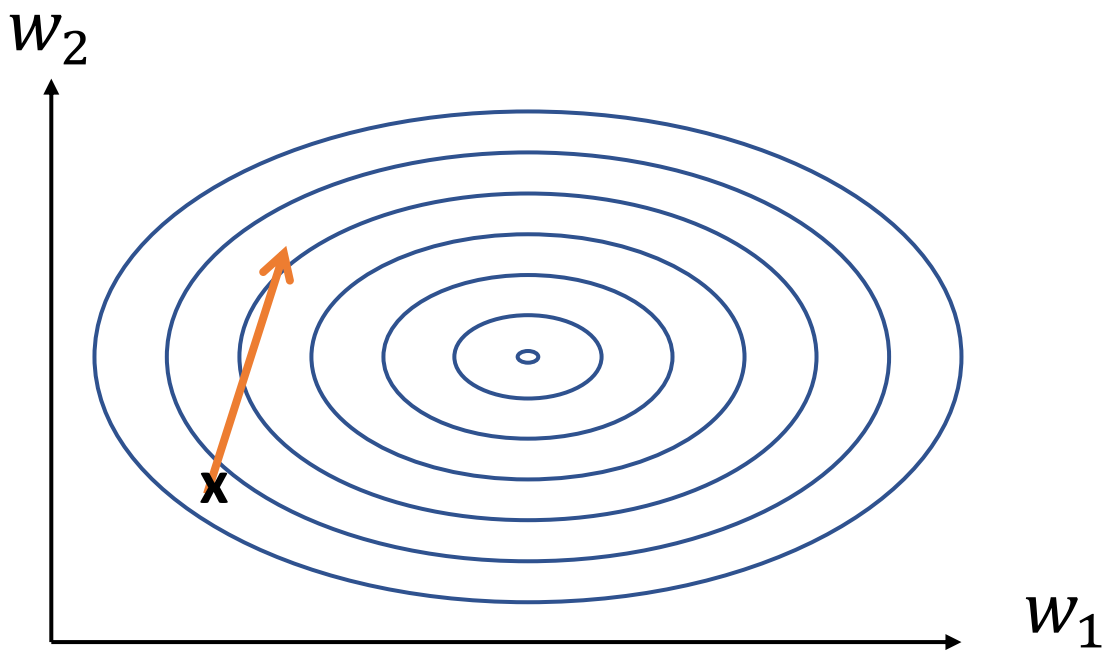
- Larger steps at steeper areas, and smaller steps at shallower areas

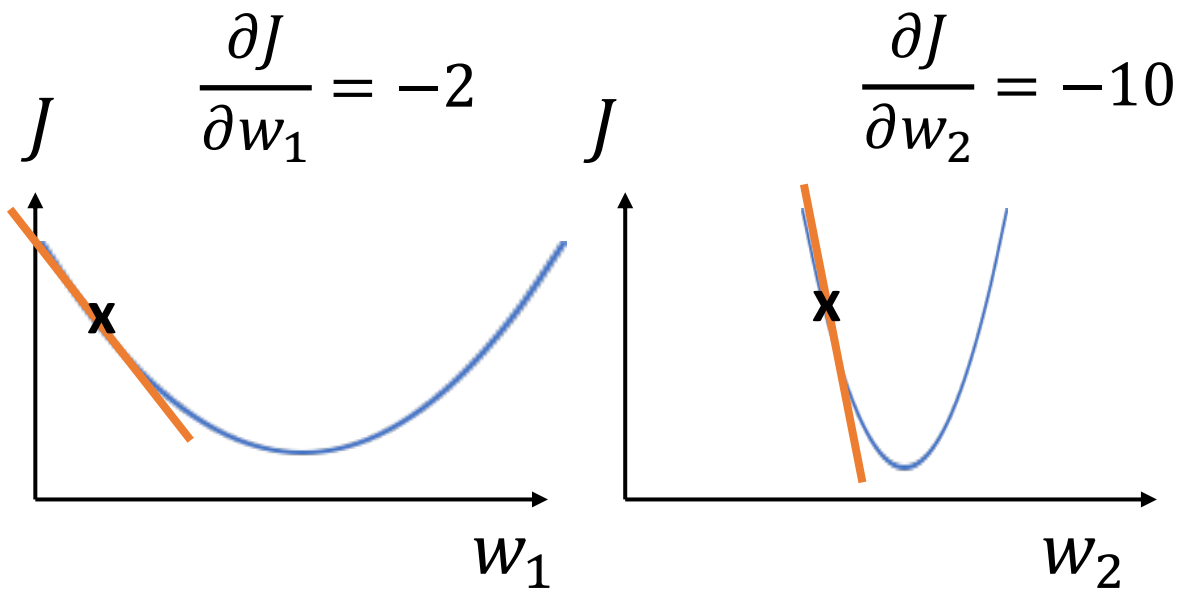
$$\frac{\partial J}{\partial w_1} \ll \frac{\partial J}{\partial w_2}$$

$$w_1 = w_1 - \alpha(-2)$$

$$w_2 = w_2 - \alpha(-10)$$

Recall when dimensions change at different rates





- Larger steps at steeper areas, and smaller steps at shallower areas

$$\frac{\partial J}{\partial w_1} \ll \frac{\partial J}{\partial w_2}$$

Adaptive Per-Parameter Learning Rates!

$$w_1 = w_1 - \alpha(-2)$$

$$w_2 = w_2 - \alpha(-10)$$

Adagrad

```
w = initialize()
grad_sq = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  grad_sq = grad_sq + dJ_dw * dJ_dw
  w = w - learning_rate*dJ_dw/sqrt(grad_sq)
```

Note: Keep a separate grad_sq term for each parameter

"Adaptive Subgradient Methods for Online and Stochastic Optimization", Duchi, Hazan, Singer, 2011, "http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf"

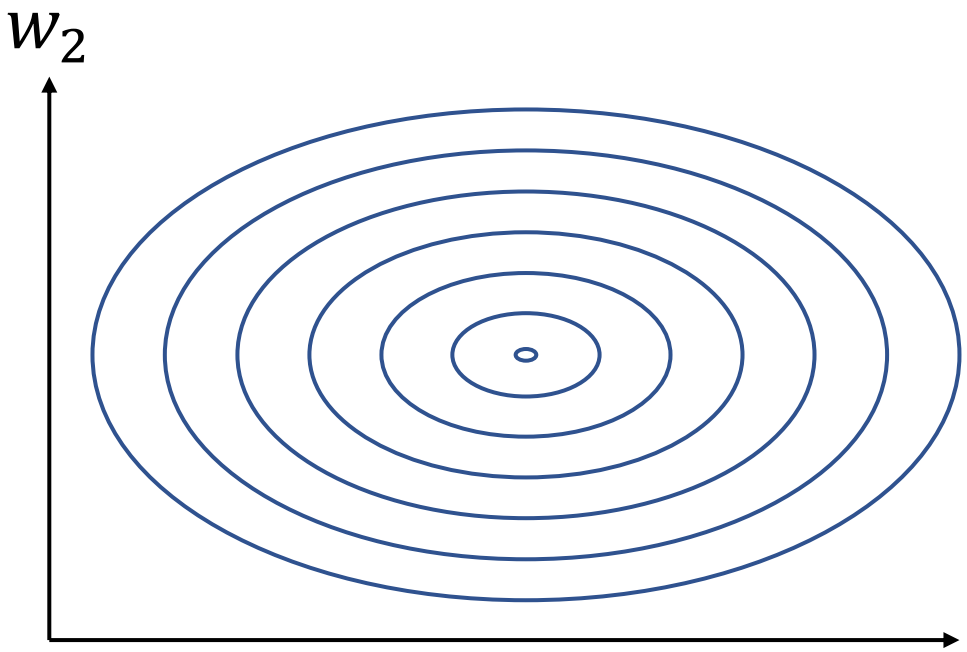
Adagrad

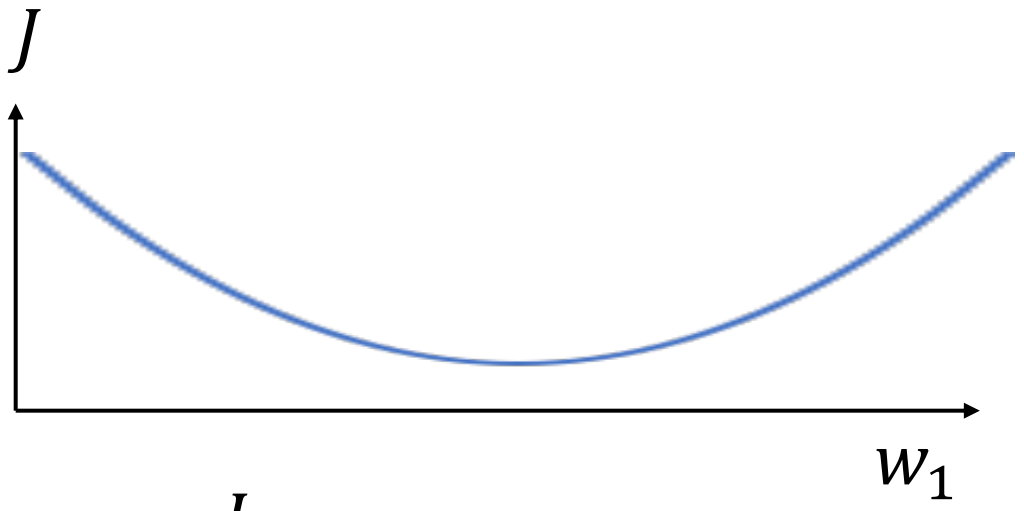
```
grad_sq = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  grad_sq = grad_sq + dJ_dw * dJ_dw
  w = w - learning_rate*dJ_dw/sqrt(grad_sq)
```

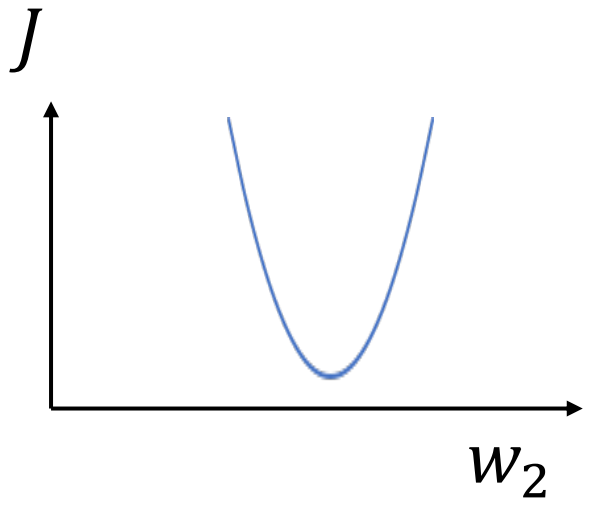
Intuition

- Square of gradients focuses on magnitude and not direction
- Dimensions moving through a region with large (steep) gradient will accumulate a larger value into grad_sq, and when you divide by this, you are making the update smaller
  - Dampen
- Dimensions moving through a region with small (shallow) gradient will accumulate a smaller value into grad_sq, and when you divide by this, you are making the update larger.
  - Accelerate

Adagrad Example



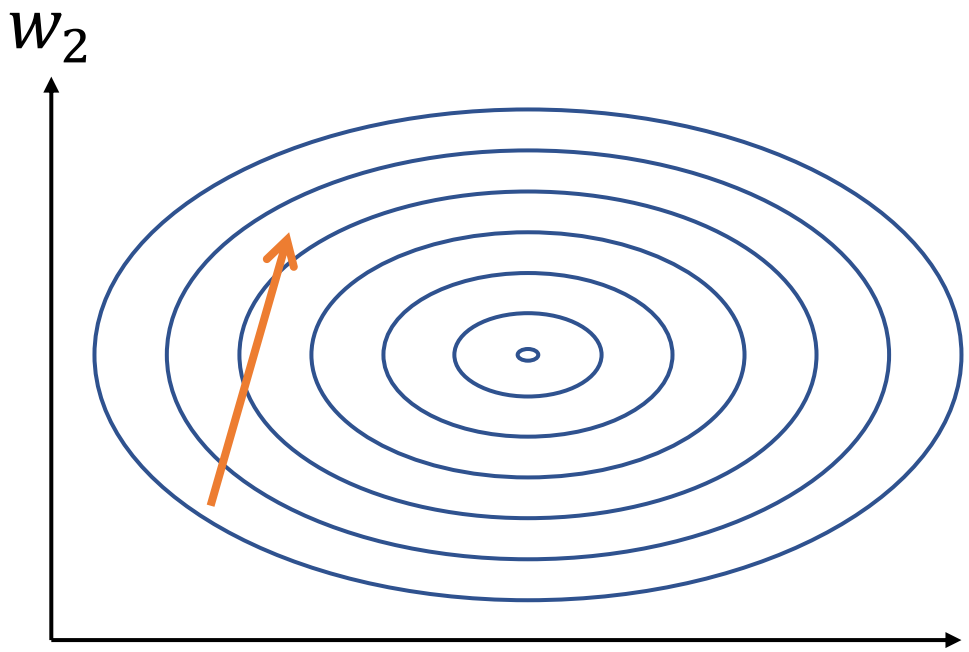


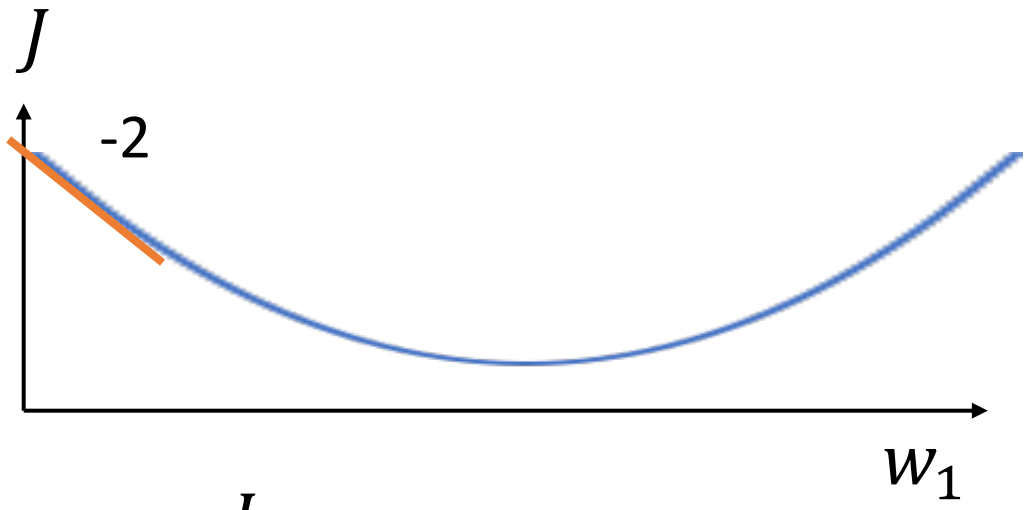


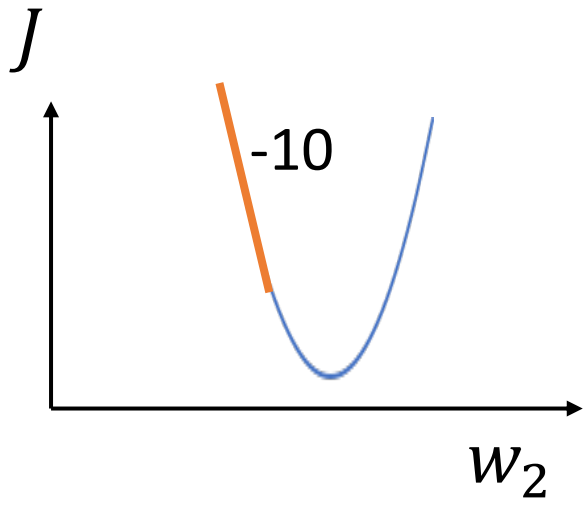
$$grad_{sq1} = 0$$

$$grad_{sq2} = 0$$

Adagrad Example



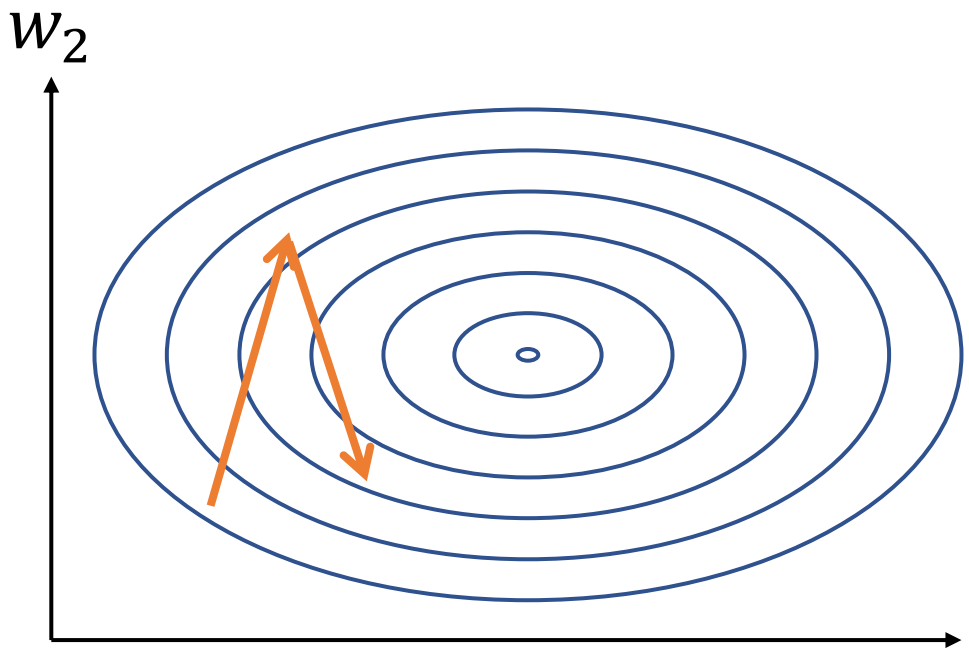


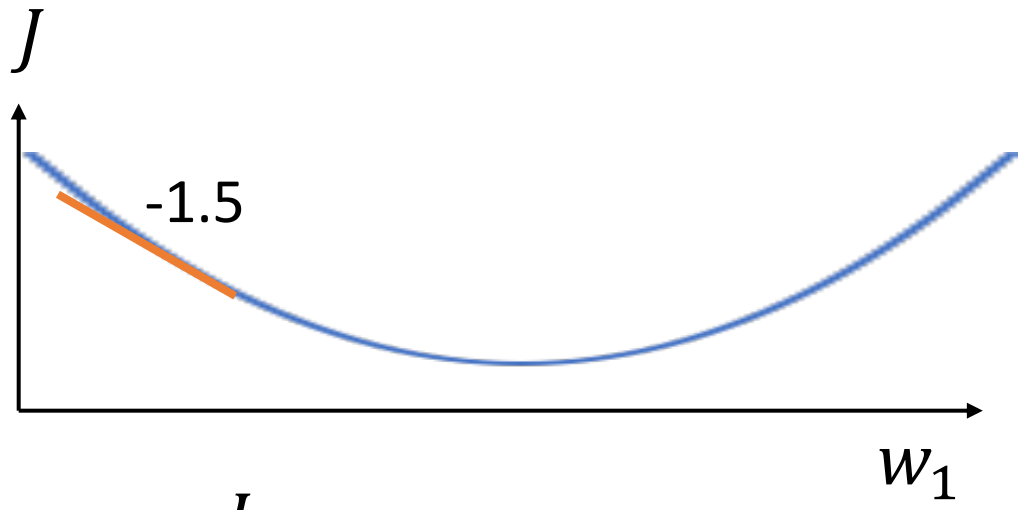


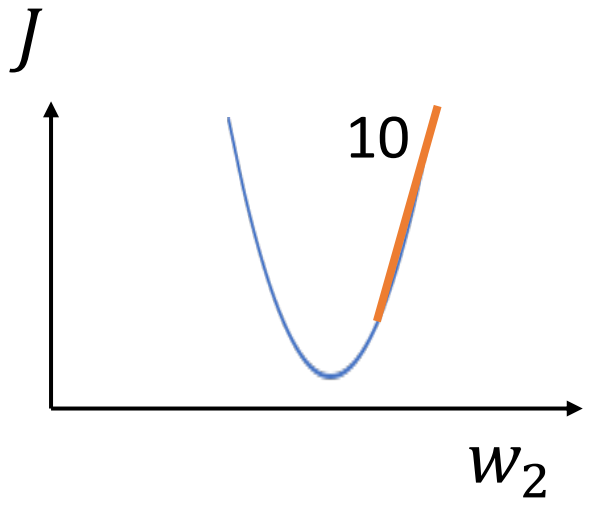
$$grad_{sq1} = 0 + (-2)(-2) = 4$$

$$grad_{sq2} = 0 + (-10)(-10) = 100$$

Adagrad Example



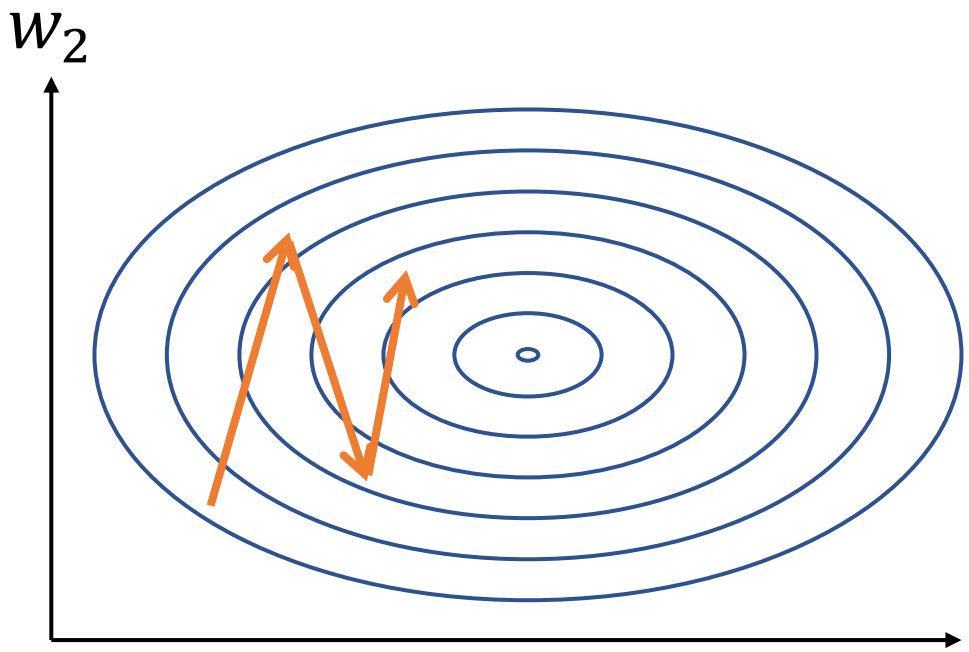


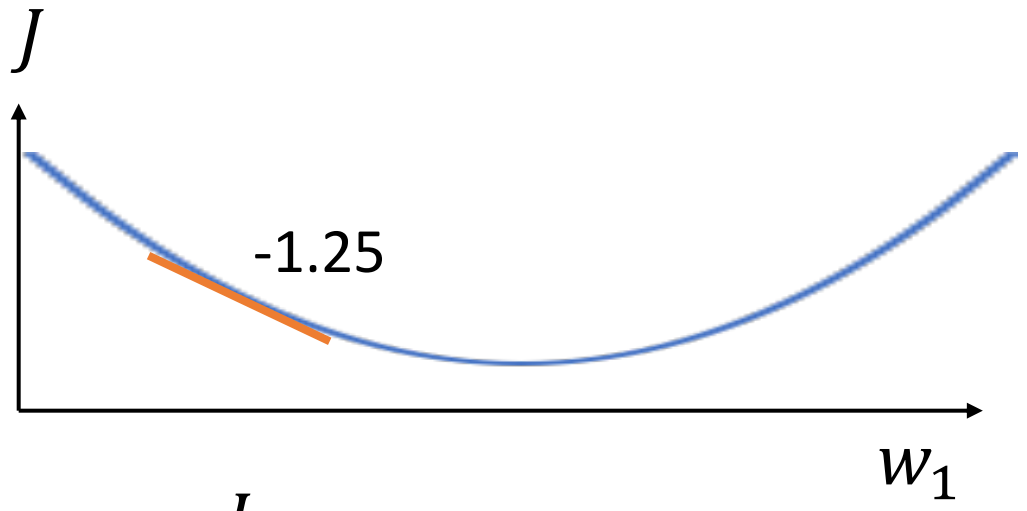


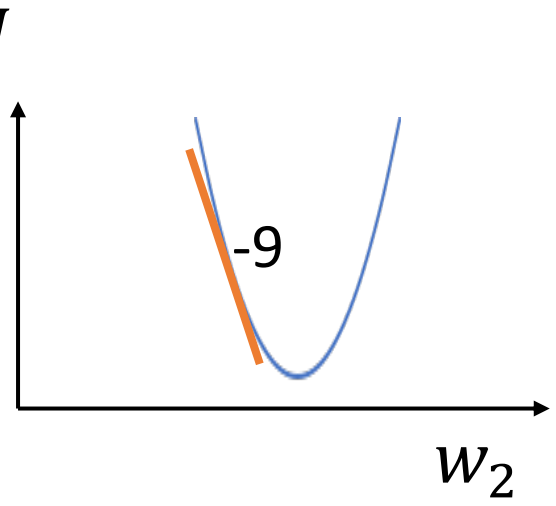
$$grad_{sq1} = 4 + (-1.5)(-1.5) = 6.25$$

$$grad_{sq2} = 100 + (-10)(-10) = 200$$

Adagrad Example



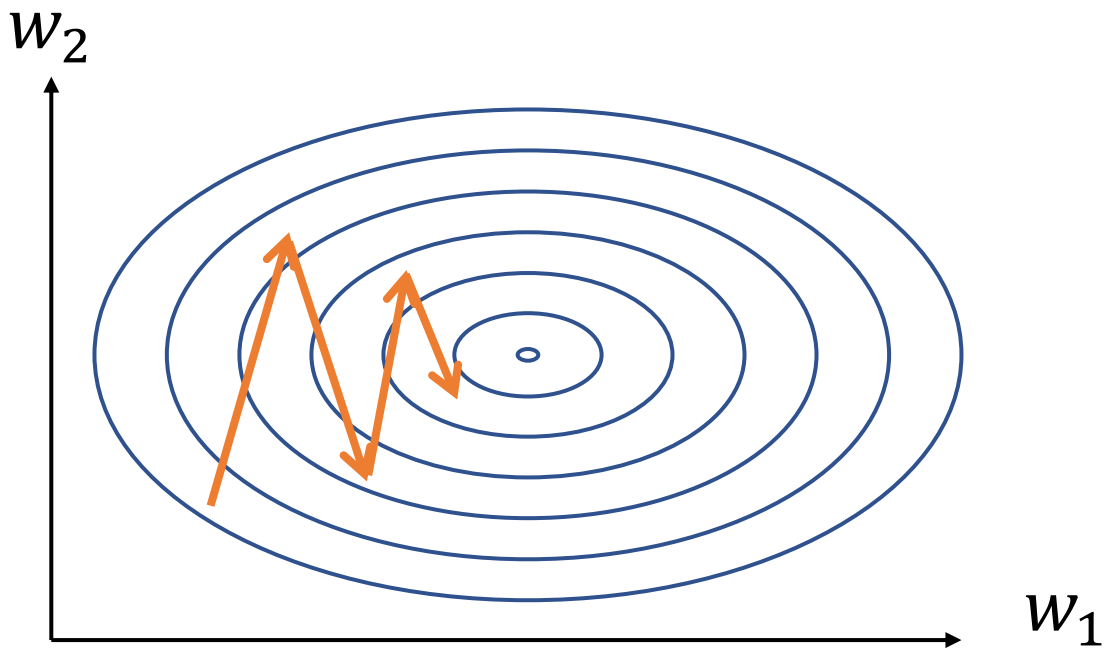


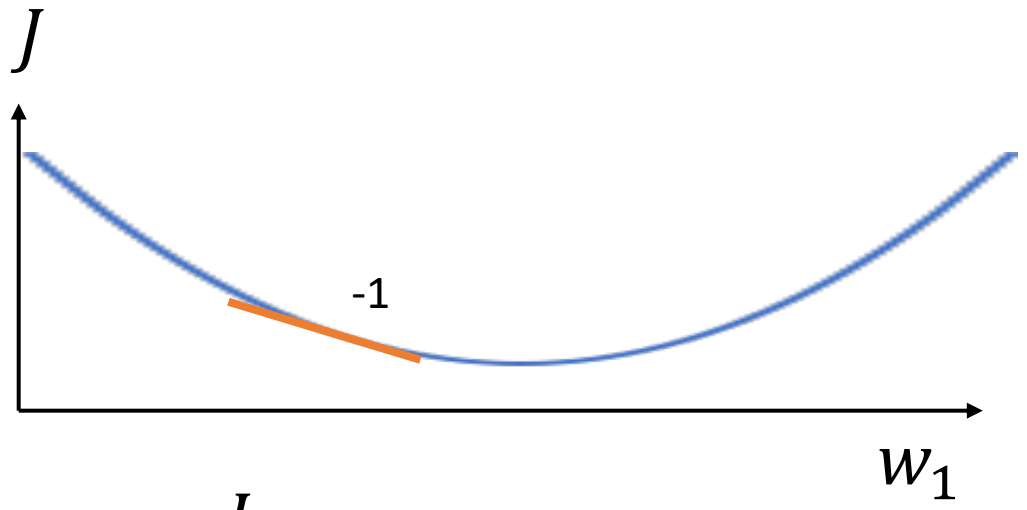


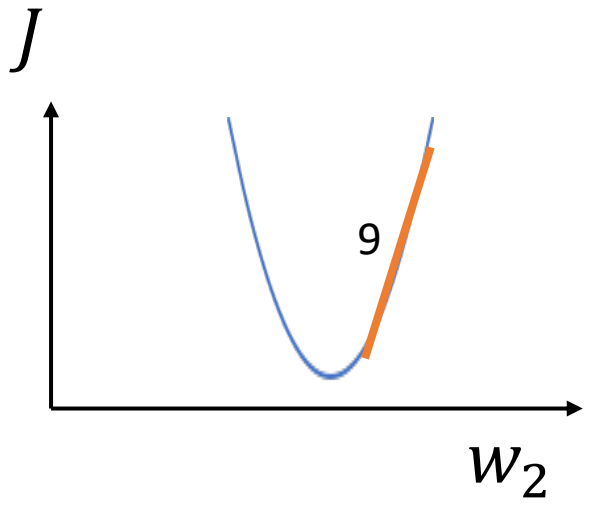
$$grad_{sq1} = 6.25 + (-1.25)(-1.25) = 7.8$$

$$grad_{sq2} = 200 + (-9)(-19) = 281$$

Adagrad Example



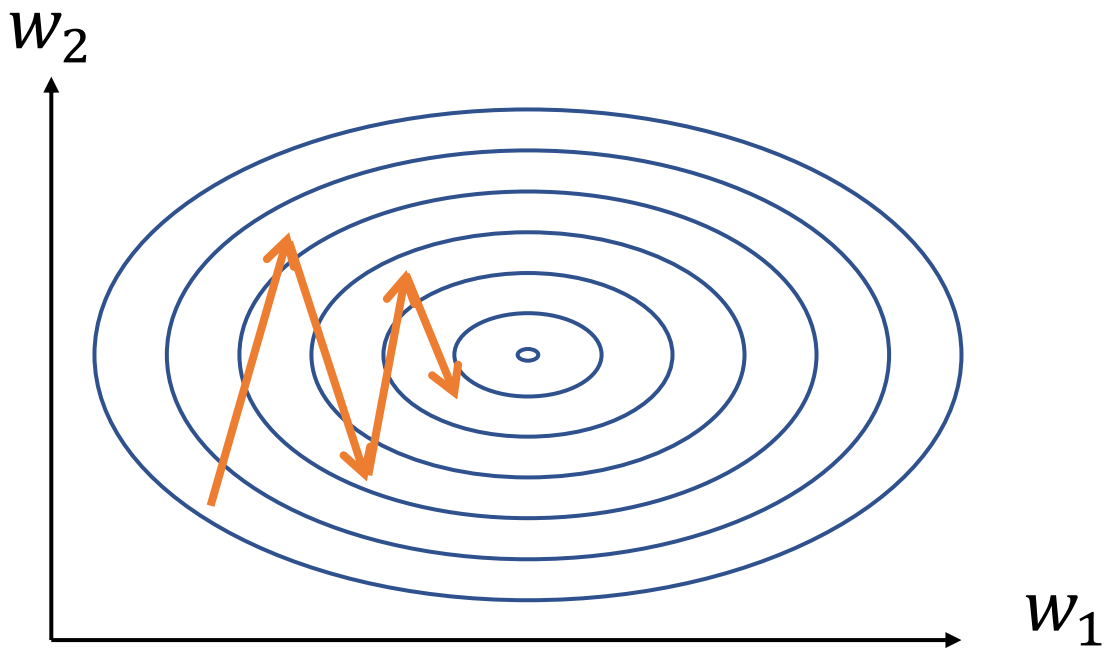


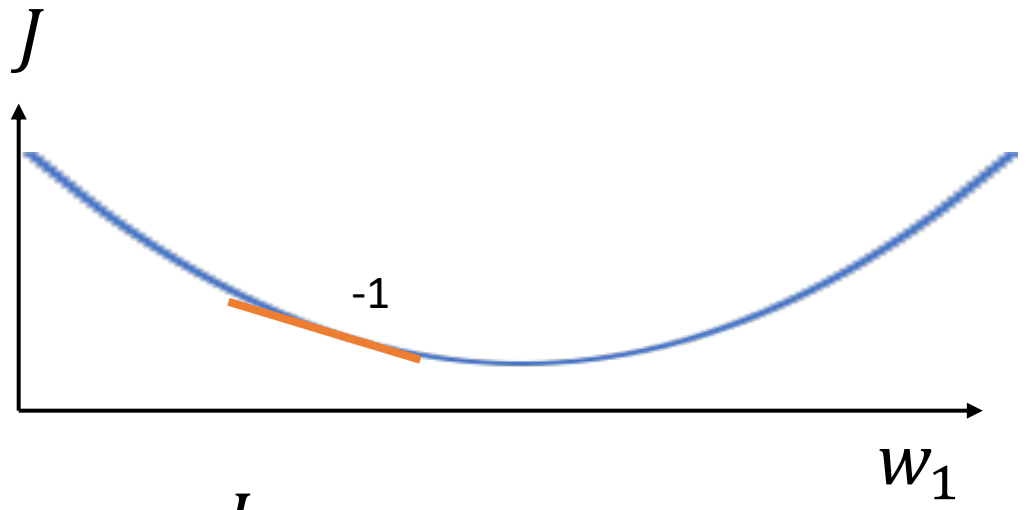


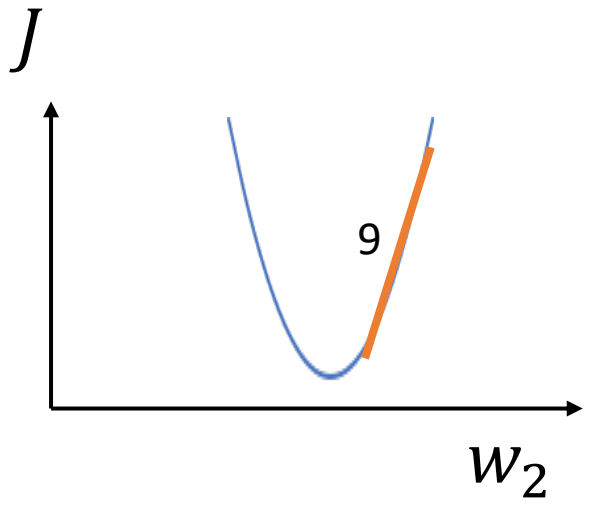
$$grad_{sq1} = 7.8 + (-1)(-1) = 8.8$$

$$grad_{sq2} = 281 + (9)(9) = 362$$

Adagrad Example







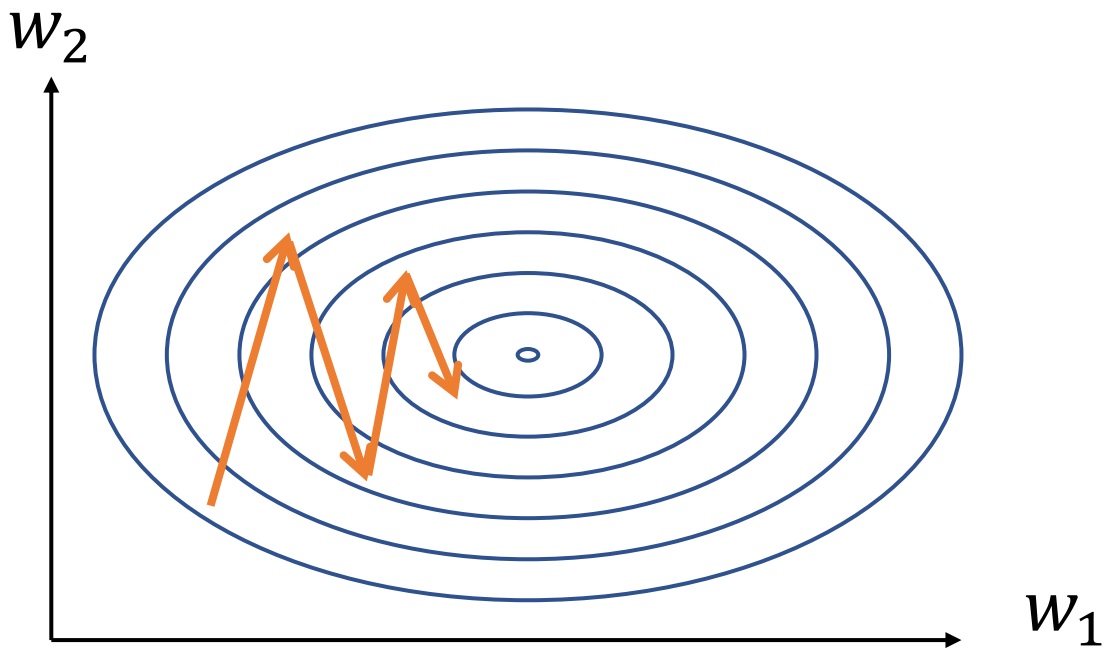
$$grad_{sq1} = 7.8 + (-1)(-1) = 8.8$$

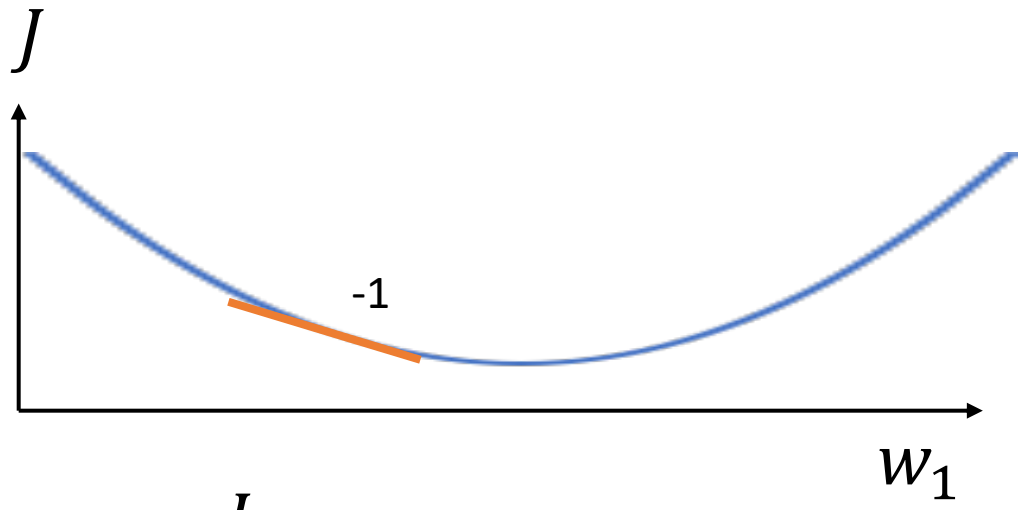
$$grad_{sq2} = 281 + (9)(9) = 362$$

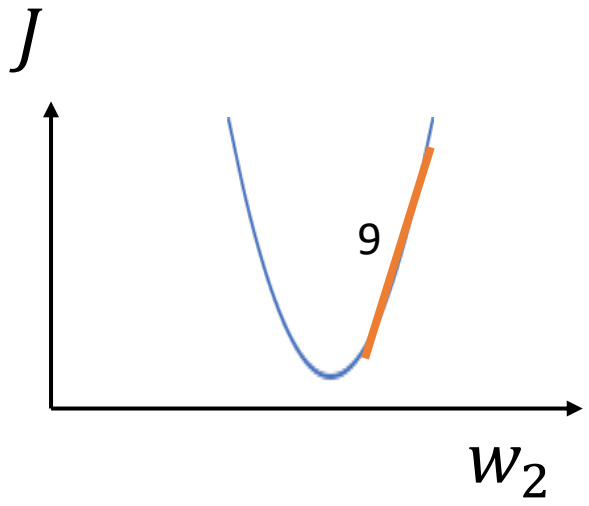
$$w_1 = w_1 - \alpha(-1)/3$$

$$w_2 = w_2 - \alpha(9)/19$$

Adagrad Example







$$grad_{sq1} = 7.8 + (-1)(-1) = 8.8$$

$$grad_{sq2} = 281 + (9)(9) = 362$$

$$w_1 = w_1 - \alpha(-1)/3$$

$$w_2 = w_2 - \alpha(9)/19$$

Effectively adjusting learning rate for each parameter

Adagrad

```
grad_sq = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  grad_sq = grad_sq + dJ_dw * dJ_dw
  w = w - learning_rate*dJ_dw/sqrt(grad_sq)
```

- Problem with Adagrad?

Adagrad

```
grad_sq = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  grad_sq = grad_sq + dJ_dw * dJ_dw
  w = w - learning_rate*dJ_dw/sqrt(grad_sq)
```

- Problem with Adagrad?
  No decay of grad_sq term. Just gets bigger and bigger!
- RMSProp and Adadelta solve this in similar way.

RMSProp

```
grad_sq = 0
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  grad_sq = beta*grad_sq + (1-beta)*dJ_dw*dJ_dw
  w = w - learning_rate*dJ_dw/sqrt(grad_sq)
```

- Use exponentially weighted average of the square of the gradients
- Recall: this is a moving average (i.e. decay of historical terms)

Hinton, http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

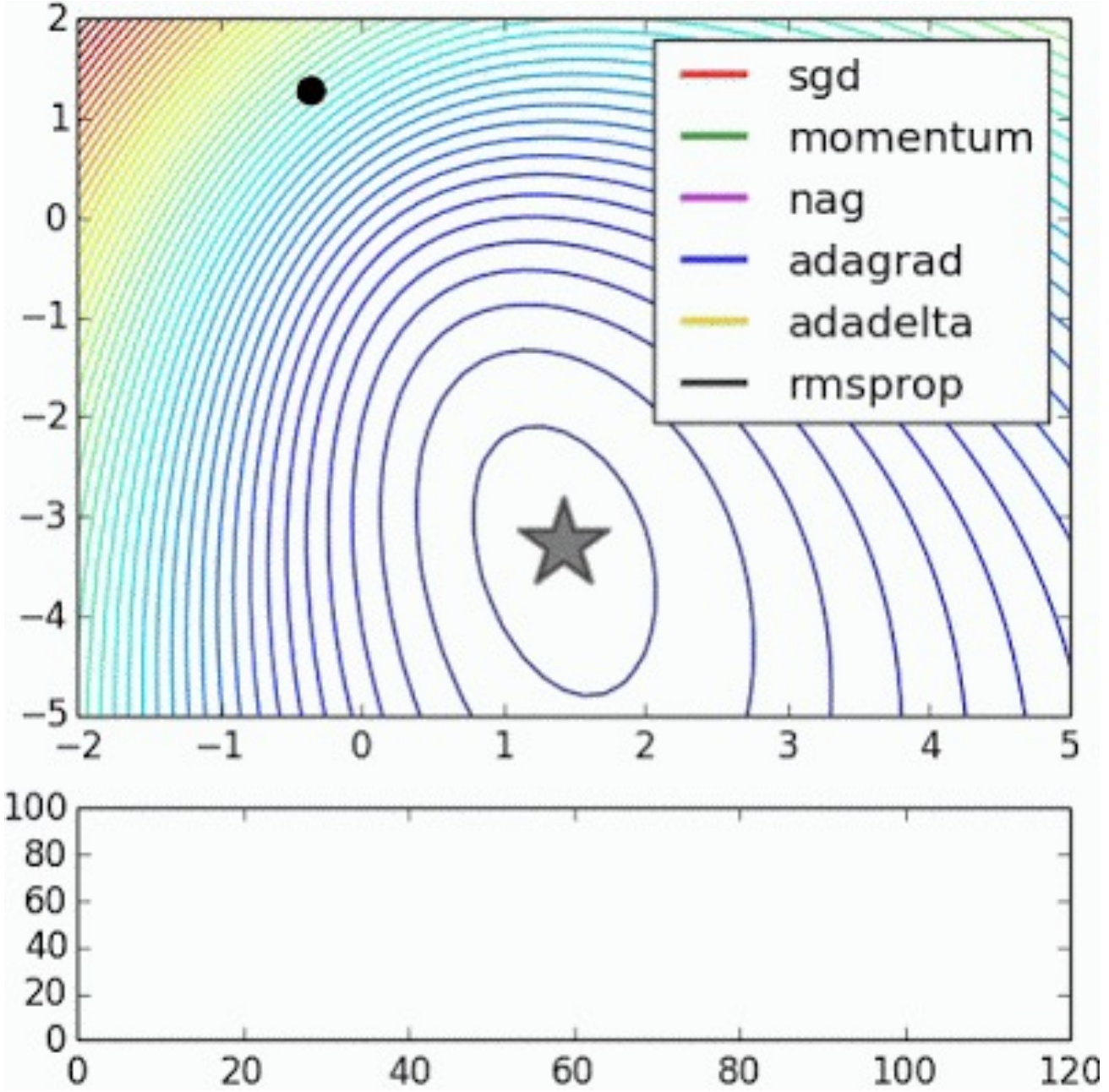


Image Source: Alec Radford https://twitter.com/alecrad
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.htmls

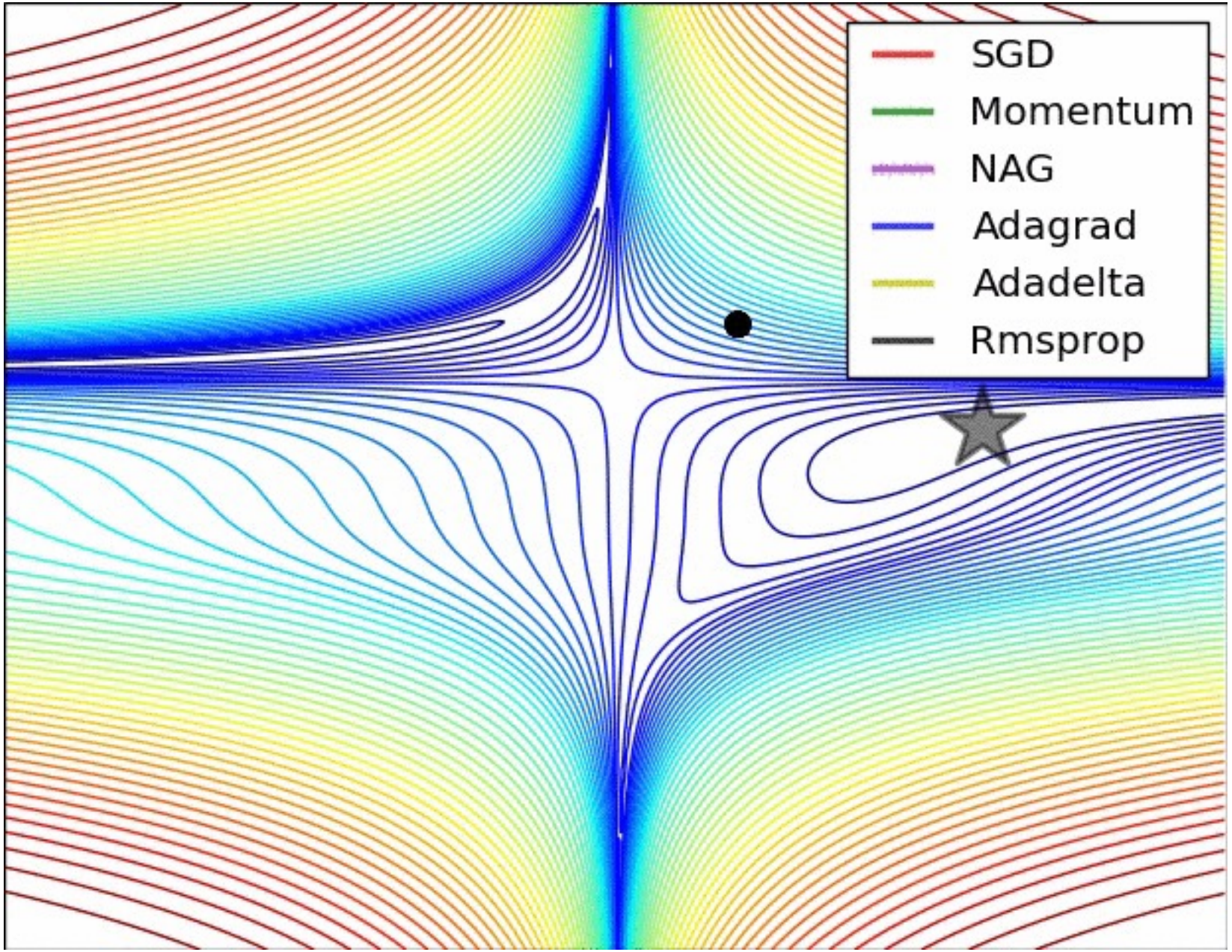


Image Source: Alec Radford https://twitter.com/alecrad
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

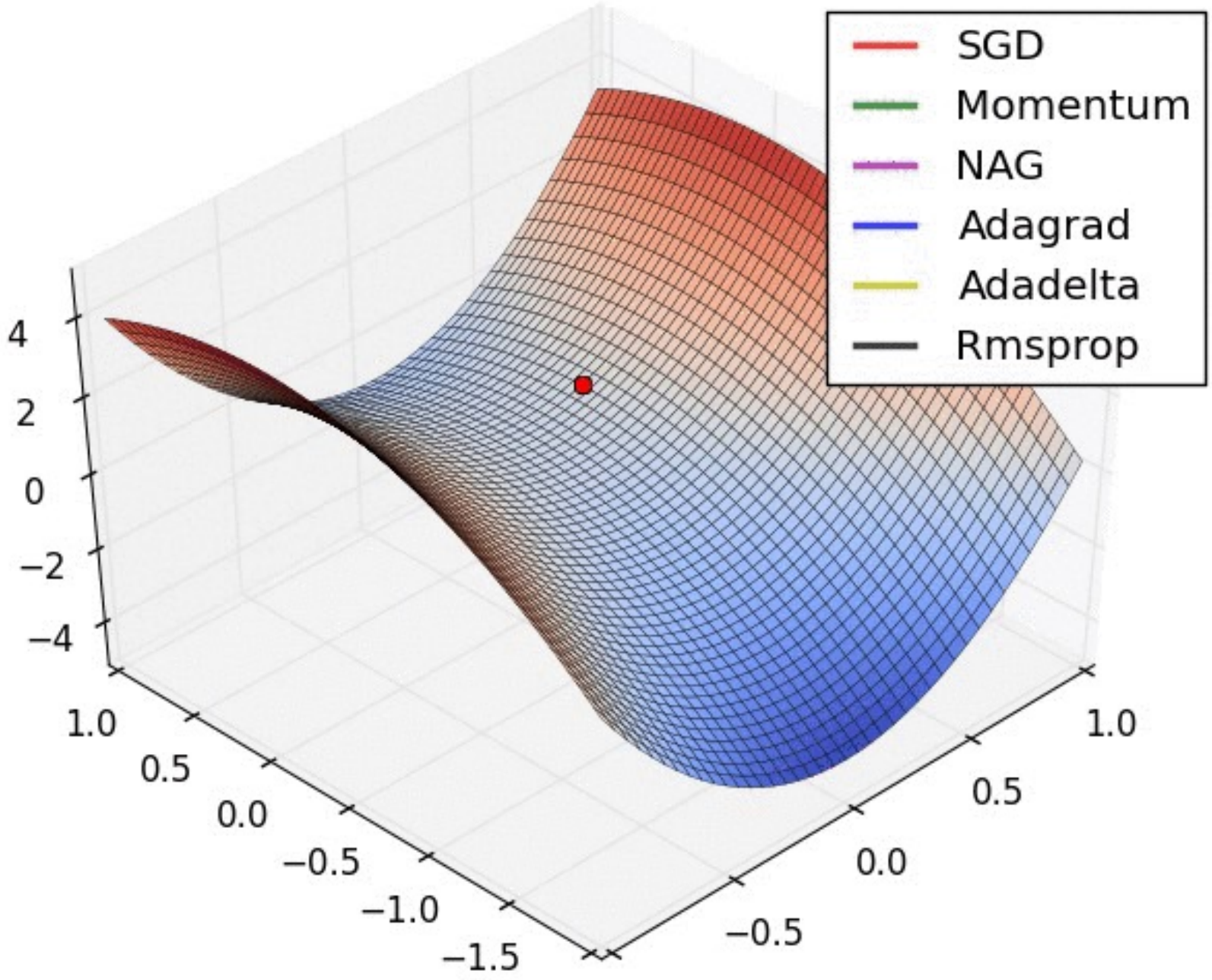


Image Source: Alec Radford https://twitter.com/alecrad
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

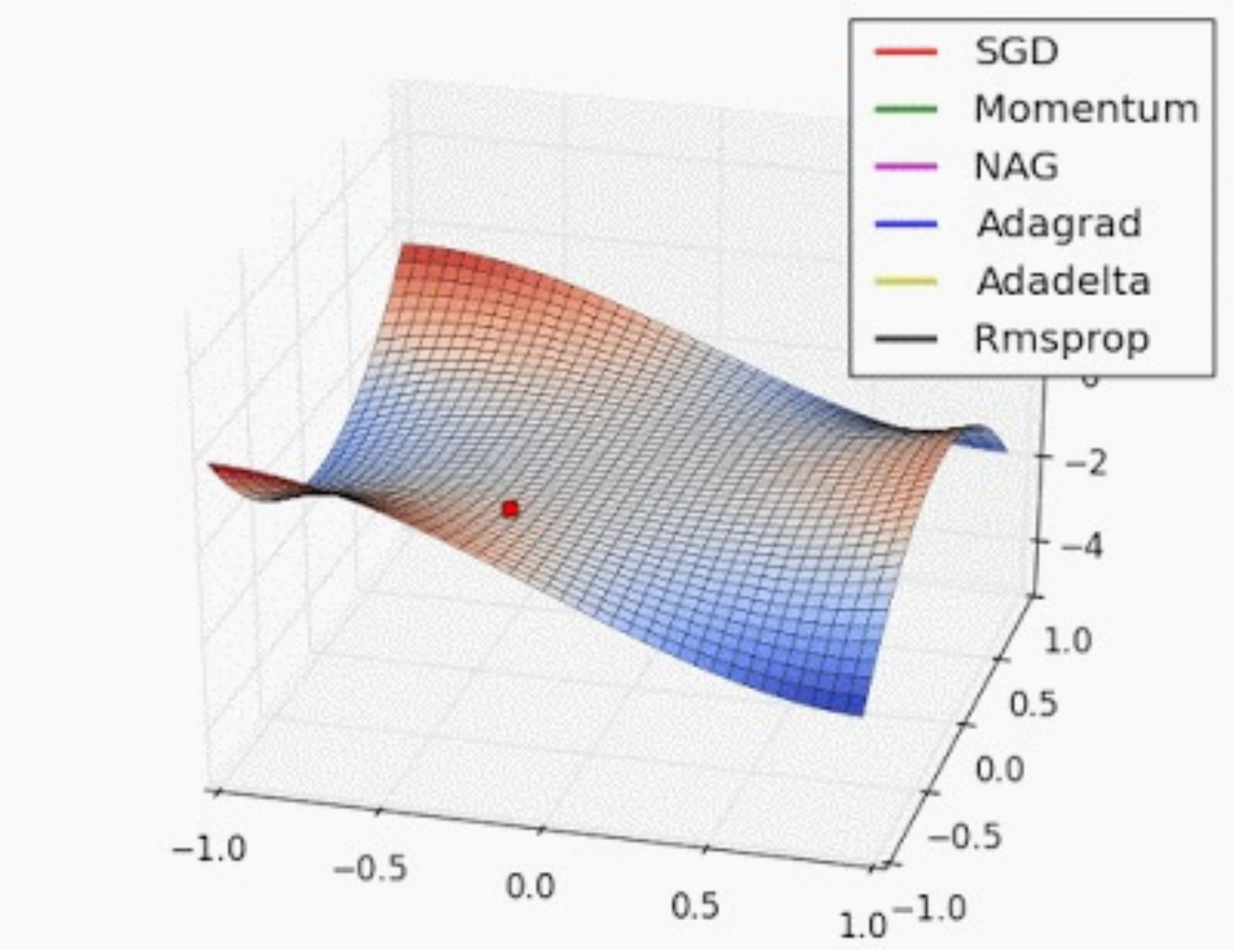


Image Source: Alec Radford https://twitter.com/alecrad
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Adam

- Combines RMSProp and Momentum
- Work well across a wide variety of deep learning problems
- A good default choice for optimizer

"Adam: A Method for Stochastic Optimization", Kingma, Ba, 2014, https://arxiv.org/abs/1412.6980v8

Adam

```
w = initialize()
v1 = 0 # Momentum

for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v1 = beta1*v1 + (1-beta1)*dJ_dw

  w = w - learning_rate*v1
```

Adam

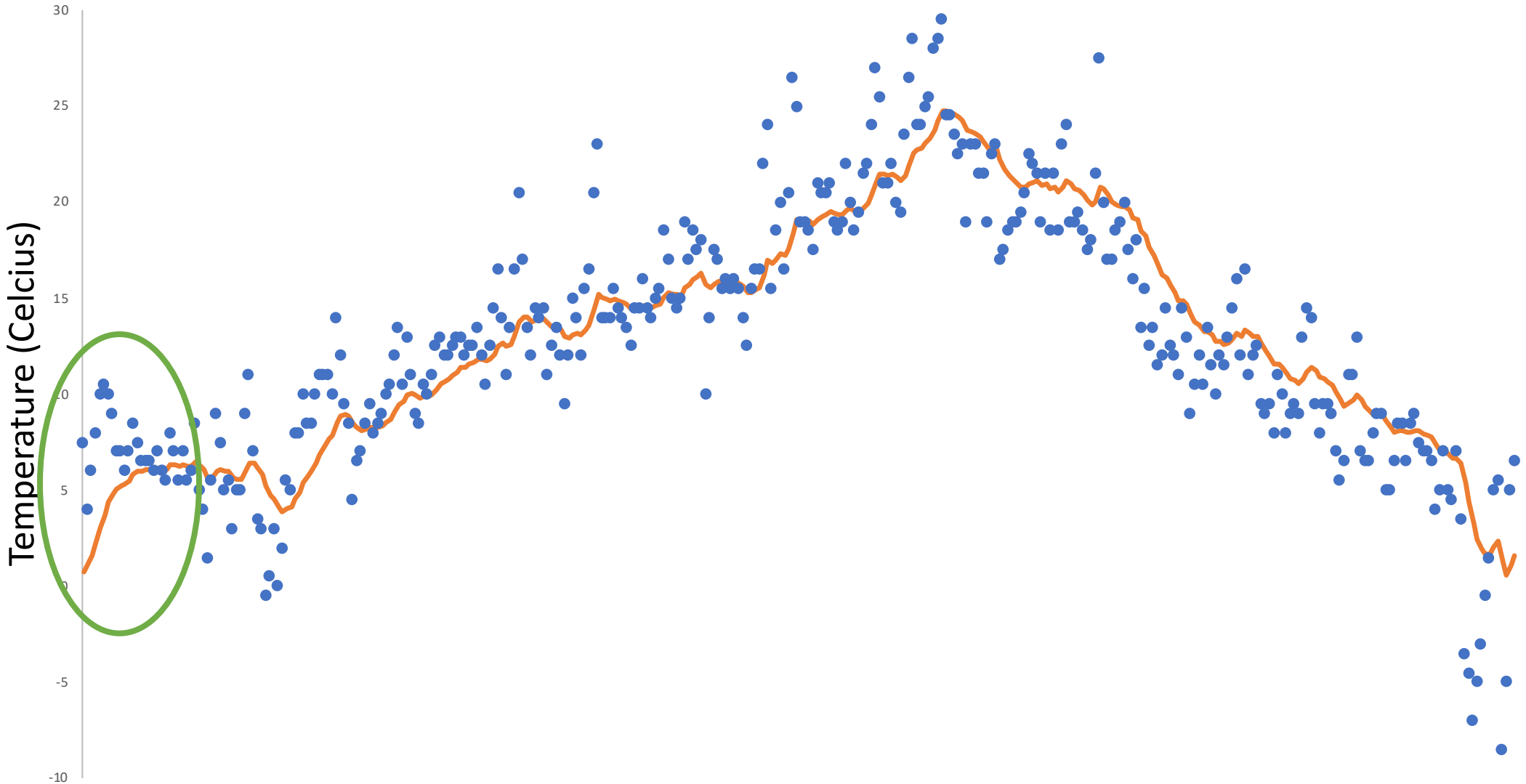
```
w = initialize()
v1 = 0 # Momentum
v2 = 0 # RMSProp
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v1 = beta1*v1 + (1-beta1)*dJ_dw
  v2 = beta2*v1 + (1-beta2)*dJ_dw*dJ_dw
  w = w - learning_rate*v1/sqrt(v2)
```

Adam

```
w = initialize()
v1 = 0 # Momentum
v2 = 0 # RMSProp
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v1 = beta1*v1 + (1-beta1)*dJ_dw
  v2 = beta2*v1 + (1-beta2)*dJ_dw*dJ_dw
  w = w - learning_rate*v1/sqrt(v2)
```

One more thing to take care of

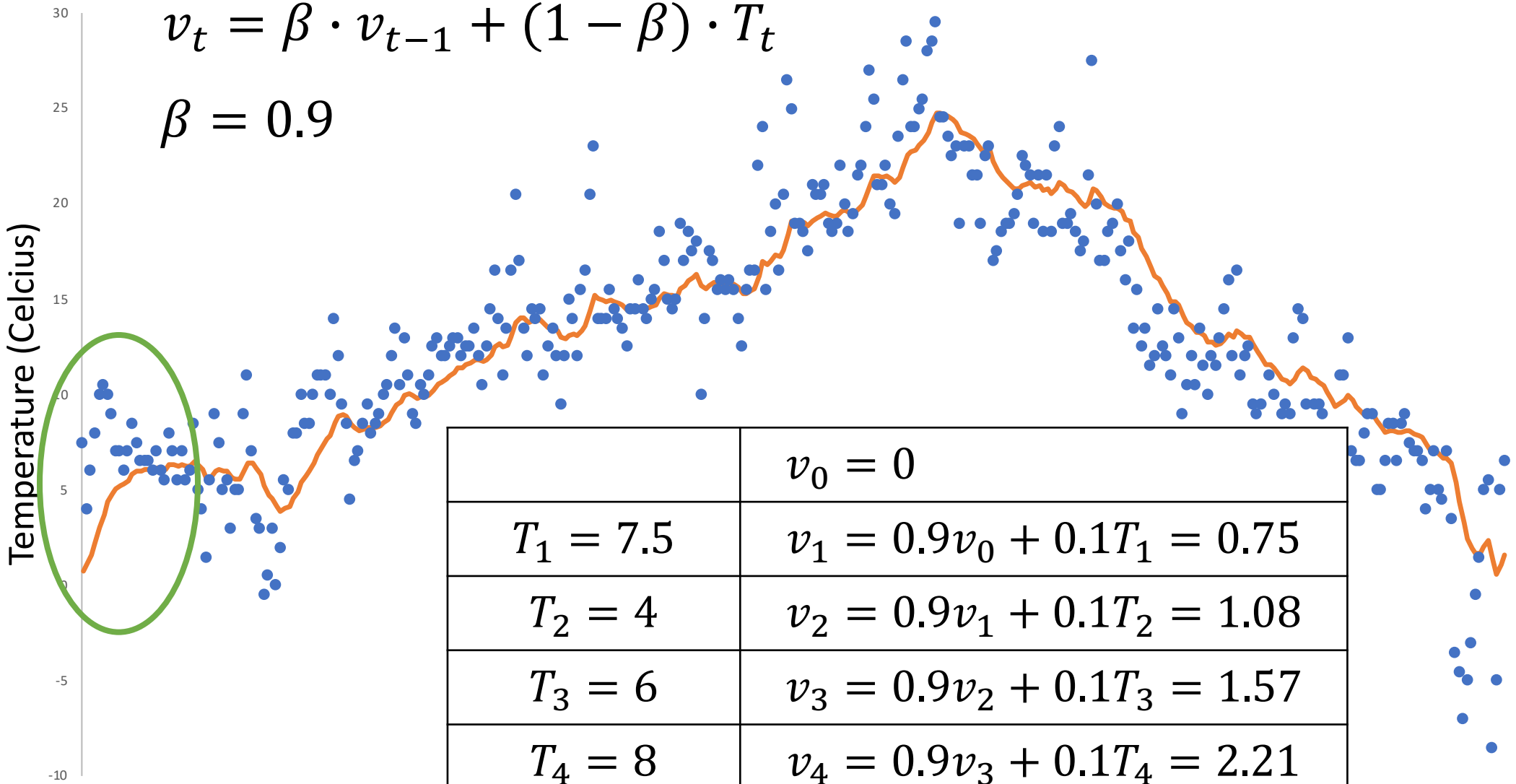
Recall: Exponentially Weighted Average



Recall: Exponentially Weighted Average

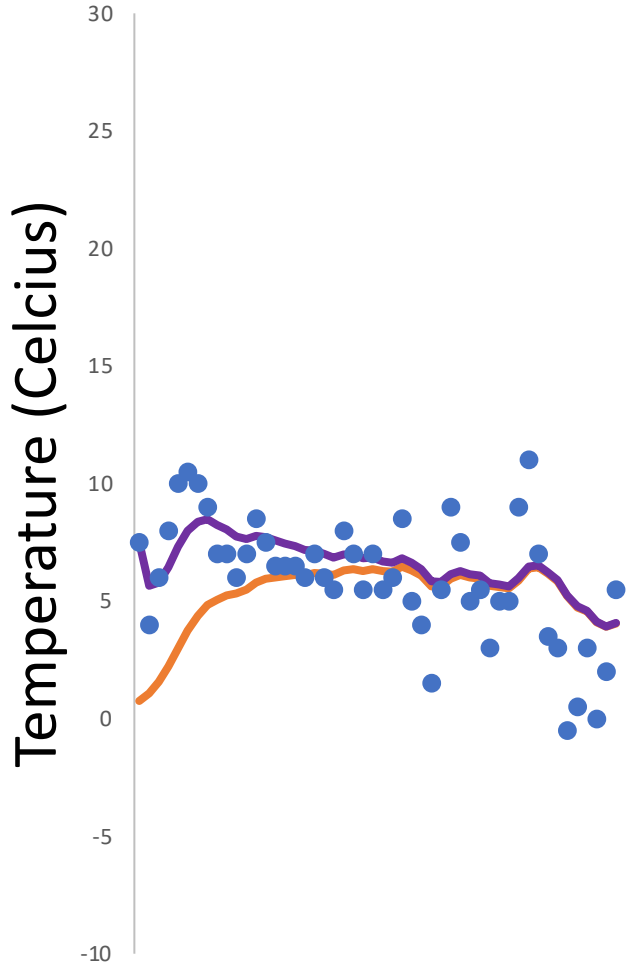
$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot T_t$$

$$\beta = 0.9$$



	$v_0 = 0$
$T_1 = 7.5$	$v_1 = 0.9v_0 + 0.1T_1 = 0.75$
$T_2 = 4$	$v_2 = 0.9v_1 + 0.1T_2 = 1.08$
$T_3 = 6$	$v_3 = 0.9v_2 + 0.1T_3 = 1.57$
$T_4 = 8$	$v_4 = 0.9v_3 + 0.1T_4 = 2.21$

Bias Correction for Exponentially Weighted Averages

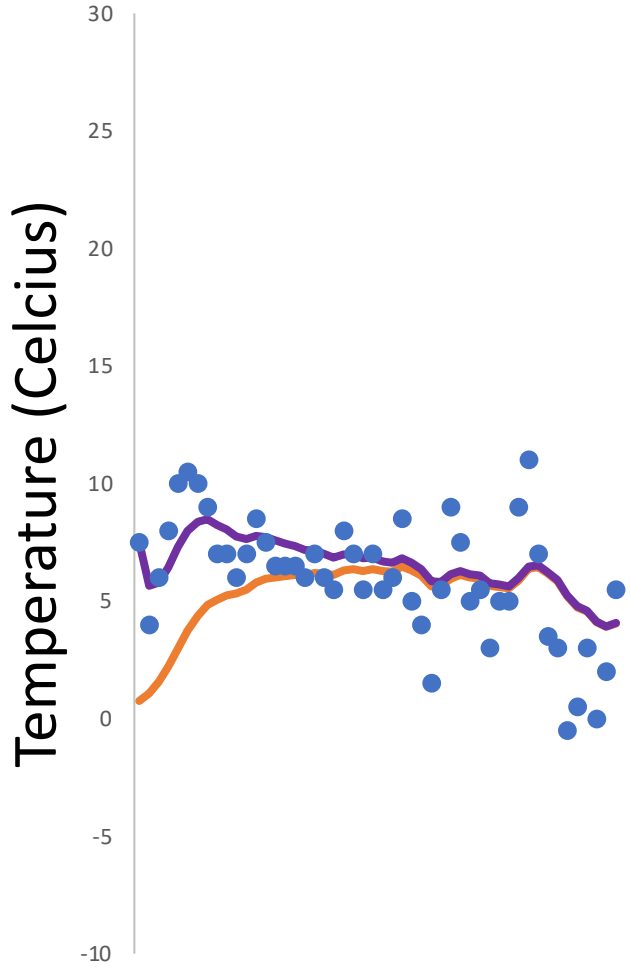


$$v_{(t)\_biased} = \beta \cdot v_{(t-1)\_biased} + (1-\beta) \cdot T_t$$

$$v_t = \frac{v_{(t)\_biased}}{1-\beta^t}$$

Denominator approaches 1 as t increases

Bias Correction for Exponentially Weighted Averages



$$v_{(t)\_biased} = \beta \cdot v_{(t-1)\_biased} + (1-\beta) \cdot T_t$$

$$v_t = \frac{v_{(t)\_biased}}{1-\beta^t}$$

Denominator approaches 1 as t increases

- Use biased values in moving average
- Use corrected version when doing gradient descent parameter update

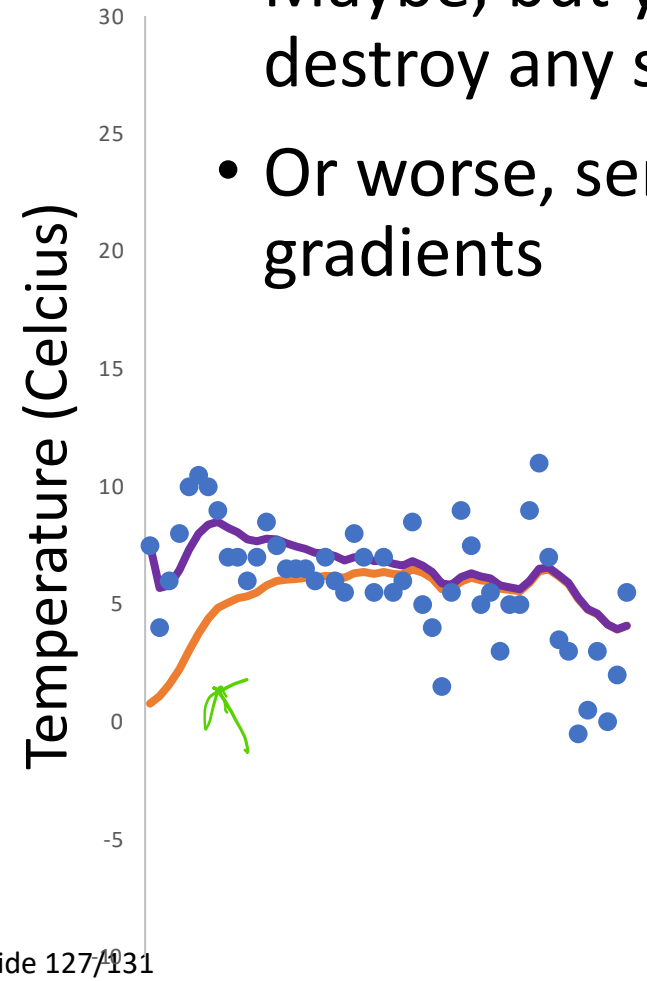
Adam

```
w = initialize()
v1_biased = 0 # Momentum
v2_biased = 0 # RMSProp
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v1_biased = beta1*v1_biased + (1-beta1)*dJ_dw
  v2_biased = beta2*v2_biased + (1-beta2)*dJ_dw*dJ_dw
  v1 = v1_biased / (1 – beta1**(i+1))
  v2 = v2_biased / (1 – beta2**(i+1))
  w = w - learning_rate*v1/sqrt(v2)
```

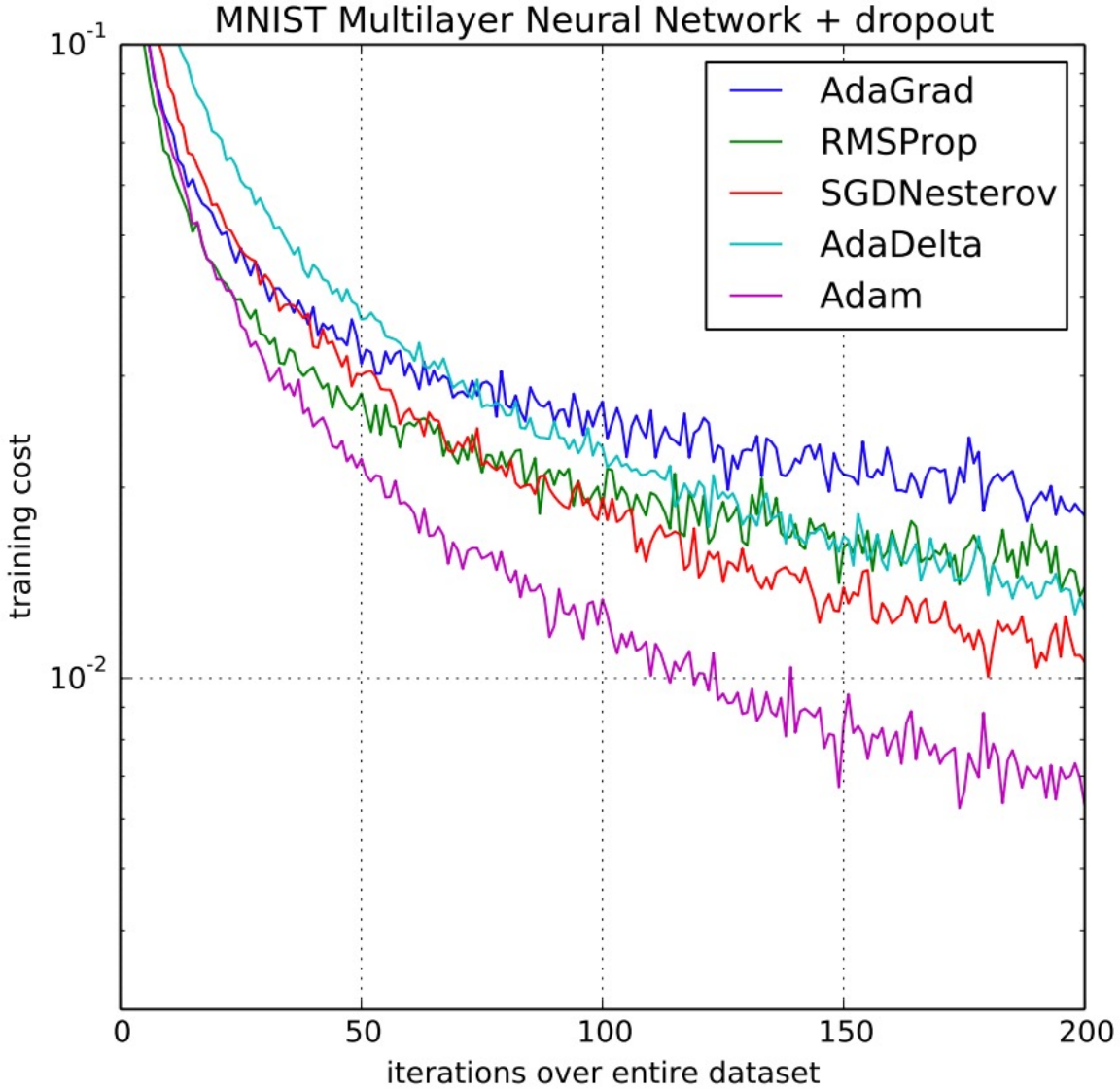
Bias Correction

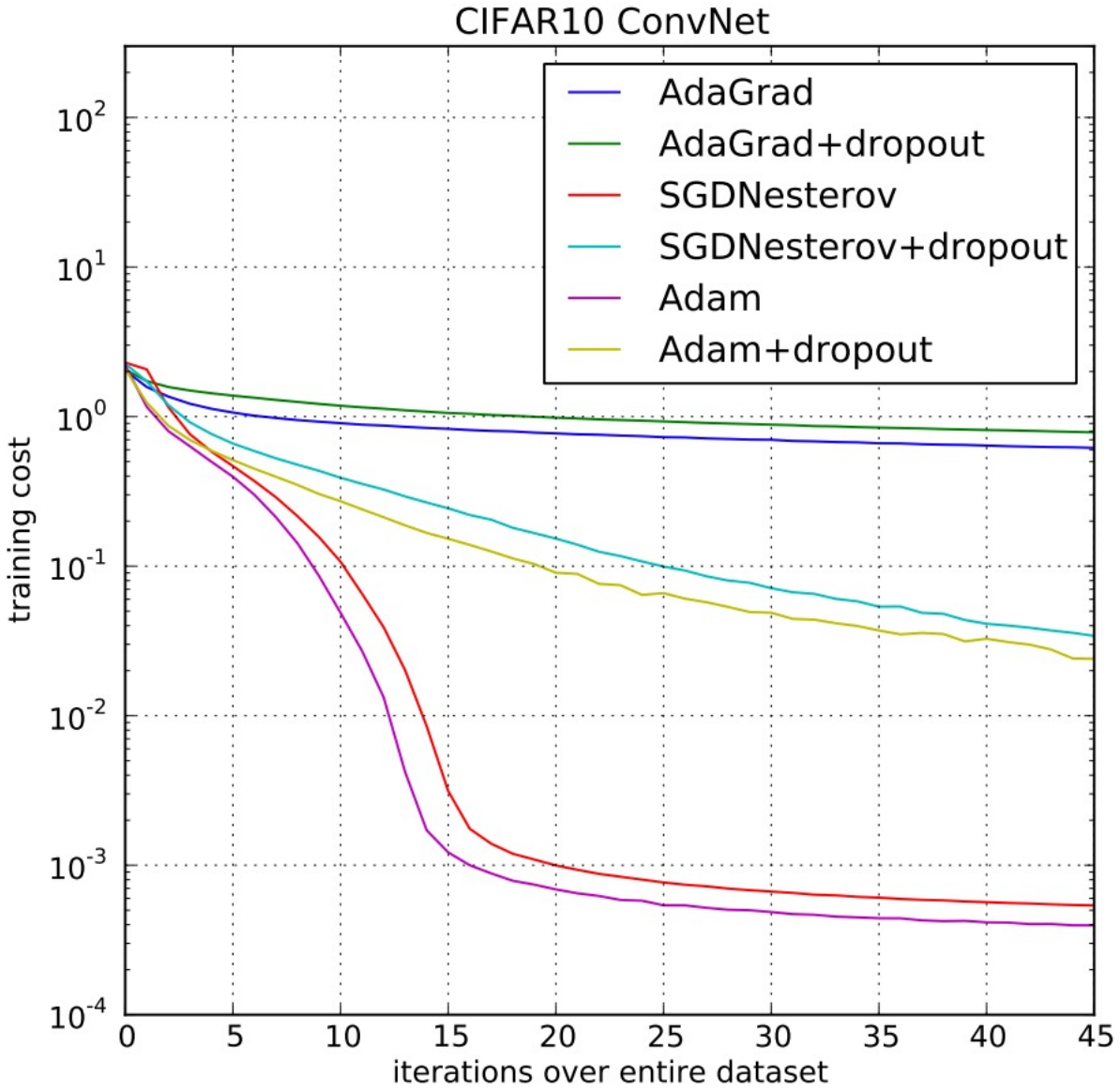
Why care about bias correction?

- It sorts itself out after several iterations right?
- Maybe, but you will make large updates at the start which will destroy any smart weight initialization that took place
- Or worse, send you into a spot in the parameter space with no gradients




```
v1_biased = 0 # Momentum
v2_biased = 0 # RMSProp
for i in range(num_iterations):
  dJ_dw = compute_gradients(train_data, cost_func, w)
  v1_biased = beta1*v1_biased + (1-beta1)*dJ_dw
  v2_biased = beta2*v2_biased + (1-beta2)*dJ_dw*dJ_dw
  v1 = v1_biased / (1 – beta1**(i+1))
  v2 = v2_biased / (1 – beta2**(i+1))
  w = w - learning_rate*v1/sqrt(v2)
```





"Adam: A Method for Stochastic Optimization", Kingma, Ba, 2014, https://arxiv.org/abs/1412.6980v8

Optimizers in Keras

- https://keras.io/optimizers/

- SGD
- RMSProp
- Adagrad
- Adadelta
- Adam
- AdaMax
- Nadam

Second-Order Optimization

- Look also at second-order derivative (Hessian)
- Tells you about curvature

Learning Objectives

- Gain a deeper understanding of the optimization problem of training a neural network
- Learn about the various other popular variants of Gradient Descent
  - Mini-batch Gradient Descent
  - Stochastic Gradient Descent
  - Gradient Descent with Momentum and Nesterov Momentum
  - Adagrad (Adaptive Learning Rates)
  - RMSProp
  - Adam