CSE 465
Lecture 10 & 11

Regularization & Optimization

What is regularization?

- Regularization is any modification made to the learning algorithm with an intention to lower the generalization error but not the training error
- Many standard regularization concepts from machine learning can be readily extended to deep models
  - In context of deep learning, most regularization strategies are based on regularizing estimators
- This is done through reducing variance at the expense of increasing the bias of the estimator
  - That is we go towards simple models
- An effective regularizer is one that decreases the variance significantly while not overly increasing the bias

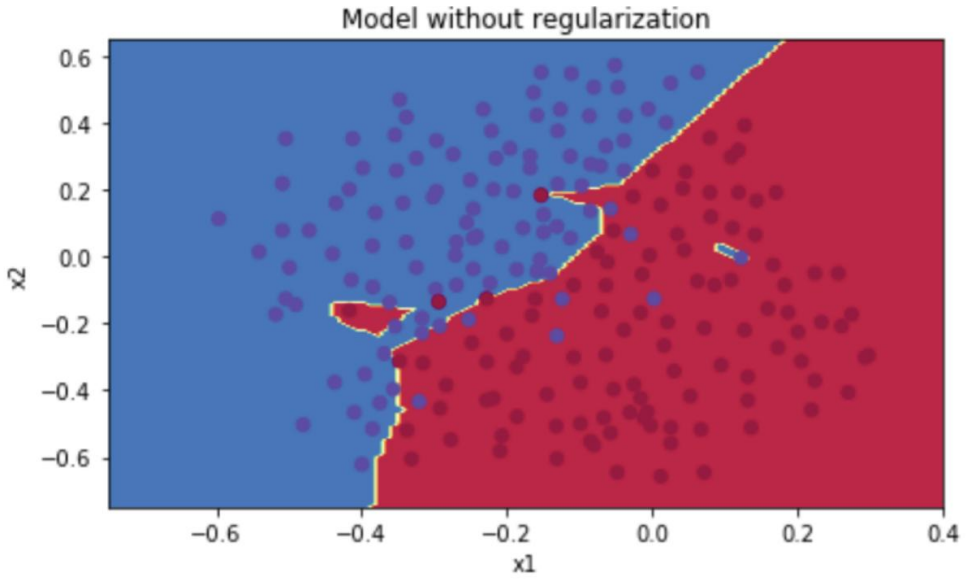
Regularization Methods

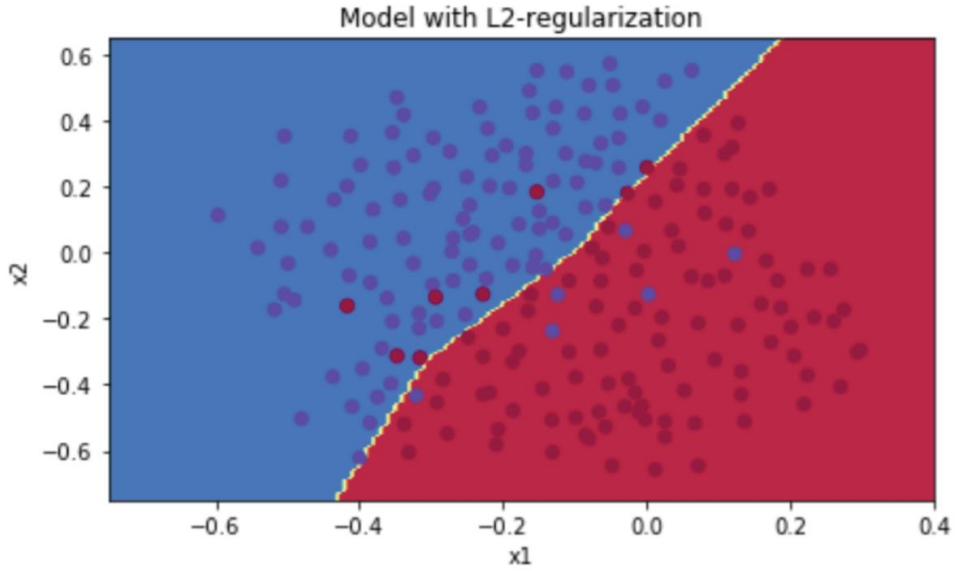
- Norm Penalties
- Data set augmentation
- Noise robustness
- Limiting capacity
  - Early stopping
  - Network reduction
    - Change no of hidden units
    - Change connections

Why do we want to regularize?

- Encode prior knowledge into the models
- Express preference for simpler models
  - Maybe for faster inferences
  - Faster training (sometimes)
  - Intuition, etc.
- Regularization sometime makes problem concrete and determined
- Prevent over-fitting
- Bayesian point of view – incorporates priors to the model parameters
- Find the right number of parameters

Effect of regularization





Regularization: parameter norm penalties

- The most traditional form of regularization applicable to deep learning is the concept of **parameter norm penalties**.
- This approach limits the capacity of the model by adding the penalty $\Omega(\theta)$ to the objective function resulting in:

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(\theta)$$

- $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty to the value of the objective function.

Parameter norm penalties

- When the optimization procedure tries to minimize the objective function, it will also decrease some measure of **size** of the parameters θ.
- *Note*: The bias terms in the affine transformations of deep models usually require less data to be fit and are usually left unregularized.
- Without loss of generality, we will assume we will be regularizing only the weights **w**.

L2 norm parameter regularization

- The L2 parameter norm penalty, also known as **weight decay** drives **w** closer to the origin by adding the regularization term:

$$\Omega(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||_2^2 = \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

- For now, assume there is no bias parameters, only weights.

- The update rule of gradient decent using L2 norm penalty is:

$$\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon\nabla_w J(\mathbf{w})$$

- The weights **multiplicatively shrink** by a constant factor at each step.

L1 Norm parameter regularization

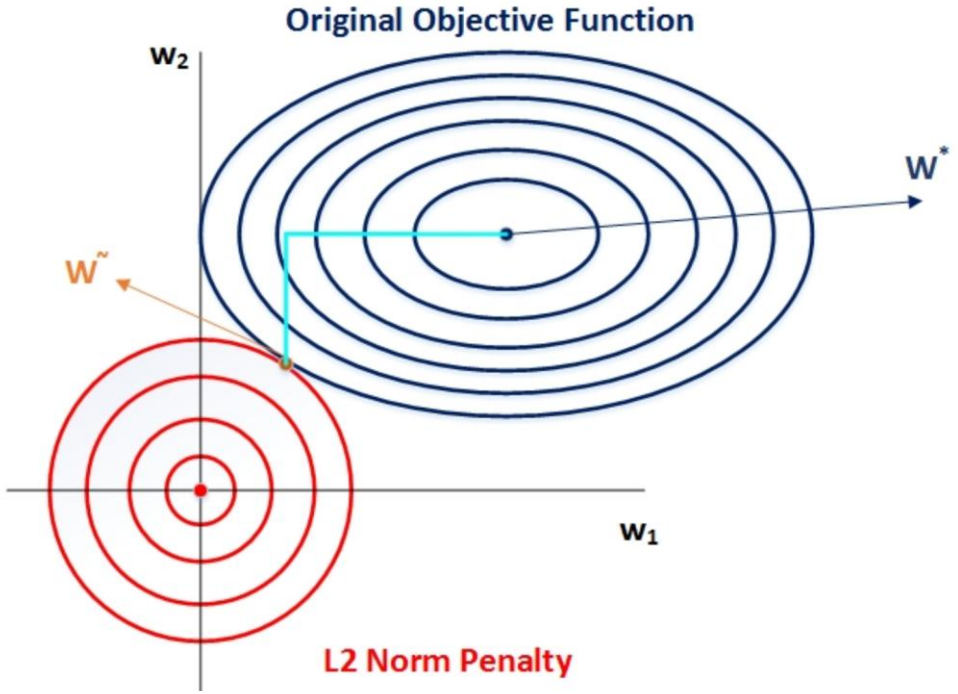
- L1 norm is another option that can be used to penalize the size of model parameters.
- L1 regularization on the model parameters **w** is:

$$\Omega(\theta) = ||\mathbf{w}|| = \sum_i |w_i|$$

- What is the difference between L2 and L1 norm penalty when applied to machine learning models ? But what happens over the entire course of training in both?

L1 Norm regularization

- The L2 Norm penalty decays the components of the vector **w** that do not contribute much to reducing the objective function.



- On the other hand, the L1 norm penalty provides solutions that are **sparse**.
- This **sparsity** property can be thought of as a **feature selection** mechanism.

Regularization: Data preprocessing

- Before we feed data to the neural network, we need to do some data cleanup and processing to get it ready for the learning model
- When given a large amount of training data, neural networks are able to extract and learn features from raw data, unlike the other traditional ML techniques
- Preprocessing still might be required to improve performance or work within specific limitations on the neural network, such as
  - Converting images to grayscale
  - Image resizing, normalization, and
  - Data augmentation

Regularization: Dataset augmentation

- We have seen that for consistent estimators, the best way to get better generalization is to train on more data.
- The problem is that under most circumstances, data is limited. Furthermore, labelling is an extremely tedious task.
- **Dataset Augmentation** provides a cheap and easy way to increase the amount of your training data.
- Certain tasks such as steering angle regression require dataset augmentation to perform well.

Dataset augmentation: Color jitter

- **Color jitter** is a very effective method to augment datasets. It is also extremely easy to apply.
- **Fancy PCA** was proposed by Krizhevsky et al. in the famous Alex net paper. It is a way to perform color jitter on images.
- Fancy PCA Algorithm:
  - Perform PCA on the three color channels of your entire dataset.
  - From the covariance matrix provided by PCA, extract the eigenvalues $\lambda_1, \lambda_2, \lambda_3$ and their corresponding eigenvectors p_1, p_2, p_3.
  - Add $p_i[a_1\lambda_1, a_2\lambda_2, a_3\lambda_3]^T$ to the ith color channel. $a_1 ... a_3$ are random variables sampled for each augmented image from a zero mean Gaussian distribution with a variance of 0.1.

Dataset augmentation: color jitter

Dataset augmentation: Horizontal flipping

- **Horizontal Flipping** is applied on data that exhibit horizontal asymmetry.
- Care must be taken to propagate the labels through this transformation.
- Horizontal flipping can be applied to natural images and point clouds. Essentially, one can double the amount of data through horizontal flipping.

Dataset augmentation: horizontal flipping

Dataset augmentation: Conclusion

- Many other task specific dataset augmentation algorithms exist. It is highly advised to always use dataset augmentation.
- However, be careful not to alter the correct output!
- Furthermore, when comparing two machine learning algorithms train both with either augmented or non-augmented dataset. Otherwise, no subjective decision can be made on which algorithm performed better.

Regularization: Noise robustness

- **Noise Injection** can be thought of as a form of regularization. The addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights (Bishop, 1995).
- Noise can be injected at different levels of deep models.

Noise robustness: Noise injection on weights

- Noise added to weights can be interpreted as a more traditional form of regularization.
- This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output.
- In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions (Hochreiter and Schmidhuber, 1995).

Noise robustness: noise injection on outputs

- Most datasets have some amount (A LOT!) of mistakes in the y labels. Minimizing our cost function on wrong labels can be extremely harmful.
- One way to remedy this is to explicitly model the noise on labels. This is done through setting a probability ϵ for which we think the labels are correct.
- This probability is easily incorporated into the cross entropy cost function **analytically**.
- An example is **label smoothing**.

Noise robustness: label smoothing

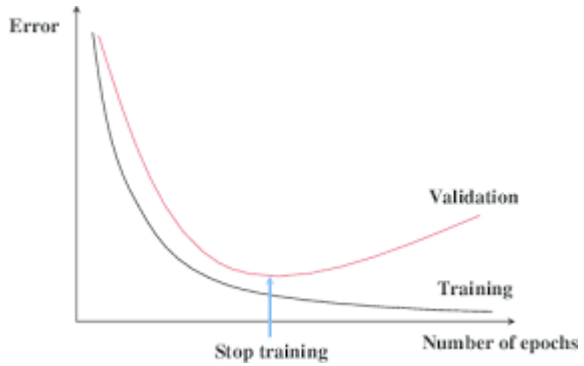
- Usually, we have output vectors provided to us as $y_{label} = [1, 0, 0, 0...0]$.
- Softmax output is usually of the form $y_{out} = [0.87, 0.001, 0.04, 0.1, ...0.03]$.
- Maximum likelihood learning with a softmax classifier and hard targets may actually never converge, the softmax can never predict a probability of exactly 0 or exactly 1, so it will continue to learn larger and larger weights, making more extreme predictions. forever

Early stopping

- Early Stopping is probably one of the most used regularization strategies in deep learning.
- Early stopping can be thought of as a hyperparameter selection method, where **training time** is the hyperparameter to be chosen.
- However, a portion of data should be reserved for validation.

Early stopping

- When training models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, while the error on the validation set begins to rise again.
- The occurrence of this behaviour in the scope of our applications is almost certain.
- This means we can obtain a model with better validation set error (and thus,hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error.
- This is termed **Early Stopping**.



Early Stopping

- Early Stopping is probably one of the most used regularization strategies in deep learning.
- Early stopping can be thought of as a hyperparameter selection method, where **training time** is the hyperparameter to be chosen.
- Choosing the training time automatically can be done through a single run through the training phase, the only addition being the evaluation of the validation set error at every n iterations. This is usually done on a second GPU.
- Overhead for writing parameters to disk is negligible.

Early Stopping: Exploiting the validation set

- To exploit all of our precious training data we can:
  - Employ early stopping as described above.
  - Retrain using all of the data up to the point that was determined during early stopping.
- Some subtleties arise regarding the definition of **point**.
- Do we train for the same number of parameter updates or for the same number of epochs(passes through training data) ?

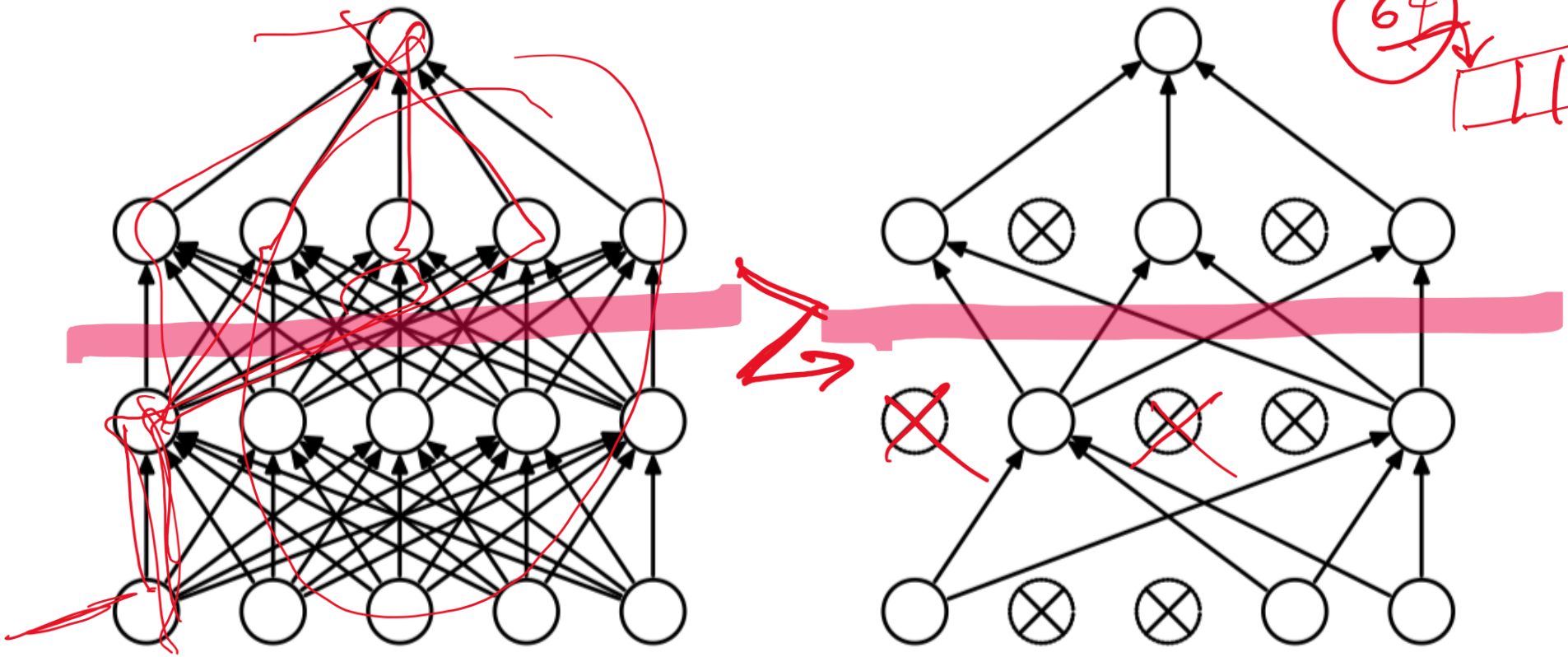
Exploiting the validation set

- A second strategy to exploit the full training dataset would be to:
  - Employ early stopping as described above.
  - Continue training with the parameters determined by early stopping, using the validation set data.
- This strategy avoids the high cost of retraining the model from scratch, but is not well-behaved.
- Since we no longer have a validation set, we cannot know if generalization error is improving or not. Our best bet is to stop training when the training error is not decreasing much any more.

Dropout

- Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.
- Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network.

Dropout regularization



(a) Standard Neural Net

(b) After applying dropout.

Dropout regularization

- Keep random neurons based on probability *"keep"*
- For each mini-batch
  - Randomly shut-down a neuron with probability *(1-keep)*
    - Keep these off while backpropagation
    - Increase the weight by *1/keep*
- Why dropout regularization works?
  - Do not rely on a single weight too much
  - Distributes the weight
  - Similar to L2 regularization

Training with Dropout

- To train with dropout,we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent.
- Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network.
- The mask for each unit is sampled independently from all of the others.
- Typically, the probability of including a hidden unit is 0.5, while the probability of including an input unit is 0.8.

Training with Dropout

- Dropout allows us to represent an exponential number of models with a tractable amount of memory.
- Furthermore, Dropout removes the need to accumulate model votes at the inference stage.
- Dropout can intuitively be explained as forcing the model to learn with missing input and hidden units.

Dropout

- Dropout training has some intricacies we need to be wary of.
- At training time, we are **required** to divide the output of each unit by the probability of that unit's dropout mask.
- The goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.
- No theoretically satisfying basis for the accuracy of this approximate training rule in deep non linear networks, but empirically it performs very well.

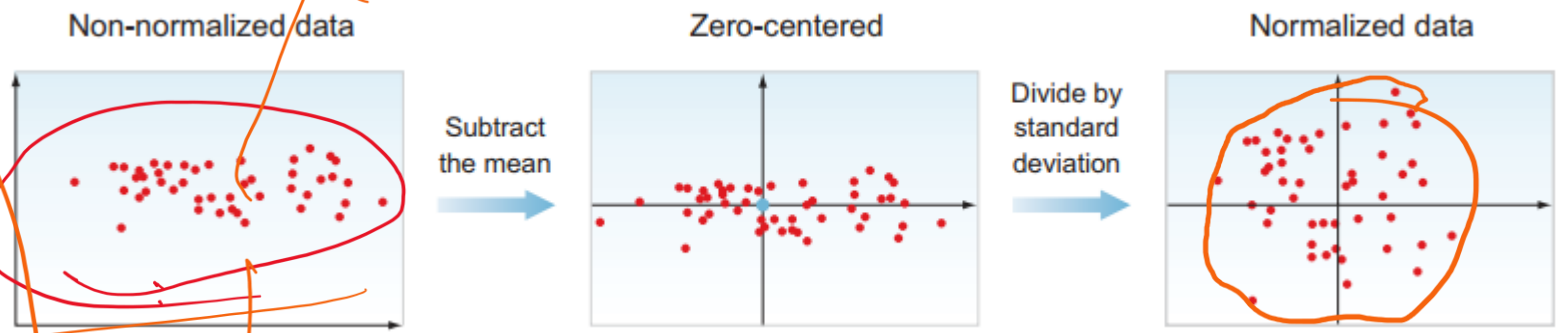
Feature normalization

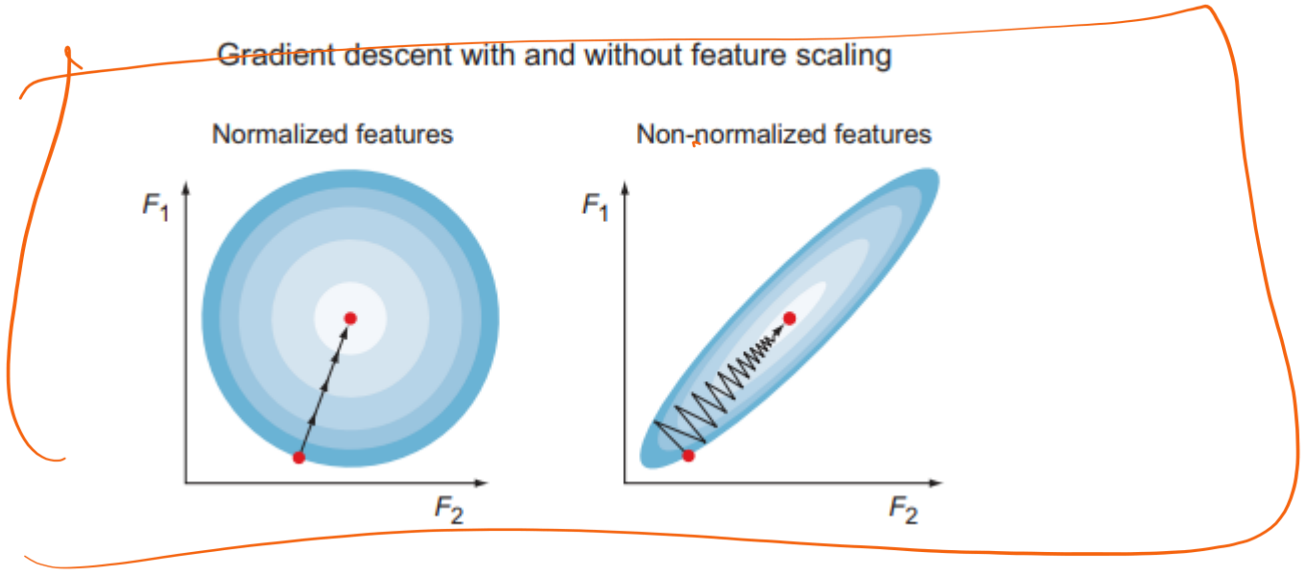
- Feature normalization: it normalizes features before applying the learning algorithm
  - Basically, rescaling the features
    - Generally done during preprocessing
- According to the paper "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", gradient descent converges much faster with feature scaling than without it

Normalizing the input

- Data normalization is the process of rescaling data to ensure that each input feature (pixel, in the image case) has a similar data distribution
- Often, raw images are composed of pixels with varying scales (ranges of values)
  - For example, one image may have a pixel value range of 0 to 255, while another may have a range of 20 to 200
  - It is preferred to normalize the pixel values to the range of 0 to 1 to boost learning performance and make the training converge faster

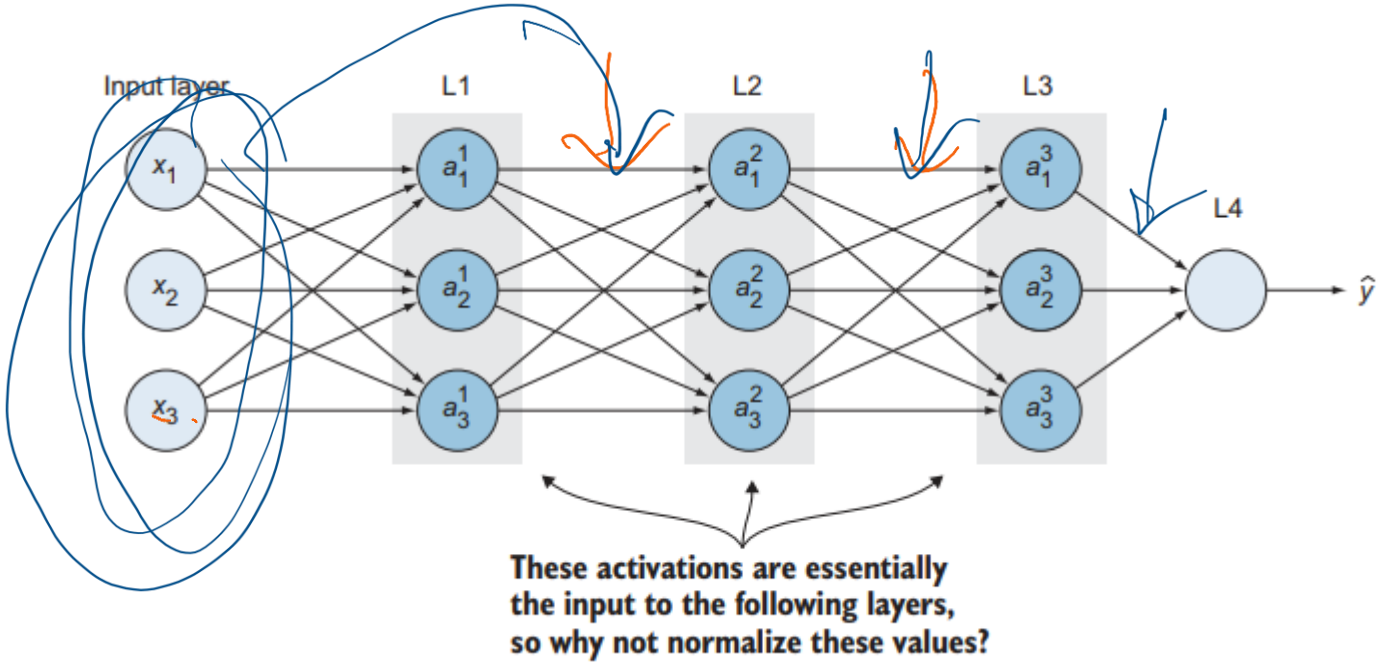
Normalizing the input





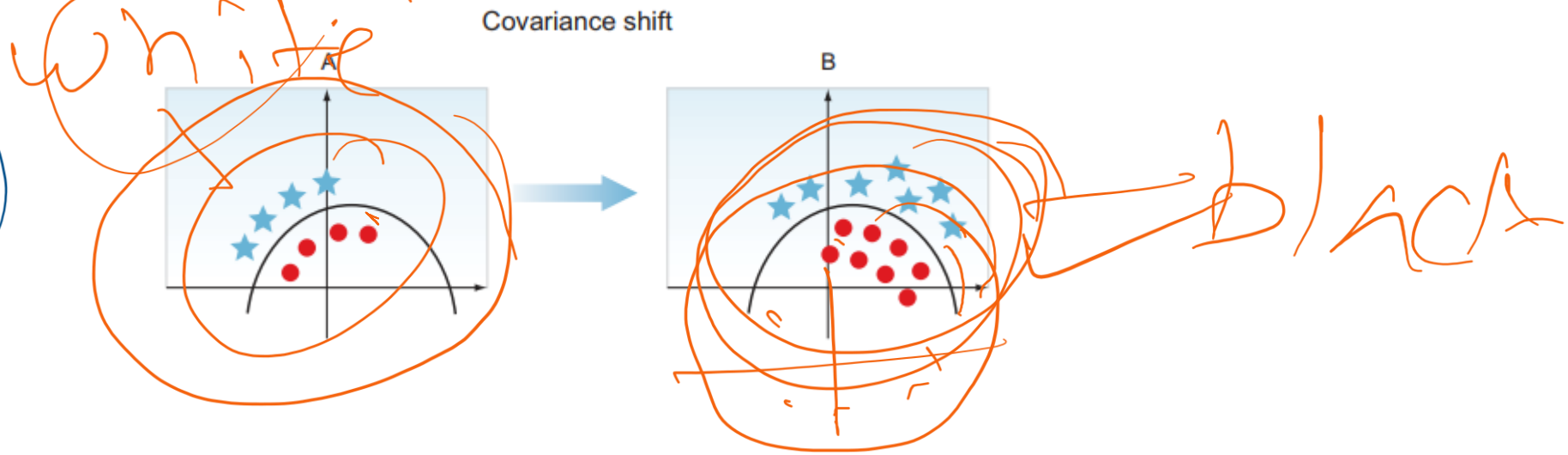
Batch Normalization

- We talked about input data normalization to speed up learning
- The normalization techniques we discussed are focused on preprocessing the input set before feeding it to the network
- If the input layer benefits from normalization, why not do the same thing for the *extracted features* in the hidden units, which are changing all the time and get much more improvement in training speed and network resilience?
- This process is called *batch normalization* (BN)



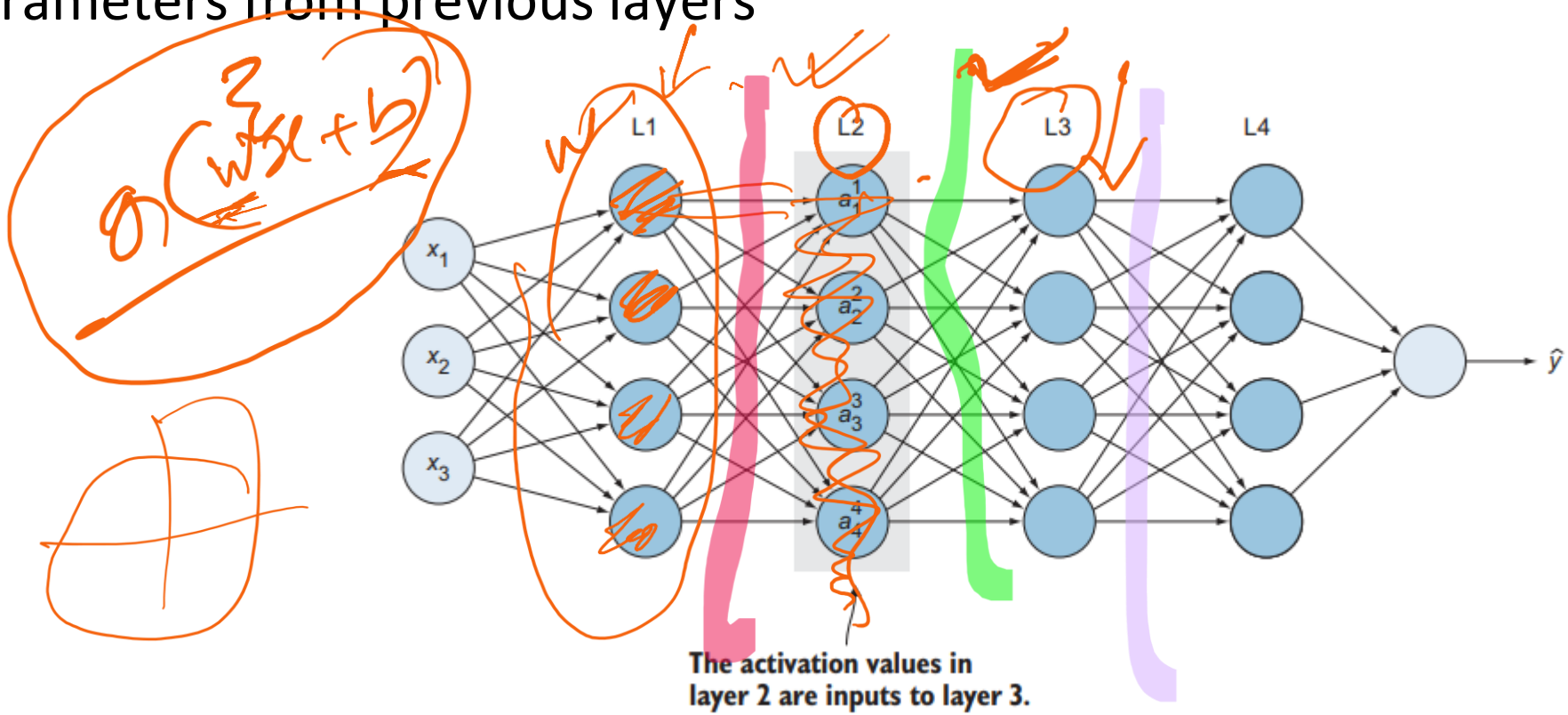
Covariance Shift

- Suppose we are building a cat classifier, and we train the network on images of white cats only
- When we test this classifier on images with cats that are different in colors, it will not perform well - Why?
- Because the model has been trained on a training set with a specific distribution (white cats)
- When the distribution changes in the test set, it confuses the model



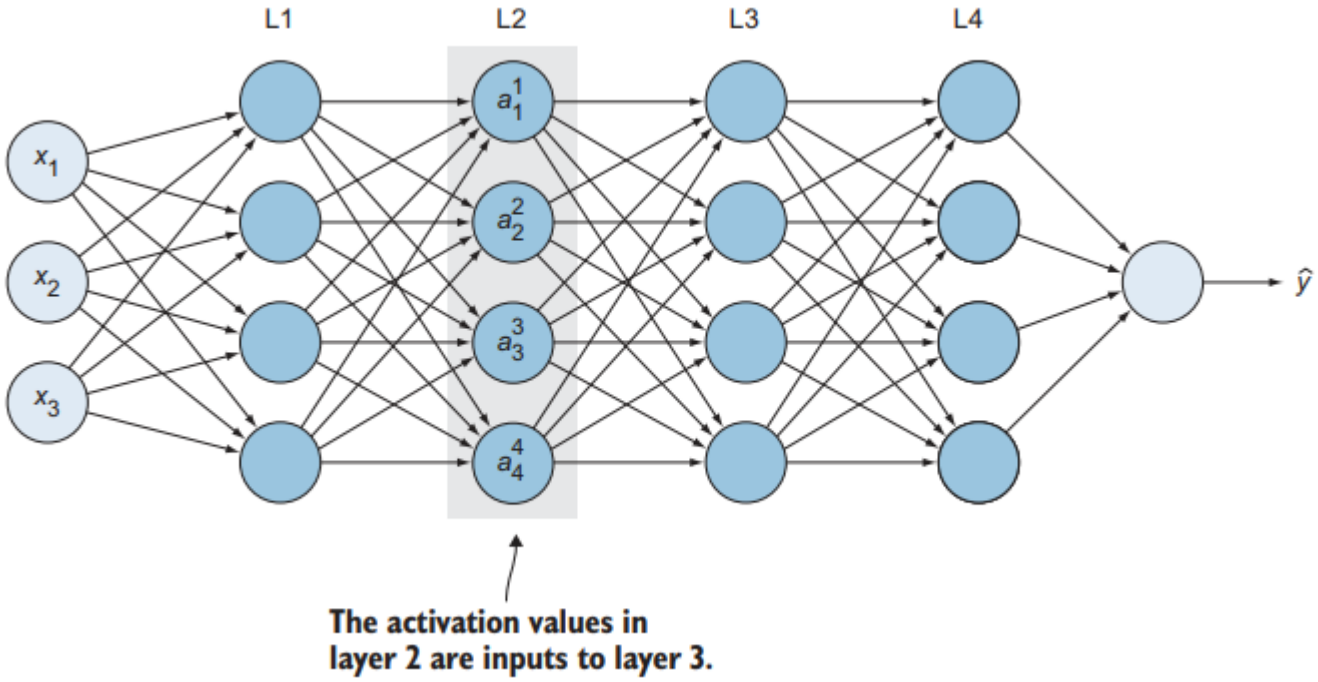
Covariance Shift in Neural Networks

- Let's look at the network from the third-layer (L3) perspective
- Its input are the activation values in L2 (a_1^2, a_2^2, a_3^2, and a_4^2), which are the features extracted from the previous layers and L3 is trying to map these inputs to $\hat{y}$
- While the third layer is doing that, the network is adapting the values of the parameters from previous layers



Covariance Shift in Neural Networks

- As the parameters (w, b) are changing in layer 1, the activation values in the second layer are changing
- So from the perspective of the third hidden layer, the values of the second hidden layer are changing all the time: the MLP is suffering from the problem of covariate shift
- Batch normalization reduces the degree of change in the distribution of the hidden unit values, causing these values to become more stable so that the later layers of the neural network have firmer ground to stand on

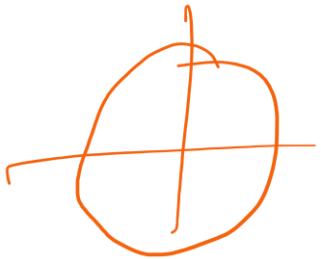


Batch Normalization Details

- Batch normalization adds an operation in the neural network just before the activation function of each layer to do the following:
  - Zero-center the inputs
    - To zero-center the inputs, the algorithm needs to calculate the input mean and standard deviation

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$



  - Normalize the zero-centered inputs

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

  - Scale and shift the results This operation lets the model learn the optimal scale and mean of the inputs for each layer

$$y_i \leftarrow \gamma x_i + \beta$$

Optimization

Gradient Descent Insights

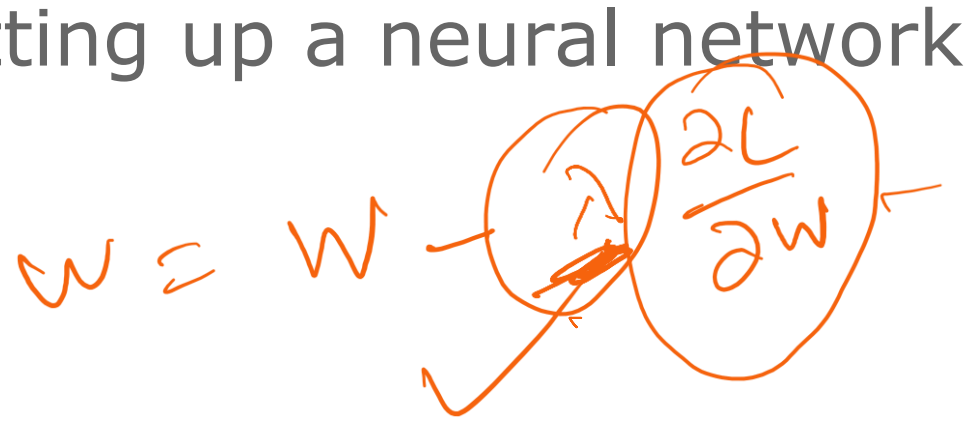
- The learning rate is controlled by a parameter λ

$$w^{(i+1)} = w^{(i)} - \lambda \frac{d\mathcal{L}}{dw}$$

- A large learning rate means more weight is put on the derivative, such that large steps can be made for each iteration of the algorithm
- A smaller learning rate means that less weight is put on the derivative, so smaller steps can be made for each iteration.

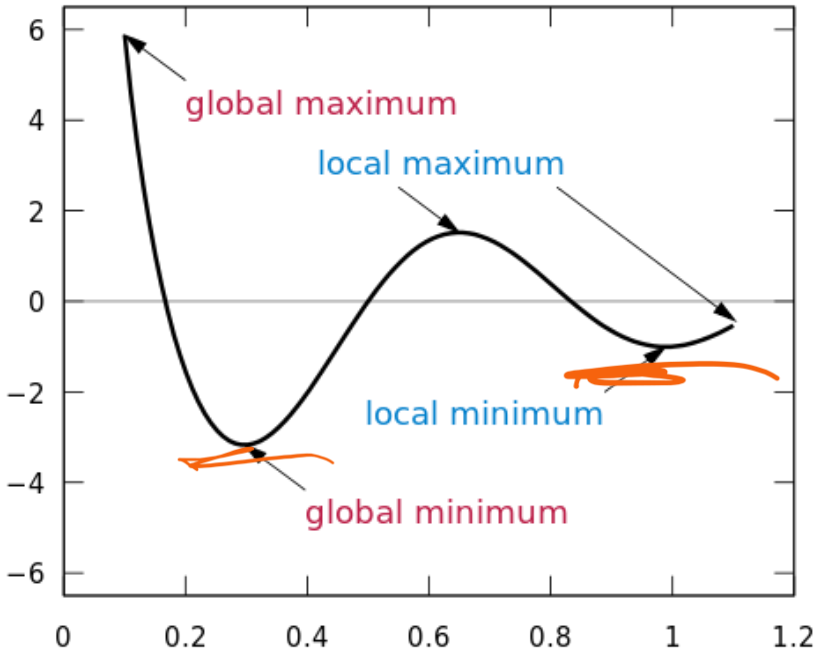
Gradient Descent Insights

- If the step size is too small, the algorithm will take a long time to converge
- If the step size is too large, the algorithm will continually miss the optimal parameter choice
- Selecting the learning rate can be an important parameter when setting up a neural network.



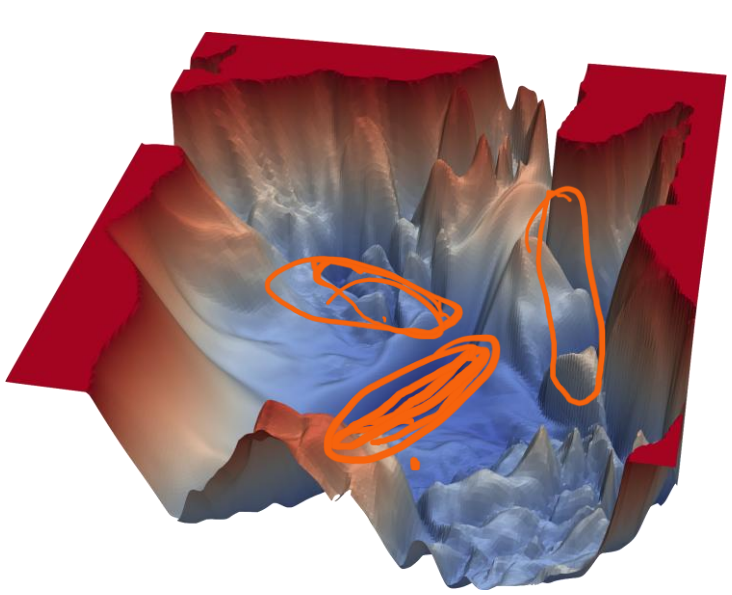
Local minima maxima

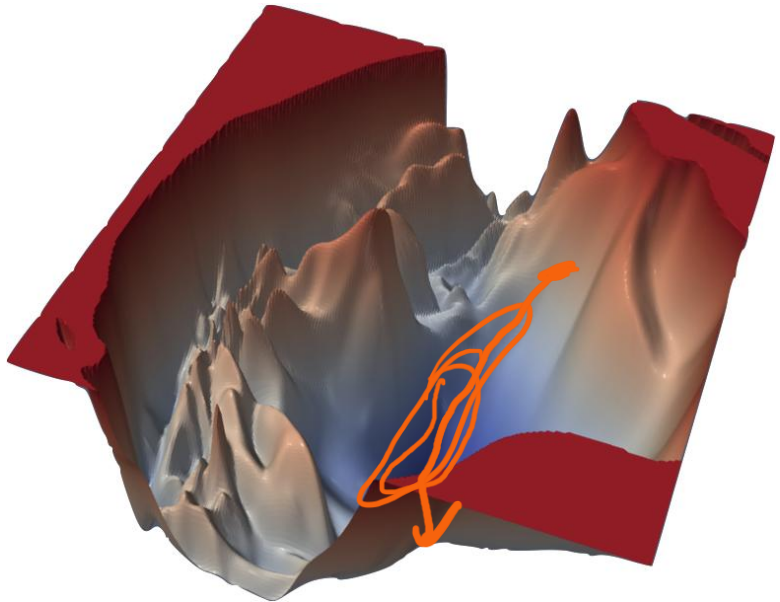
- Local minimum can be very problematic for neural networks since the formulation of neural networks gives no guarantee that we will attain the global minimum



Local minima maxima

- Getting stuck in a local minimum means we have a locally good optimization of our parameters, but there is a better optimization somewhere on our loss surface
- Neural network loss surfaces can have many of these local optima, which is problematic for network optimization



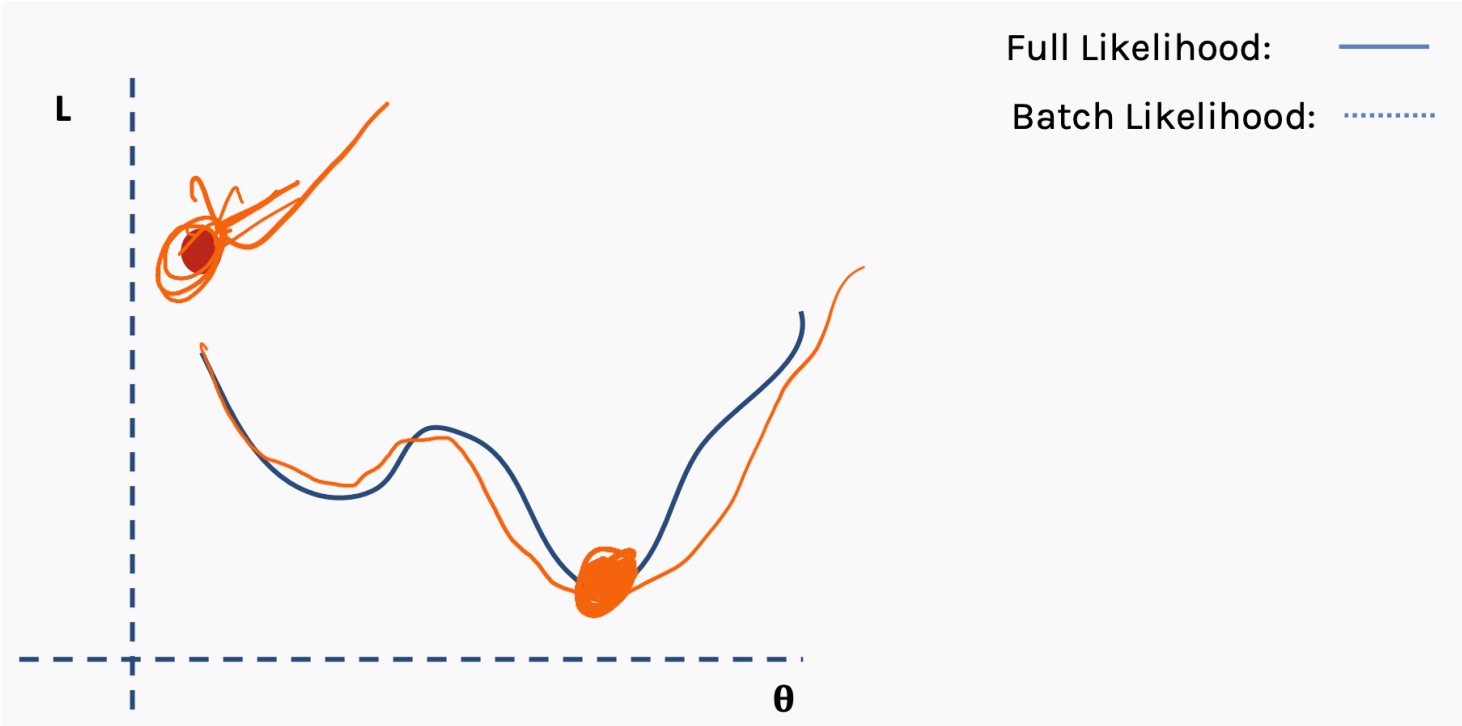


How do we solve this problem?

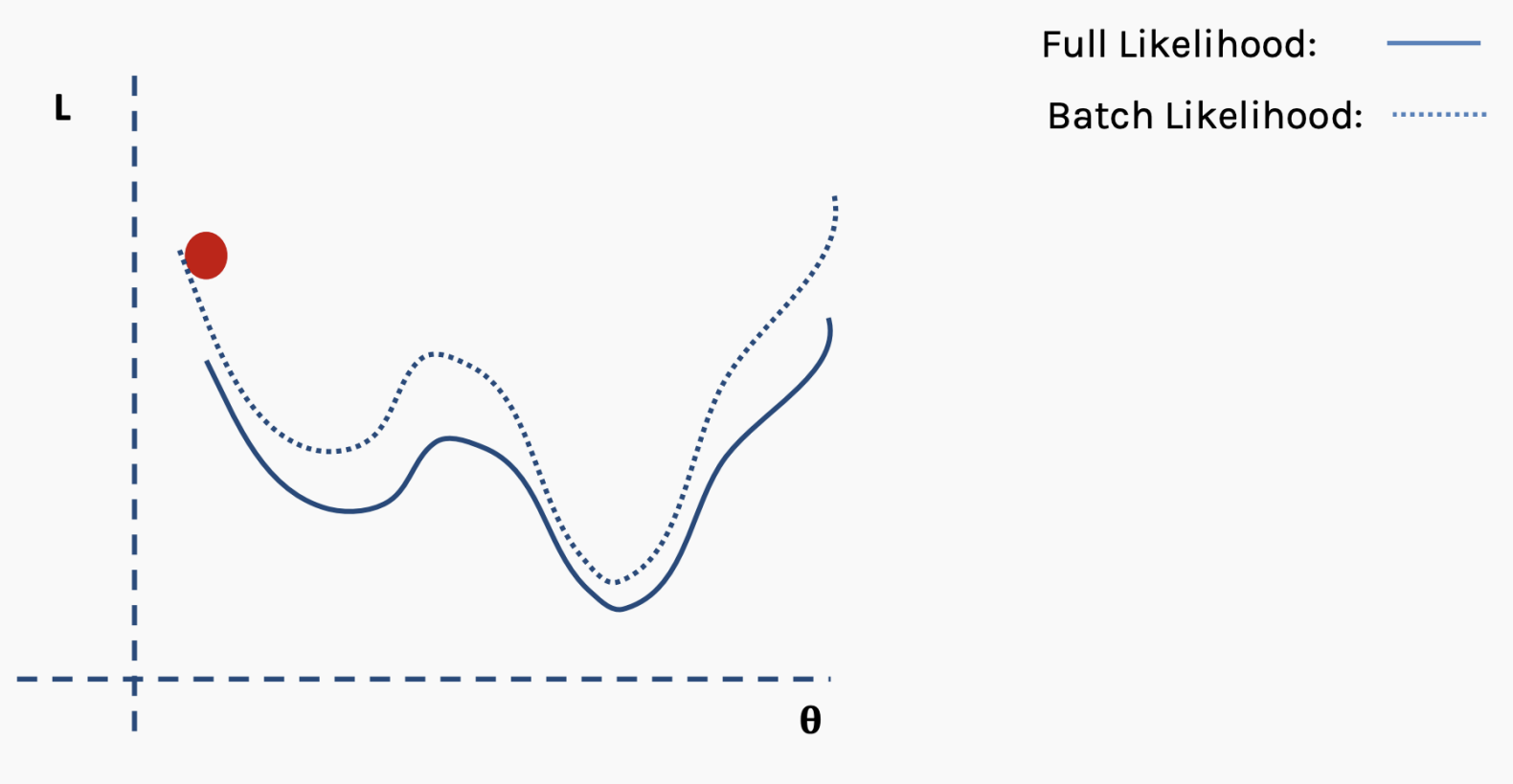
- One suggestion is the use of mini-batch and stochastic gradient descent
- The idea is simple — to use a mini-batch (a subset) of data as opposed to the whole set of data, such that the loss surface is partially morphed during each iteration.
- For each iteration k, the following loss (likelihood) function can be used to derive the derivatives:

$$\mathcal{L}^k = -\sum_{i \in b^k} [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

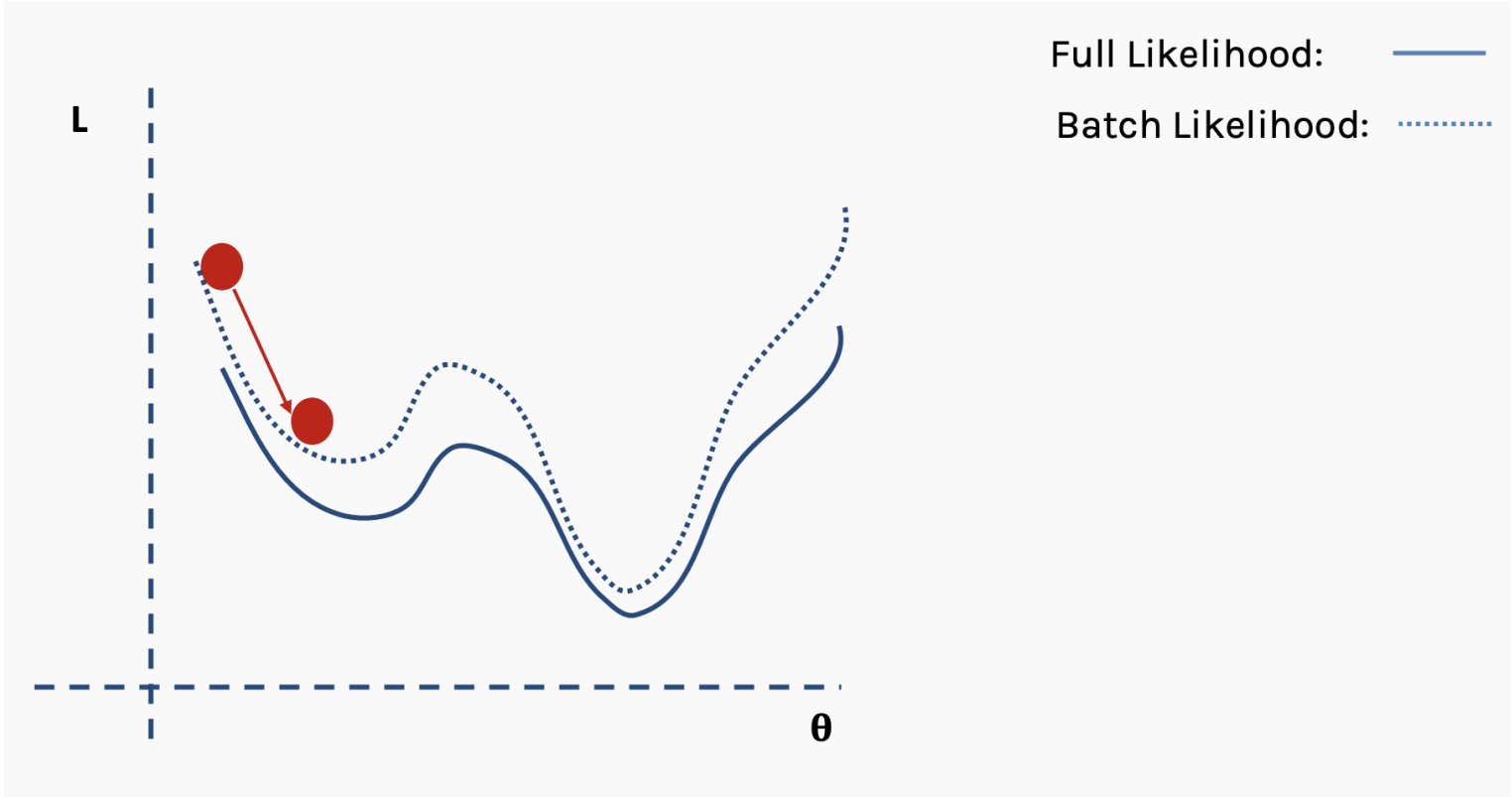
How batch gradient descent helps?

- We start off with the full loss (likelihood) surface, and our randomly assigned network weights provide us an initial value
- 

How batch gradient descent helps?

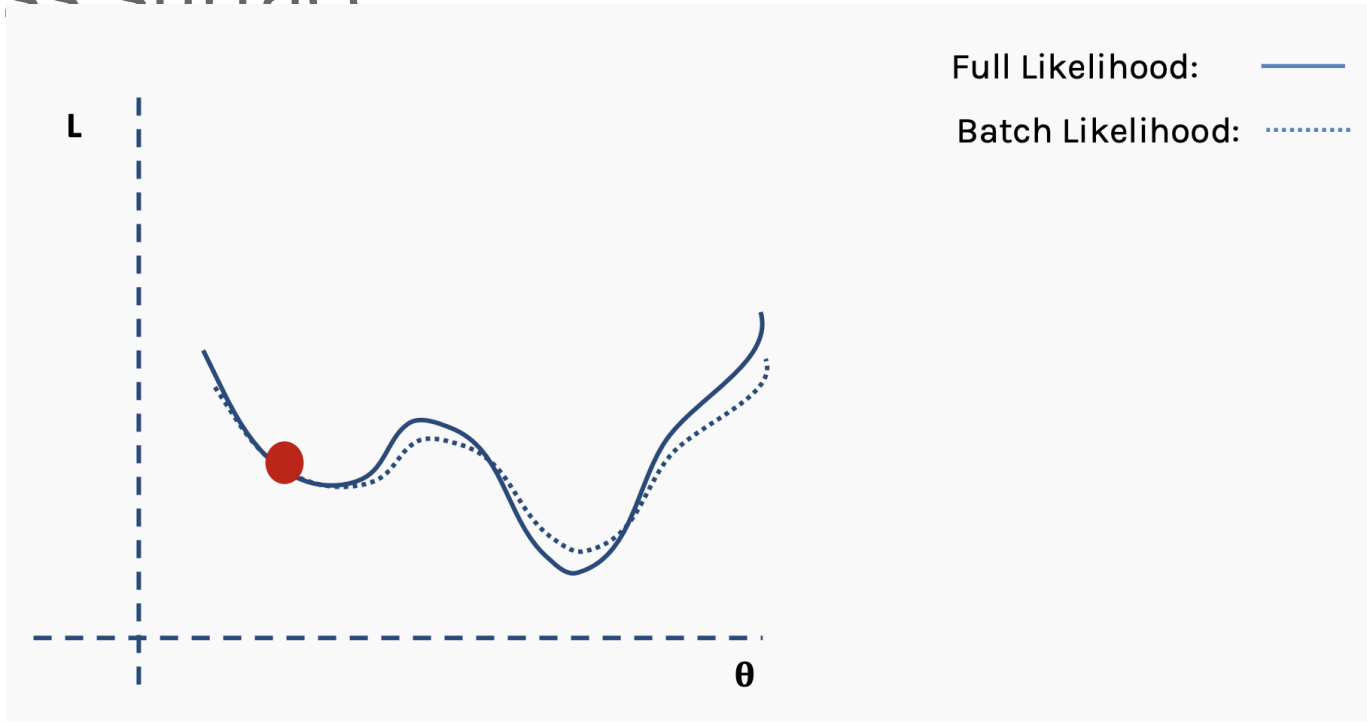
- We then select a batch of data, perhaps 10% of the full dataset, and construct a new loss surface.
- 

How batch gradient descent helps?

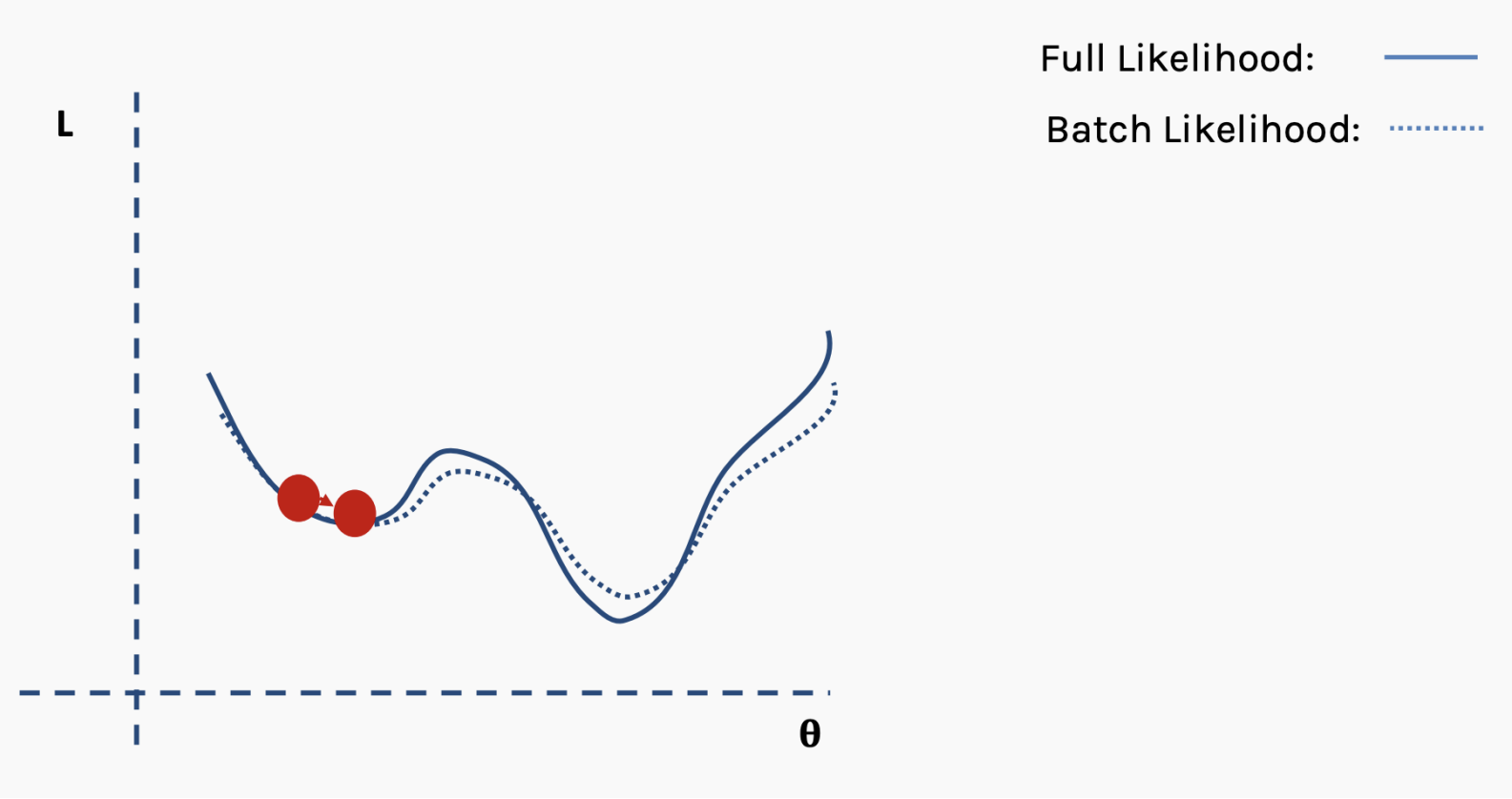
- We then perform gradient descent on this batch and perform our update.
- 

How batch gradient descent helps?

- We are now in a new location
- We select a new random subset of the full data set and again construct our loss surface

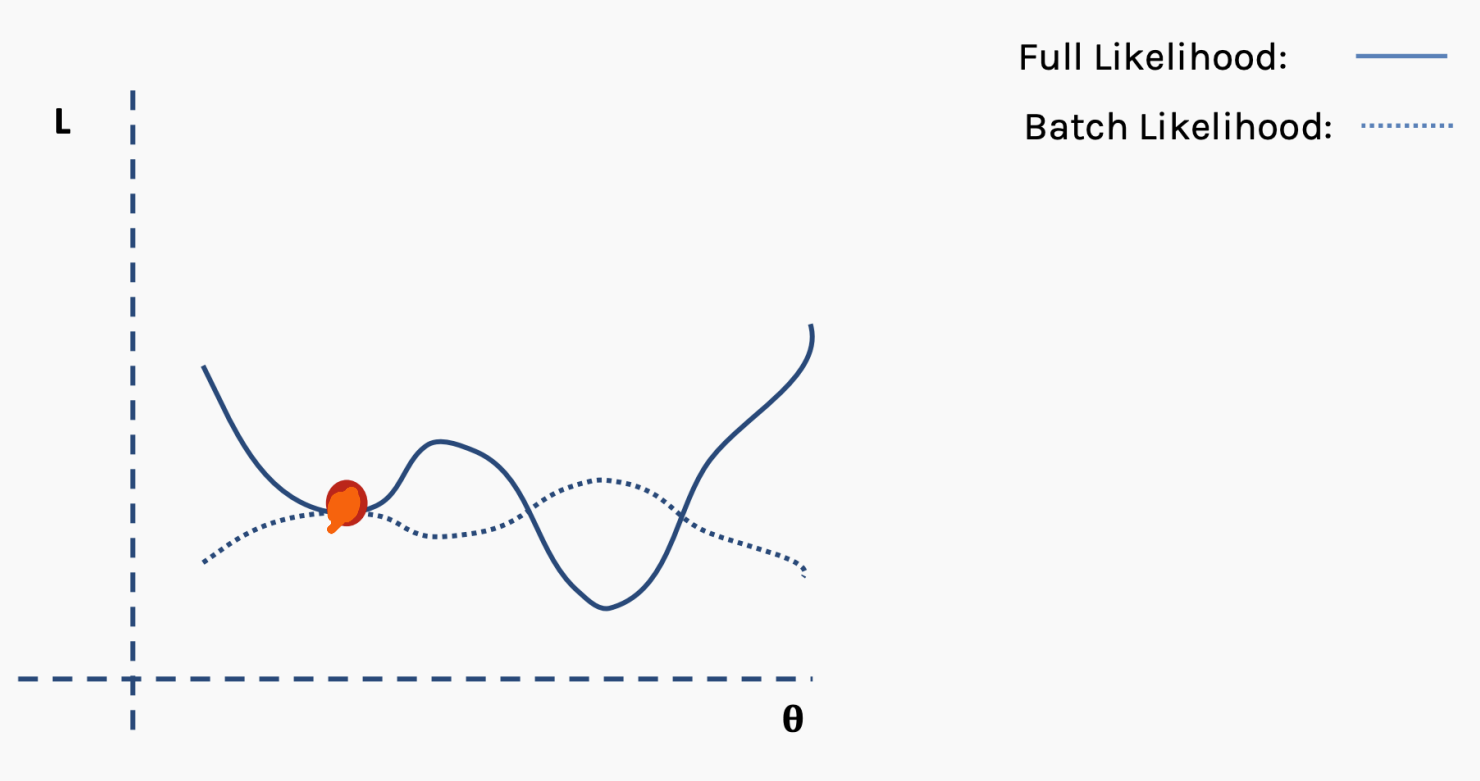


How batch gradient descent helps?

- We then perform gradient descent on this batch and perform our update.
- 

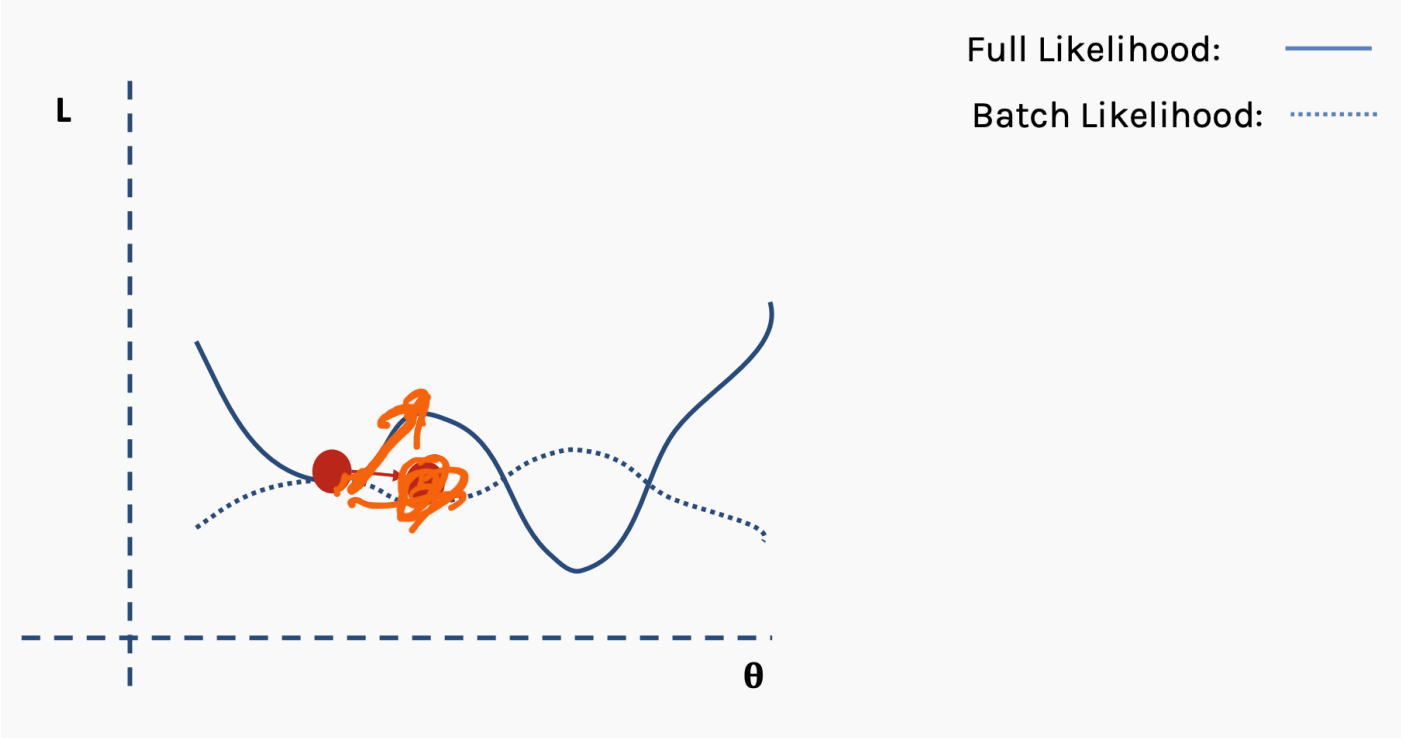
How batch gradient descent helps?

- We continue this procedure again with a new subset



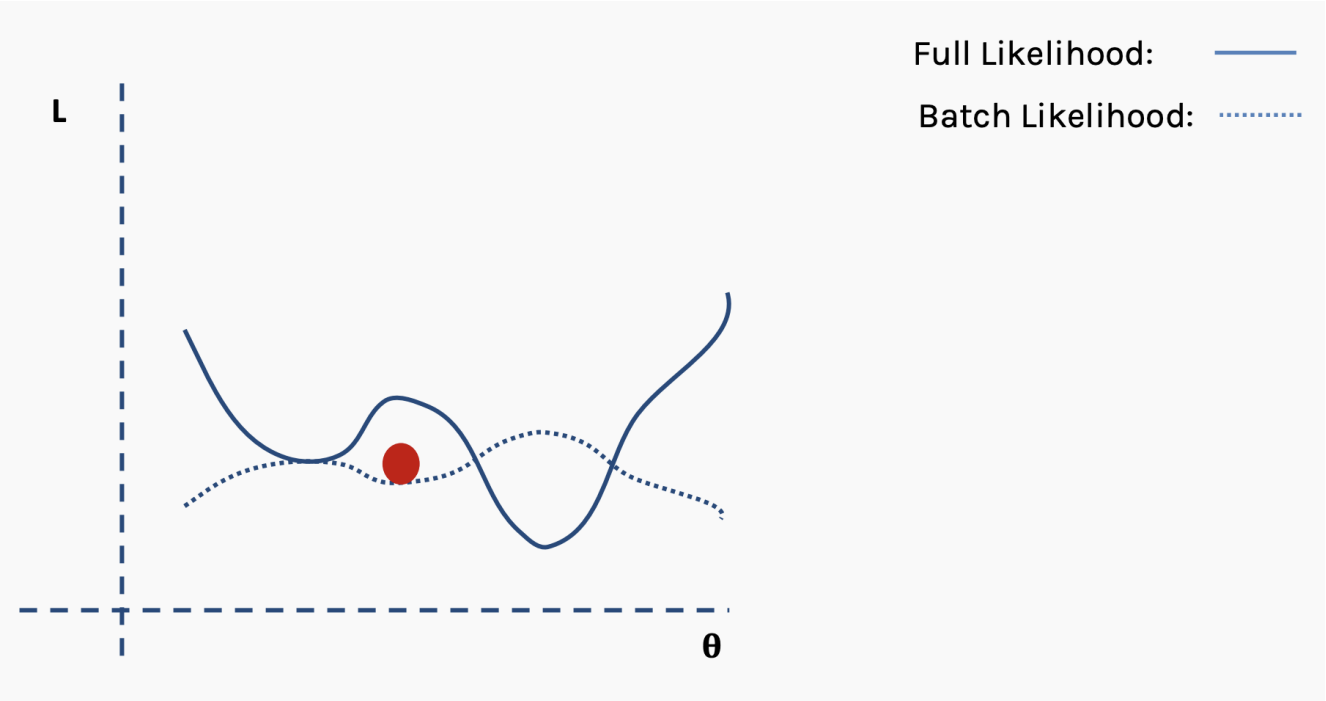
How batch gradient descent helps?

- And perform our another update



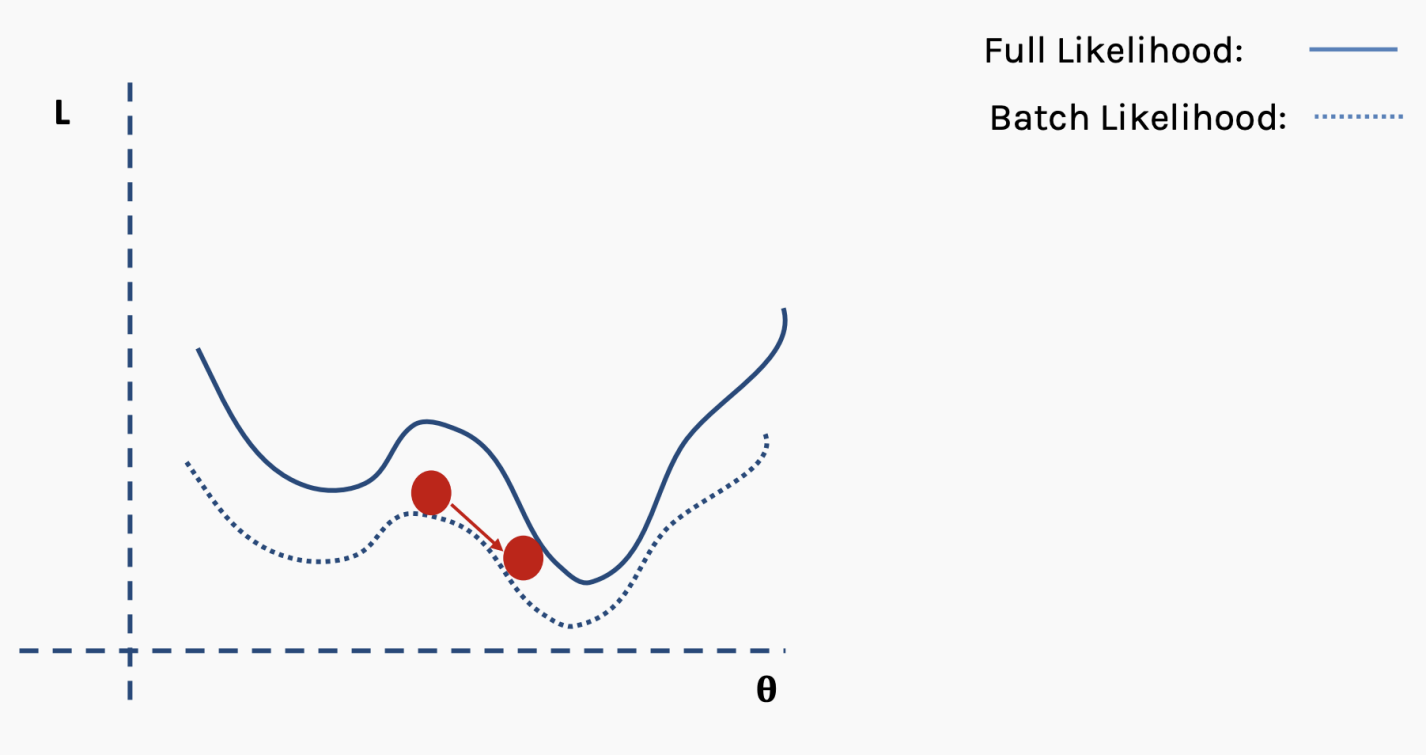
How stochastic gradient descent helps?

- This procedure continues for multiple iterations



How stochastic gradient descent helps?

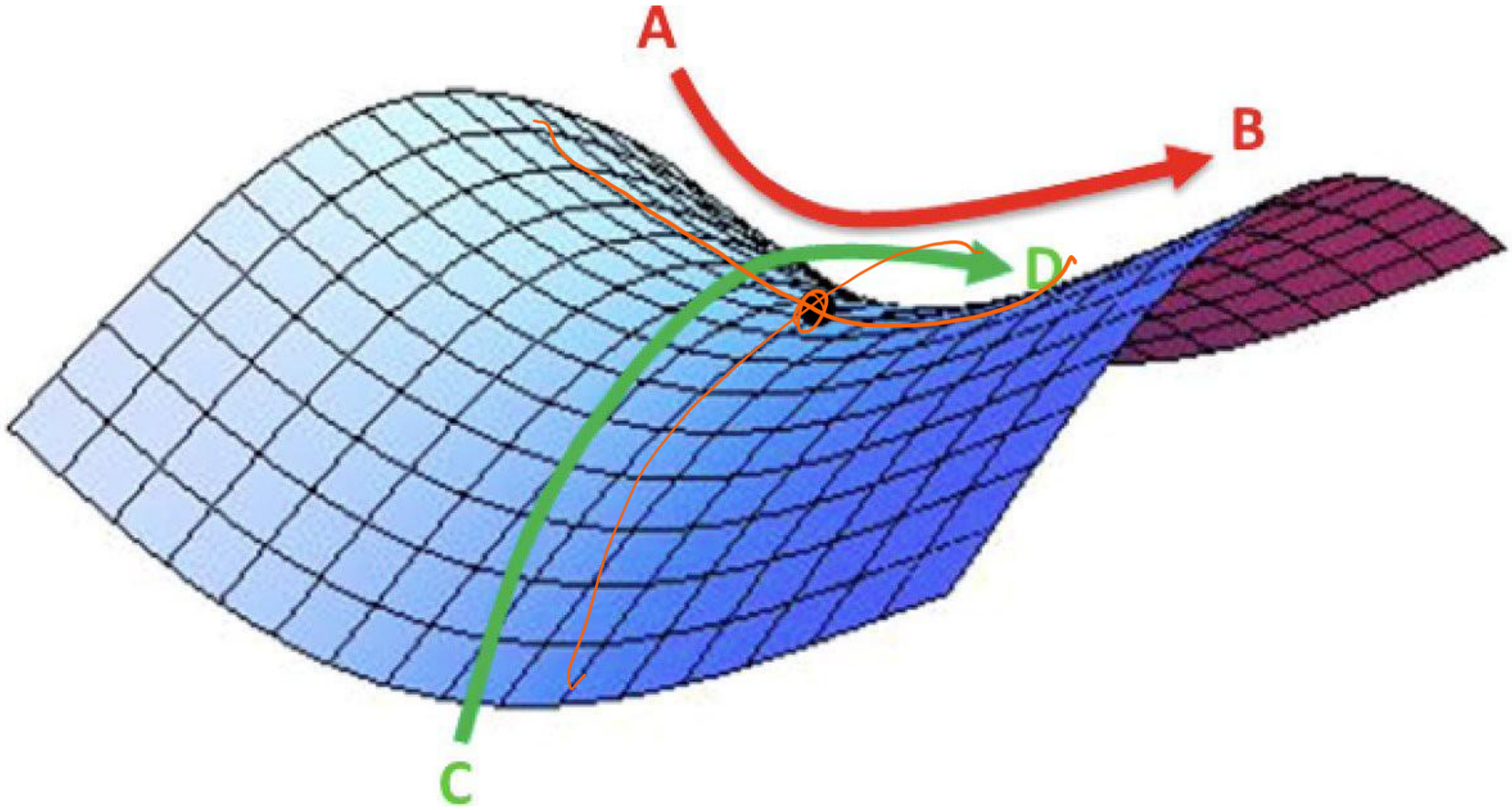
- Until the network begins to converge to the global minimum



More issues: Saddle points

- Recent studies indicate that in high dimensions, saddle points are more likely than local minima
- Saddle points are also more problematic than local minima because close to a saddle point the gradient can be very small
- Gradient descent will result in negligible updates to the network and hence network training will cease

Saddle points

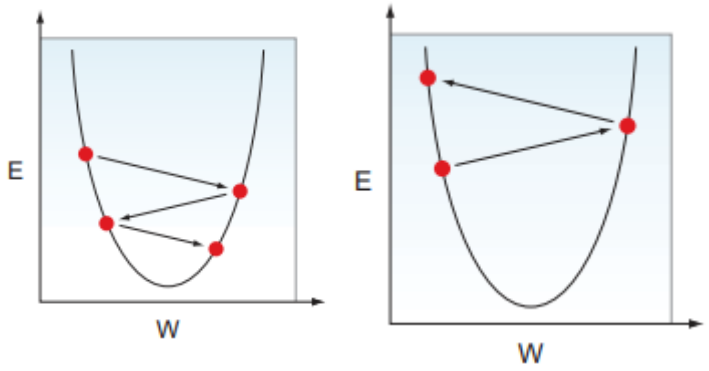


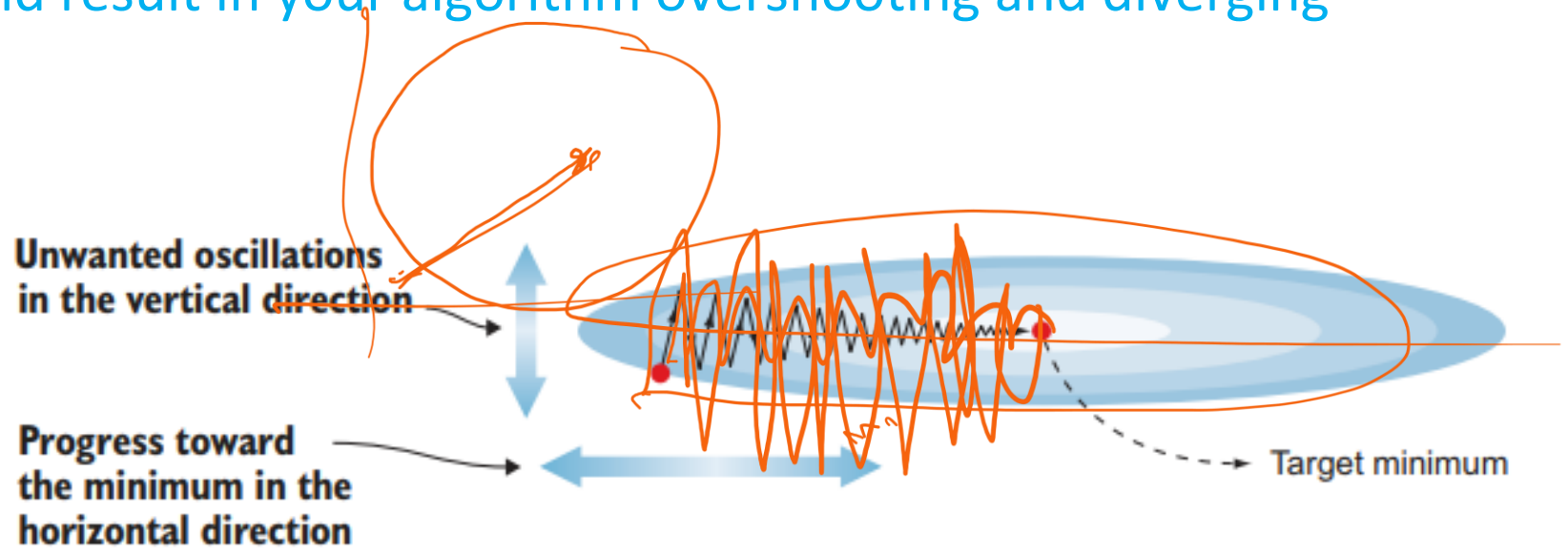
How to find saddle points?

- The Hessian can be used to determine whether a given stationary point is a saddle point or not
- If the Hessian is indefinite at that location, that stationary point is a saddle point
- This can also be reasoned in a similar way by looking at the eigenvalues

SGD Optimization

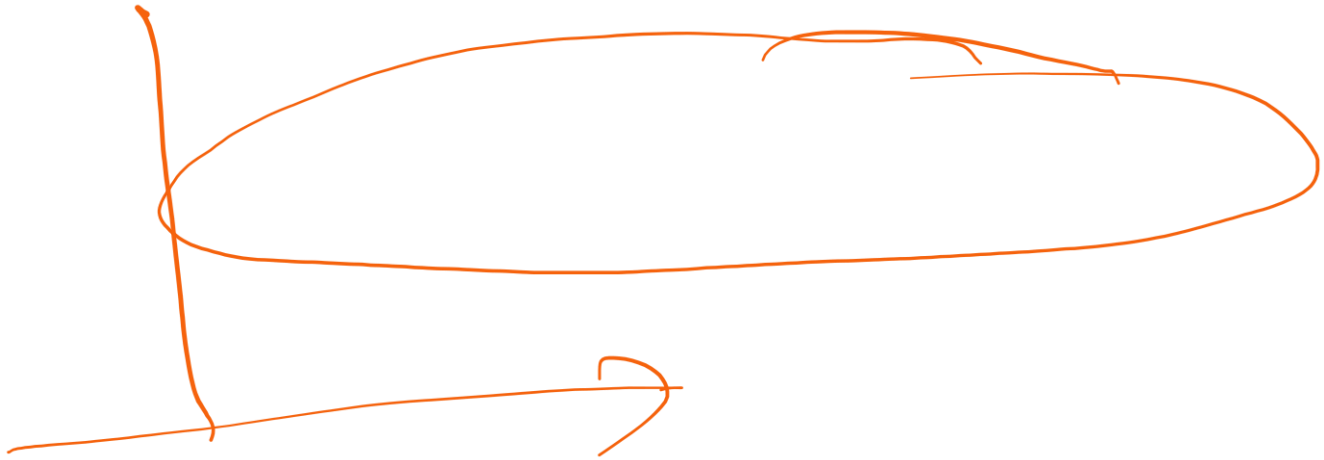
- Choosing a proper learning rate can be challenging
  - A too small learning rate leads to painfully slow convergence
  - A too-large learning rate can hinder convergence and cause the loss function to fluctuate around the minimum or even diverge
- SGD ends up with some oscillations in the vertical direction toward the minimum error
- These oscillations slow down the convergence process and make it harder to use larger learning rates, which could result in your algorithm overshooting and diverging





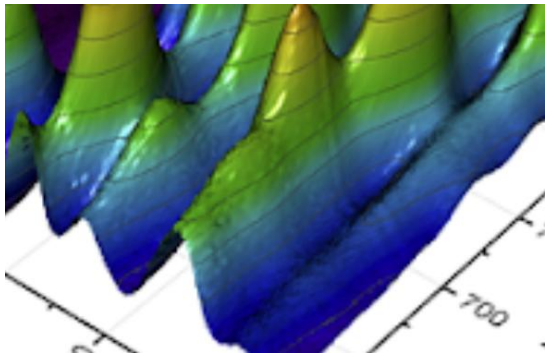
SGD Optimization

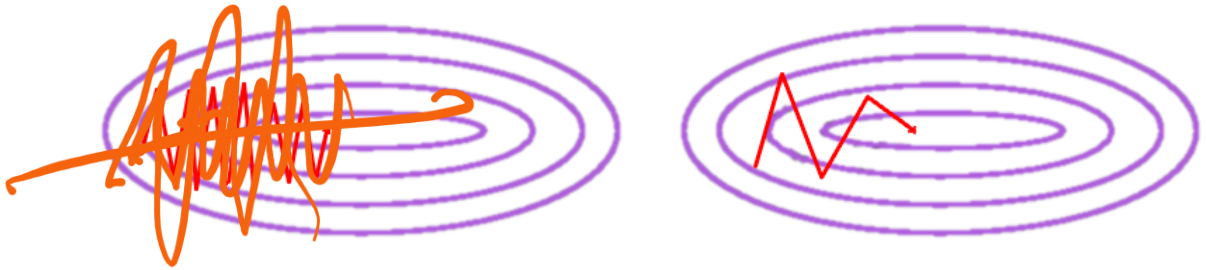
- To reduce these oscillations, a technique called *momentum* was invented
  - It lets the GD navigate along relevant directions and
  - Reduces the oscillation in irrelevant directions
- In other words, it makes learning algorithm slower in the vertical-direction oscillations
- But faster in the horizontal-direction progress that helps the optimizer reach the target minimum much faster



Momentum

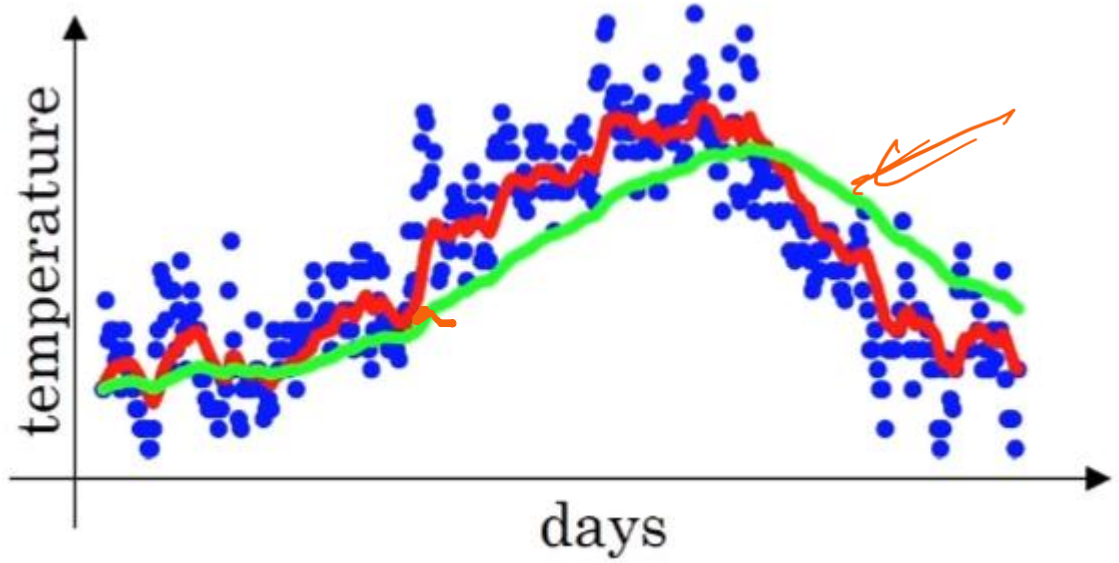
- By taking the average gradient, we can obtain a faster path to optimization
- This helps to dampen oscillations because gradients in opposite directions get canceled out



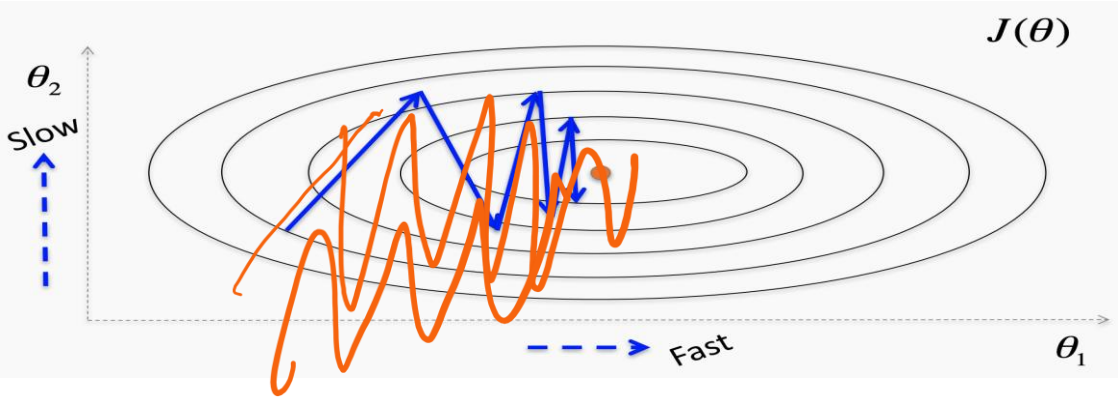


Exponentially weighted moving average

- Sort of a locally weighted linear regression
- Smooth the gradient values
- Update and convergence becomes much faster
- Classes of algorithms are defined based on this idea
- We use it for SGD with momentum



Gradient Descent with Momentum



On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta)dW$$

$$v_{db} = \beta v_{db} + (1-\beta)db$$

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: α, β $\beta = 0.9$

RMSProp

- We use an exponentially weighted average for gradient accumulation
- The big values are divided by big values, thus has a normalizing effect

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

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

Adam Optimizer

- Adam is a combination of RMSprop and momentum
- Adam refers to adaptive moment estimation, and it is the most popular optimizer used for neural networks today.
- Adam computes adaptive learning rates for each parameter
- In addition to storing an exponentially decaying average of past squared gradients like RMSProp, Adam also keeps an exponentially decaying average of past gradients, similar to momentum.
-

Adam Optimizer

1. Estimate the first and second moment of the gradient values

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

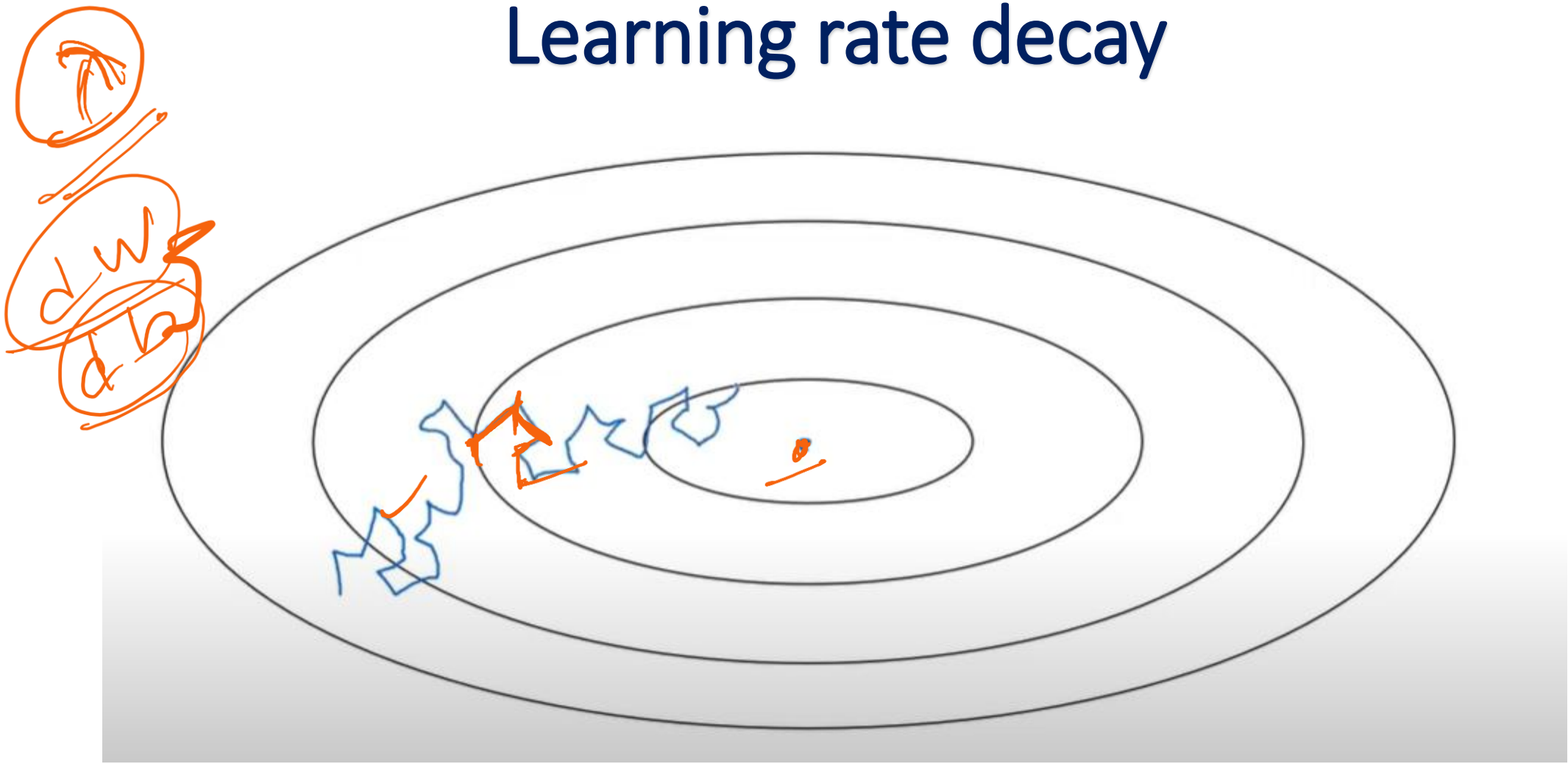
2. Bias correct the moment values

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

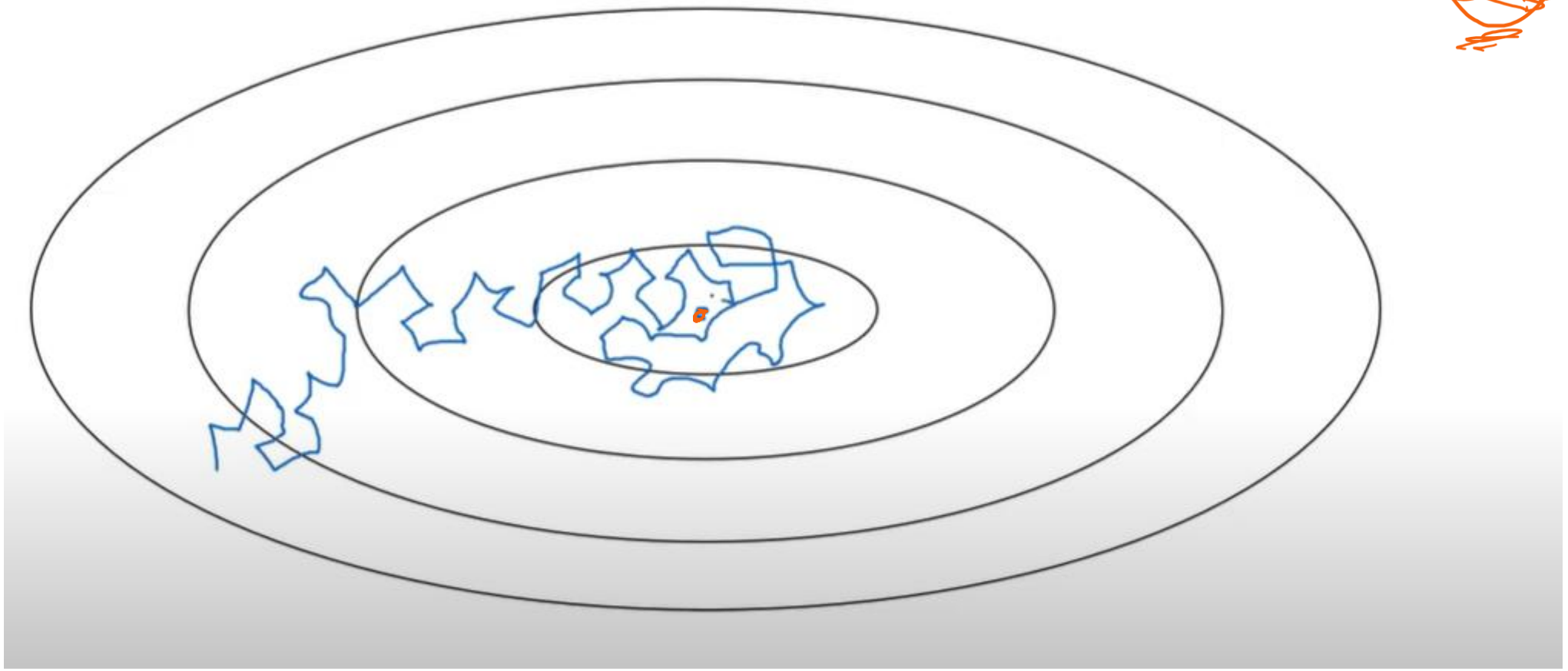
3. Update the parameters

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

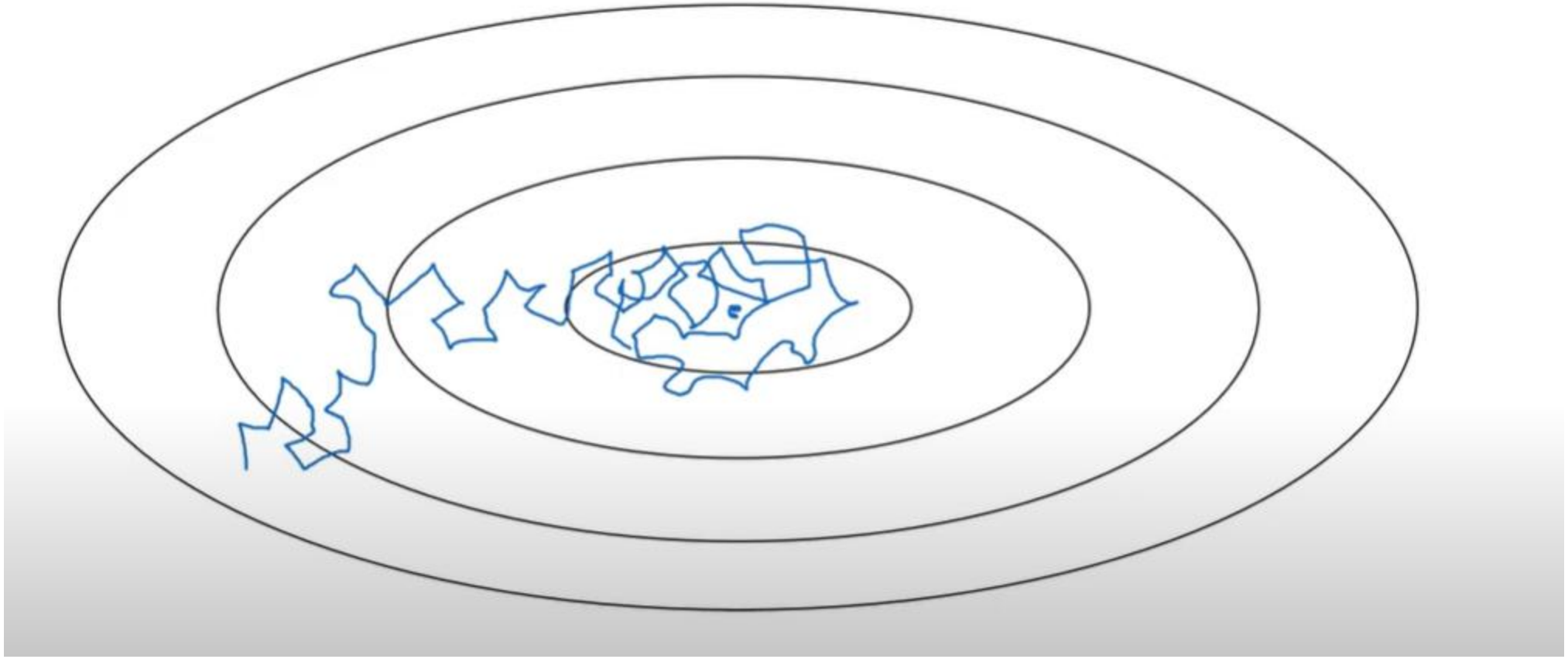
Learning rate decay



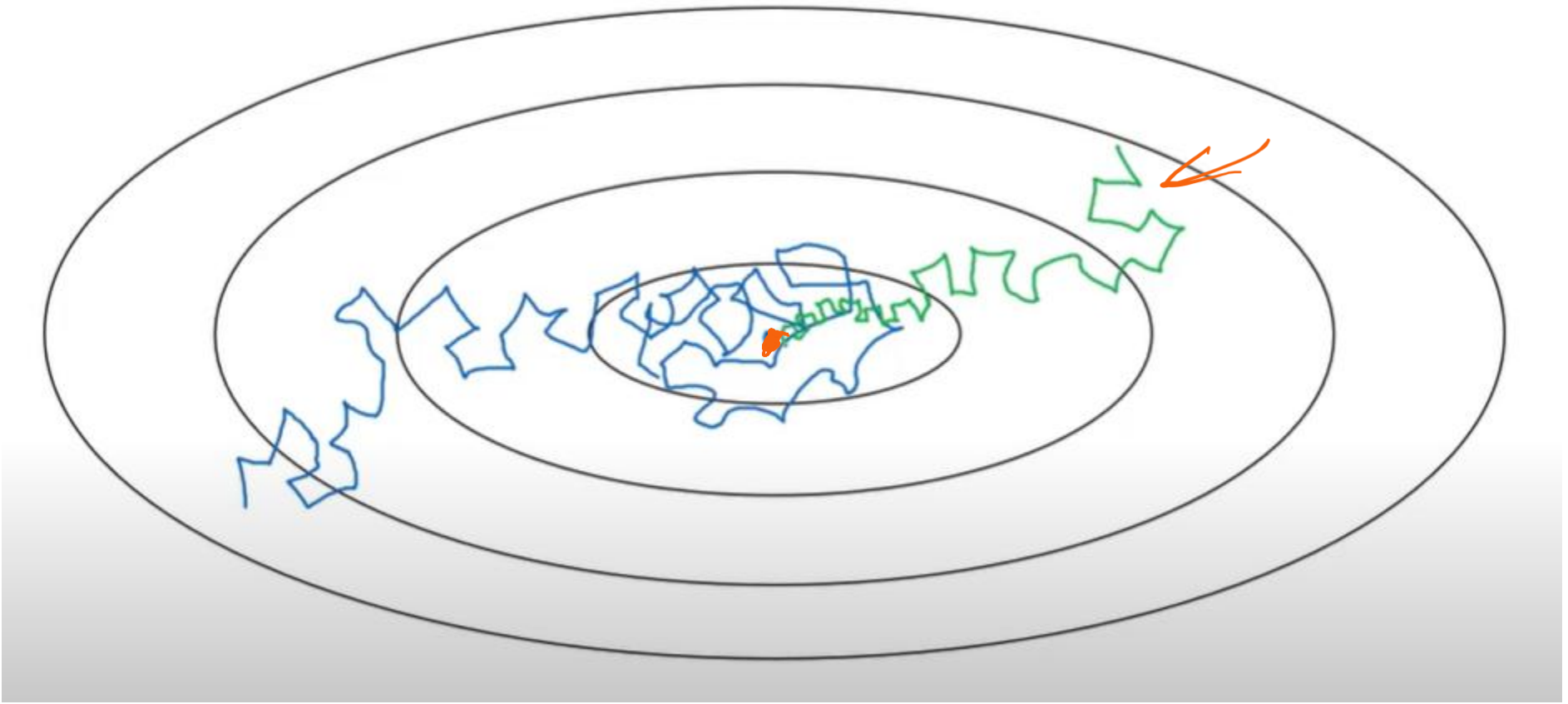
Learning rate decay



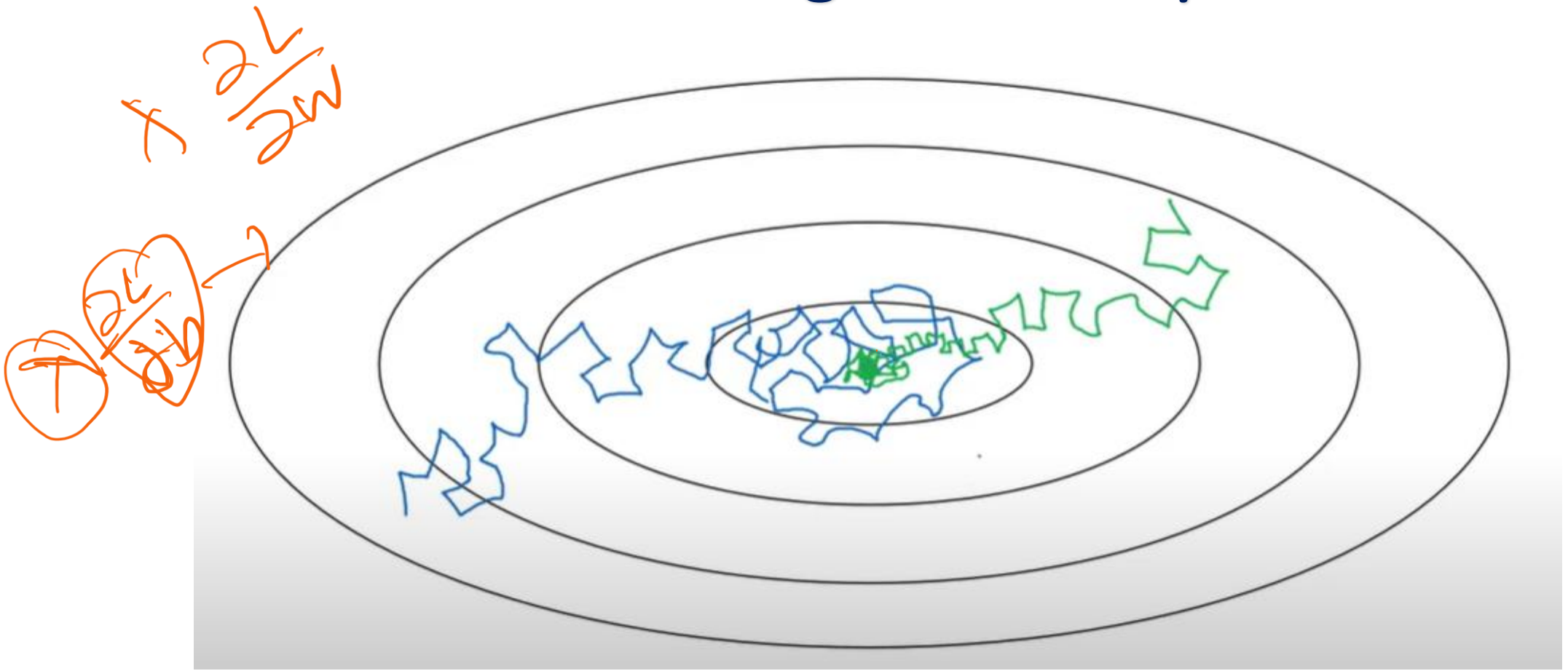
Learning rate decay



Learning rate decay



Learning rate decay



Parameter Initialization

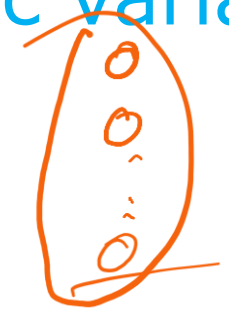
- We looked at how best to navigate the loss surface of the neural network objective function in order to converge to the global optimum (or an acceptable good local optimum)
- Now we will look at how we can manipulate the network itself in order to aid the optimization procedure.

Parameter Initialization

- Take for example a network that we initialize with all values of zero
  - What would happen in this scenario?
  - The network would actually not learn anything at all
  - Even after a gradient update, all the weights would still be zero, because of the inherent way we are calculating the gradient updates
- Imagine that we implemented this and worked out it was an issue, and then decided to set our network initialization all to the same value of 0.5. What would happen now?
  - The network would actually learn something, but we have prematurely prescribed some form of symmetry between neural units

Xavier Initialization

- Xavier initialization is a simple heuristic for assigning network weights. With each passing layer, we want the variance to remain the same. This helps us keep the signal from exploding to high values or vanishing to zero. In other words, we need to initialize the weights in such a way that the variance remains the same for both the input and the output.
- The weights are drawn from a distribution with zero mean and a specific variance



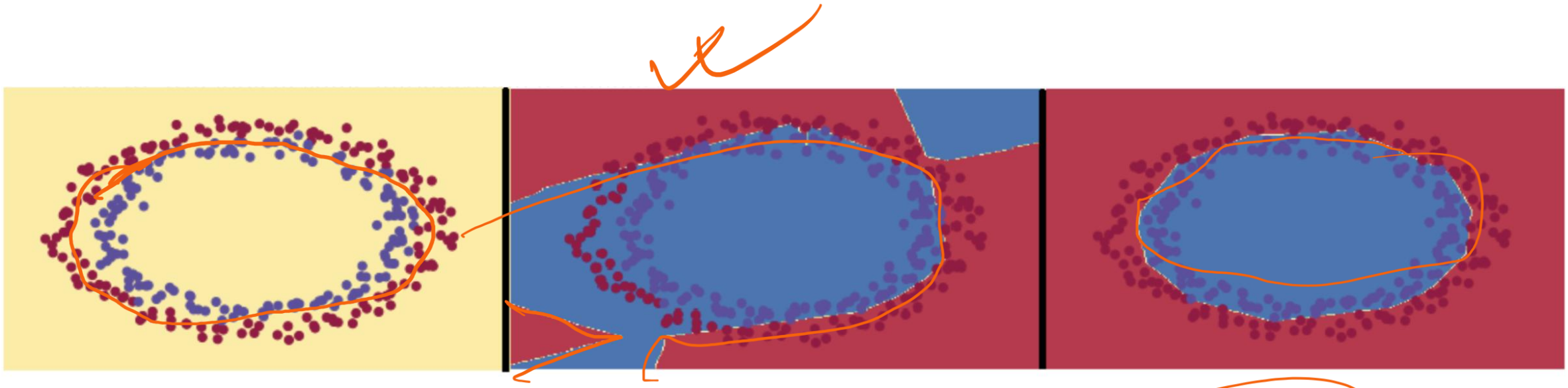
$$W_{ij} \sim N\left(0, \frac{1}{m}\right)$$

He normal initialization

- He normal initialization is essentially the same as Xavier initialization, except that the variance is multiplied by a factor of two.
- In this method, the weights are initialized keeping in mind the size of the previous layer which helps in attaining a global minimum of the cost function faster and more efficiently. The weights are still random but differ in range depending on the size of the previous layer of neurons. This provides a controlled initialization hence the faster and more efficient gradient descent.
-

$$W_{ij} \sim N\left(0, \frac{2}{m}\right)$$

Weight initialization



Zero Initailization

Random Initialization

He Initialization