Optimization Algorithms

FPT Education
FPT UNIVERSITY

Learning Objectives:

- Apply optimization methods such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam
- Use random minibatches to accelerate convergence and improve optimization
- Describe the benefits of learning rate decay and apply it to your optimization

Optimization Algorithms

Optimization Algorithms

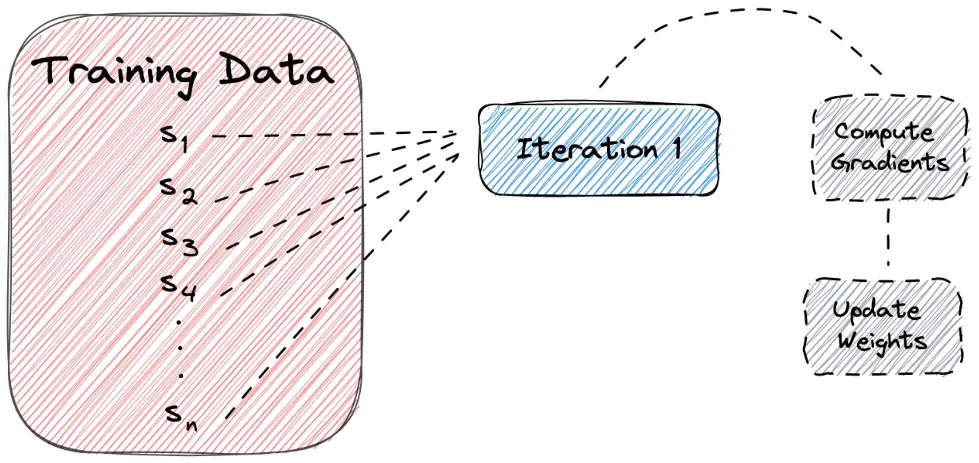
Mini-batch gradient descent

Mini-batch gradient descent

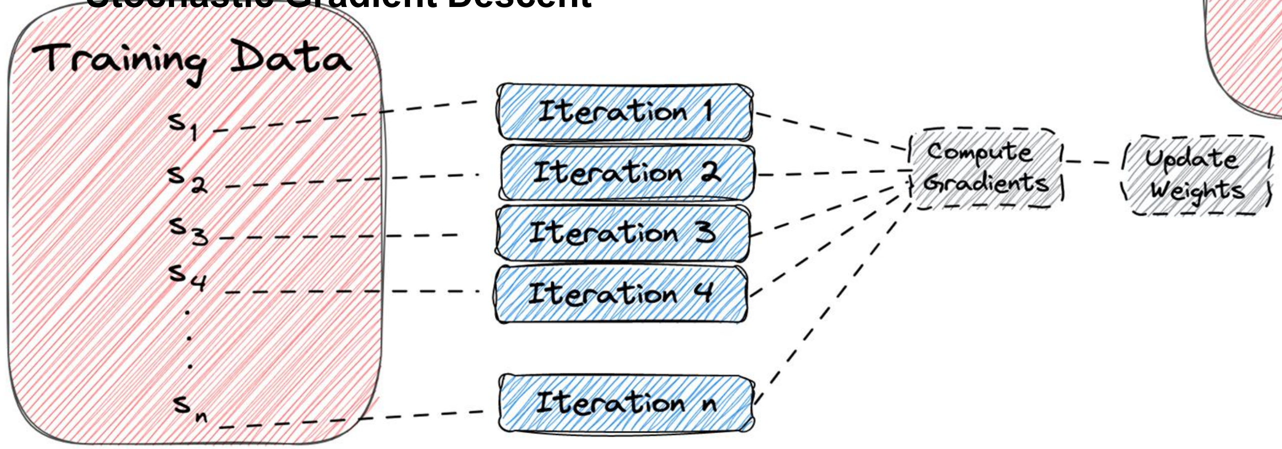
- In machine learning, gradient descent is an optimization technique used for computing the model parameters (coefficients and bias) for algorithms like linear regression, logistic regression, neural networks, etc. In this technique, we repeatedly iterate through the training set and update the model parameters in accordance with the gradient of the error with respect to the training set. Depending on the number of training examples considered in updating the model parameters, we have 3-types of gradient descents:
  - Batch Gradient Descent: Parameters are updated after computing the gradient of the error with respect to the entire training set
  - Stochastic Gradient Descent: Parameters are updated after computing the gradient of the error with respect to a single training example
  - Mini-Batch Gradient Descent: Parameters are updated after computing the gradient of the error with respect to a subset of the training set

Mini-batch gradient descent

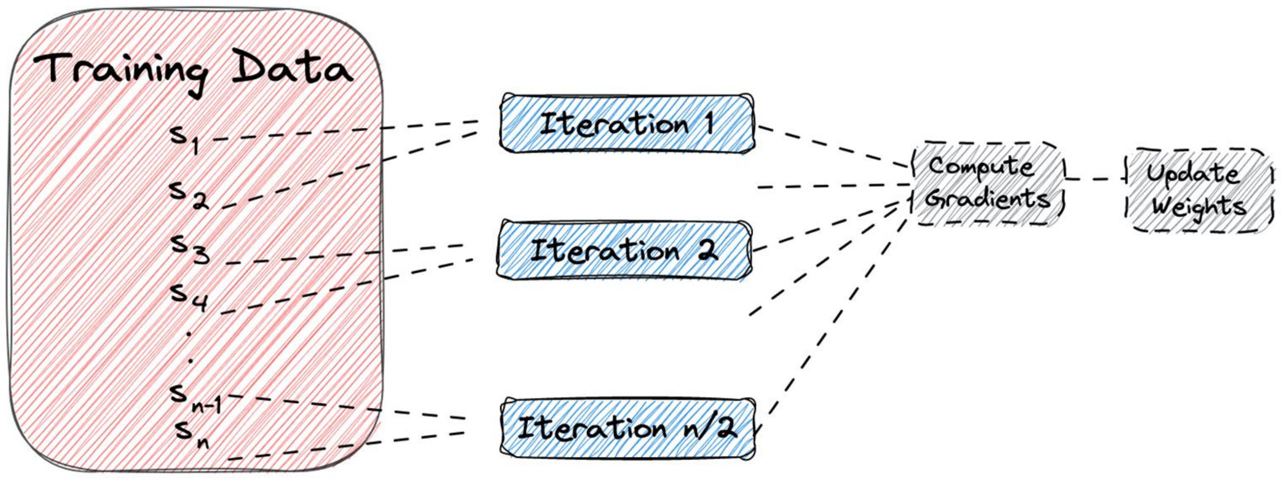
Batch Gradient Descent



Stochastic Gradient Descent



Mini-Batch Gradient Descent



Mini-batch gradient descent

Batch Gradient Descent	Stochastic Gradient Descent	Mini-Batch Gradient Descent
Since the entire training data is considered before taking a step in the direction of gradient, therefore it takes a lot of time for making a single update.	Since only a single training example is considered before taking a step in the direction of gradient, we are forced to loop over the training set and thus cannot exploit the speed associated with vectorizing the code.	Since a subset of training examples is considered, it can make quick updates in the model parameters and can also exploit the speed associated with vectorizing the code.
It makes smooth updates in the model parameters	It makes very noisy updates in the parameters	Depending upon the batch size, the updates can be made less noisy – greater the batch size less noisy is the update

- In this lesson, we only study mini-batch gradient descent.

Mini-batch gradient descent

- An epoch means that we have passed each sample of the training set one time through the network to update the parameters. Generally, the number of epochs is a hyperparameter that defines the number of times that gradient descent will pass the entire dataset.
- If we look at the previous methods, we can see that:
  - In batch gradient descent, one epoch corresponds to a single iteration through the entire training dataset.
  - In stochastic gradient descent, one epoch corresponds to n iterations, where n is the number of training samples.
  - In mini-batch gradient descent, one epoch corresponds to n/b iterations, where b is the size of the mini-batch.
- A batch is equal to the total training data used in batch gradient descent to update the network's parameters.
- A mini-batch is a subset of the training data used in each iteration of the training algorithm in mini-batch gradient descent.

Mini-batch gradient descent

- Let’s assume that we have a dataset with n = 2000 samples, and we want to train a deep learning model using gradient descent for 10 epochs and mini-batch size b = 4:
  - In batch gradient descent, we’ll update the network’s parameters (using all the data) 10 times which corresponds to 1 time for each epoch.
  - In stochastic gradient descent, we’ll update the network’s parameters (using one sample each time) 2000*10 = 20000 times which corresponds to 2000 times for each epoch.
  - In mini-batch gradient descent, we’ll update the network’s parameters (using b = 4 samples each time) (2000/4)*10 = 5000 times that corresponds to 2000/4 = 500 times for each epoch.

- **Given L(x,y) = x^3 + y^2 – 8y +6x -12 and the initial point P0 = (1,2).** Performing the gradient descent algorithm with learning rate = 0.1, the first iteration will lead us the point P1 which is:
- Let $f(x,y)$ = (2x^2+3y^2–2xy–20x+4), the minimum value of $f(x,y)$ is:

Mini-batch gradient descent

- Mini-batch gradient descent accelerates deep learning training on large datasets by processing smaller subsets, or mini-batches, at a time. It employs a for loop to iterate through each mini-batch, performing forward and backward propagation to update weights and compute gradients, making the training process more efficient.

Mini-batch gradient descent

- Vectorization allows you to efficiently compute on m examples. But m if is really large that can still be slow. A solution to this is only ingest a small fixed amount of examples (1000, for example) and use each one to iteratively compute the errors.
  - Notation:
    - $x^{(i)}$ ith example in the test set
    - $Z^{[l]}$ lth layer in the neural network
    - $X^{\{i\}}$ ith batch in the mini-batch test set. $X^{\{i\}}$. shape = (n_x, m)
- Mini-Batch algorithm pseudo code:

```
for t = 1:No_of_batches # this is called an epoch
      $A^L$, caches = forward_prop($X^{\{t\}}$, $Y^{\{t\}}$)
      cost = compute_cost($A^L$, $Y^{\{t\}}$)
      grads = backward_prop($A^L$, caches)
      update_parameters(grads)
```

The code inside an epoch should be vectorized.

Mini-batch gradient descent

- If you have 5 million training samples total and each of these little mini batches has a thousand examples, that means you have 5,000 mini-batches.

$$\underset{(n_x\,,\,m)}{X} = [\underbrace{x^{(1)}, x^{(2)}, x^{(3)}, \ldots, x^{(1{,}000)}}_{\underset{(n_x\,,\,1000)}{X^{\{1\}}}}, \underbrace{x^{(1{,}001)}, \ldots, x^{(2{,}000)}}_{\underset{(n_x\,,\,1000)}{X^{\{2\}}}}, \ldots, \underbrace{x^{(4{,}999{,}000)}, \ldots, x^{(5{,}000{,}000)}}_{\underset{(n_x\,,\,1000)}{X^{\{5{,}000\}}}}]$$

$$\underset{(1,\,m)}{Y} = [\underbrace{y^{(1)}, y^{(2)}, y^{(3)}, \ldots, y^{(1{,}000)}}_{\underset{(1\,,\,1000)}{Y^{\{1\}}}}, \underbrace{y^{(1{,}001)}, \ldots, y^{(2{,}000)}}_{\underset{(1\,,\,1000)}{Y^{\{2\}}}}, \ldots, \underbrace{y^{(4{,}999{,}000)}, \ldots, y(5{,}000{,}000)}_{\underset{(1\,,\,1000)}{Y^{\{5{,}000\}}}}]$$

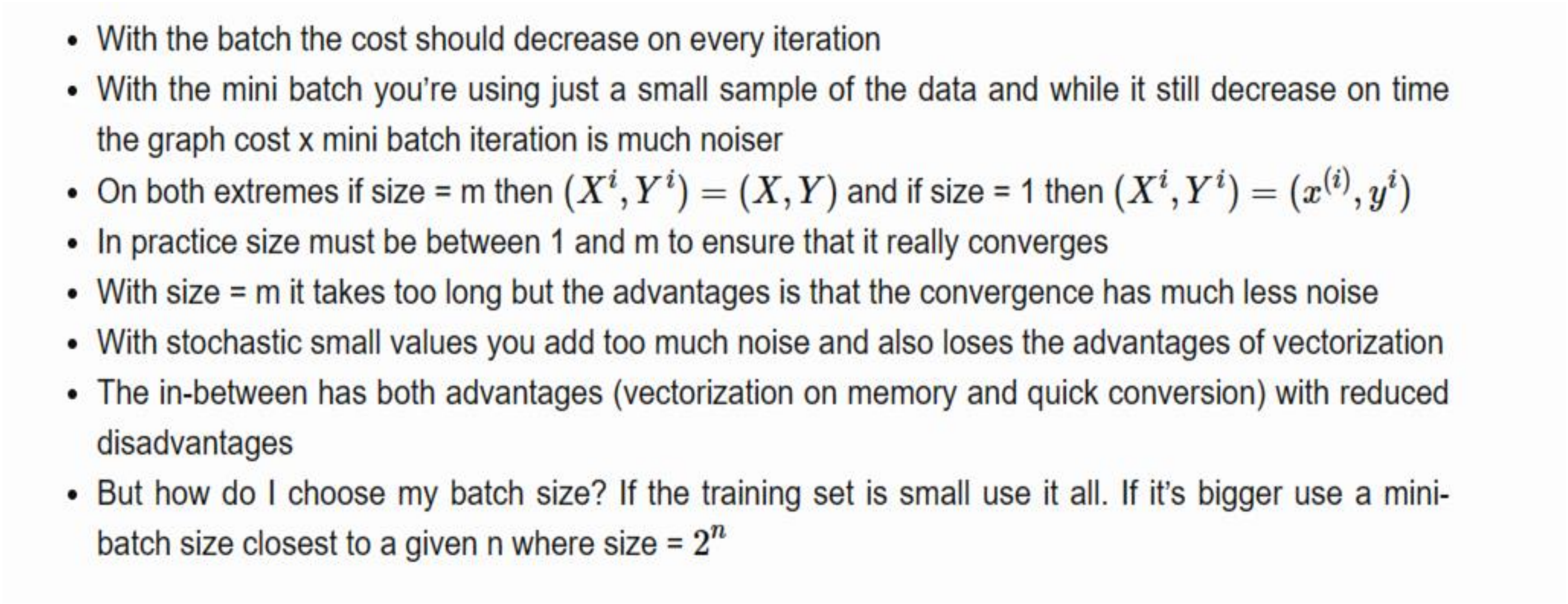
Mini-batch gradient descent

- Mini-Batch algorithm pseudo code:

```
for t = 1:5,000
      forward prop on $X^{\{t\}}$
            $Z^{[1]} = W^{[1]} * X^{\{t\}} + b^{[1]}$
                $A^{[1]} = g(Z^{[1]})$
                         ⋮
                $A^{[l]} = g(Z^{[l]})$
      compute_cost:
            $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2*1000} \sum_l \|W^2\|^l$
      backward prop to compute grads and update parameters:
      $W^{[l]} = W^{[l]} - \alpha * dW^{[l]}$ , $b^{[l]} = b^{[l]} - \alpha * db^{[l]}$
```

Vectorized implementation (1,000 examples)

Understanding mini-batch gradient descent

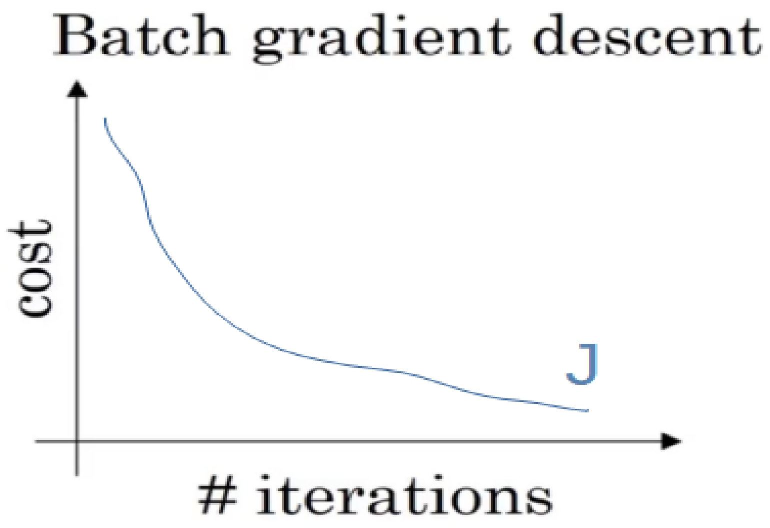


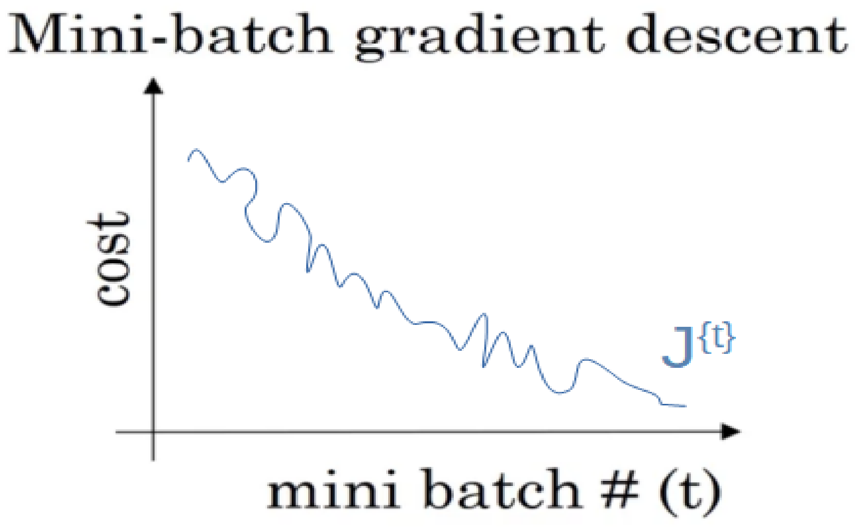
- With the batch the cost should decrease on every iteration
- With the mini batch you're using just a small sample of the data and while it still decrease on time the graph cost x mini batch iteration is much noiser
- On both extremes if size = m then $(X^{i}, Y^{i}) = (X, Y)$ and if size = 1 then $(X^{i}, Y^{i}) = (x^{(i)}, y^{i})$
- In practice size must be between 1 and m to ensure that it really converges
- With size = m it takes too long but the advantages is that the convergence has much less noise
- With stochastic small values you add too much noise and also loses the advantages of vectorization
- The in-between has both advantages (vectorization on memory and quick conversion) with reduced disadvantages
- But how do I choose my batch size? If the training set is small use it all. If it's bigger use a mini-batch size closest to a given n where size = 2^{n}

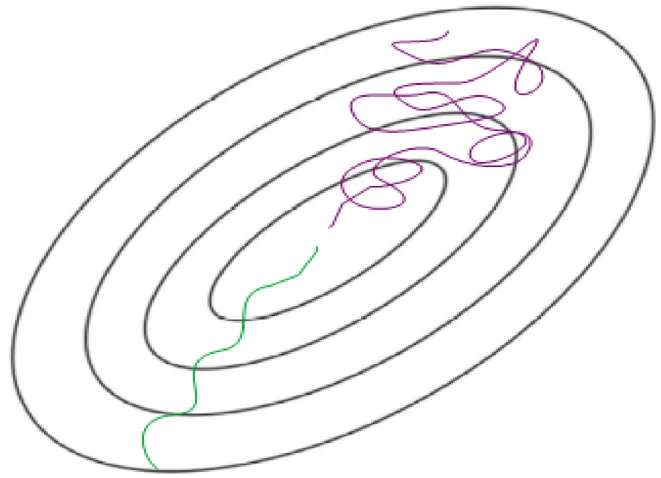
Optimization Algorithms

Understanding mini-batch gradient descent

Training with mini batch gradient descent







If mini-batch size = 1: Stocastic Gradient Descent = every example (row) it is used as mini-batch: lose sped-up from vectorization.

If mini-batch size = m: Batch Gradient Descent: $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$ = the entire training set is used: size = n: too long interations.

Choosing your mini-batch size

- Let's see various cases:
- If the mini-batch size = m:
  - It is a batch gradient descent where all the training examples are used in each iteration. It takes too much time per iteration.
- If the mini-batch size = 1:
  - It is called stochastic gradient descent, where each training example is its own mini-batch. Since in every iteration we are taking just a single example, it can become extremely noisy and takes much more time to reach the global minima.
- If the mini-batch size is between 1 to m:
  - It is mini-batch gradient descent. The size of the mini-batch should not be too large or too small.

Choosing your mini-batch size

- Below are a few general guidelines to keep in mind while deciding the mini-batch size:
  - If the training set is small, we can choose a mini-batch size of m<2000
  - For a larger training set, typical mini-batch sizes are: 64, 128, 256, 512
  - Make sure that the mini-batch size fits your CPU/GPU memory. If data fits in CPU/GPU, we can leverage the speed of processor cache, which significantly reduces the time required to train a model.

FPT Education

FPT UNIVERSITY

Optimization Algorithms

Exponentially weighted averages

Exponentially weighted averages

- Exponentially weighted averages (EWAs) are a way to compute a moving average of a sequence of data points that assigns exponentially decreasing weights to the previous values in the sequence. This means that more recent data points are given more weight than older ones.
- The formula for computing the exponentially weighted average is given by

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

where β is a parameter between 0 and 1 that controls the weight given to the previous average, V_{t-1}, and the current data point, θ_t. The parameter β is often referred to as the smoothing factor and is a hyperparameter that can be tuned to obtain the desired results.

Exponentially weighted averages

- The parameter β in exponentially weighted averages determines the balance between emphasizing recent data points and smoothing the sequence.
  - A higher β, near 1, leads to a smoother sequence with slower adaptation to changes.
  - A lower β, near 0, results in a more responsive but volatile sequence.
- Exponentially weighted averages are widely used in time series analysis, finance, and machine learning for efficiently smoothing data sequences and responding to changes without overfitting.

Exponentially weighted averages

- Consider example of the daily temperature of London from the last year is taken as the sequence of data points.
- The goal is to compute the moving average of the temperature over a certain window of time, which is done by taking an exponentially weighted average.

Exponentially weighted averages

Below is a sample of hypothetical temperature data collected for an entire year:

θ_1= 40ºF

θ_2 = 49ºF

θ_3 = 45ºF

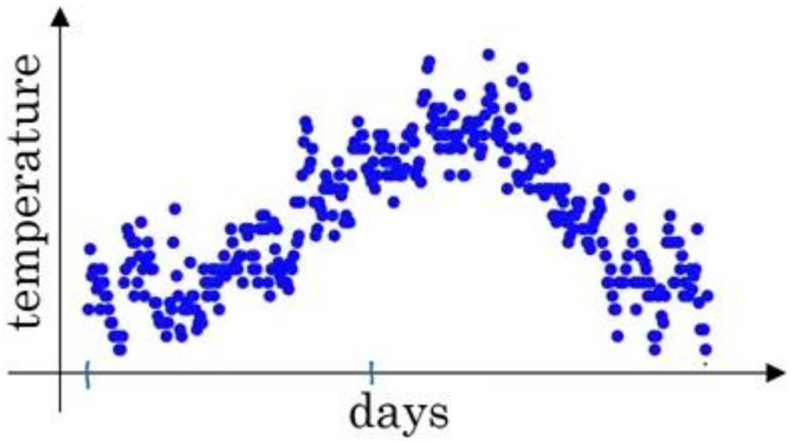
$\vdots$

θ_{180} = 60ºF

θ_{181} = 56ºF

$\vdots$

The below plot neatly summarizes this temperature data for us:



Exponentially weighted averages

Exponentially weighted average, or exponentially weighted moving average, computes the trends. We will first initialize a term as 0:

$V_0 = 0$

Now, all the further terms will be calculated as the weighted sum of V0 and the temperature of that day:

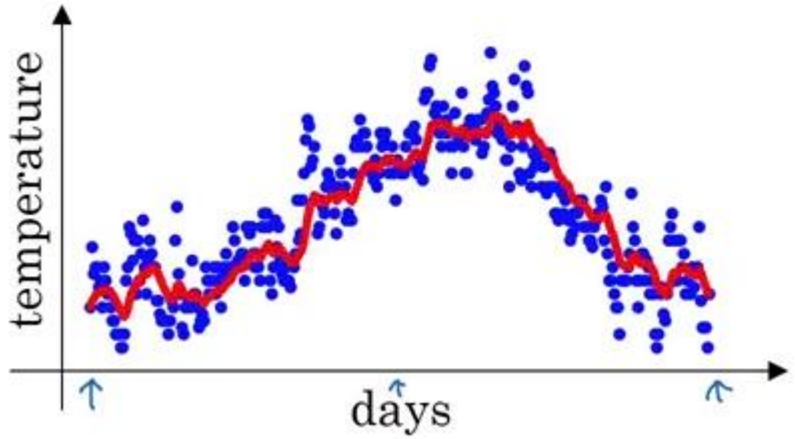
$V_1 = 0.9V_0 + 0.1\theta_1$

$V_2 = 0.9V_1 + 0.1\theta_2$

And so on. A more generalized form of exponentially weighted average can be written as:

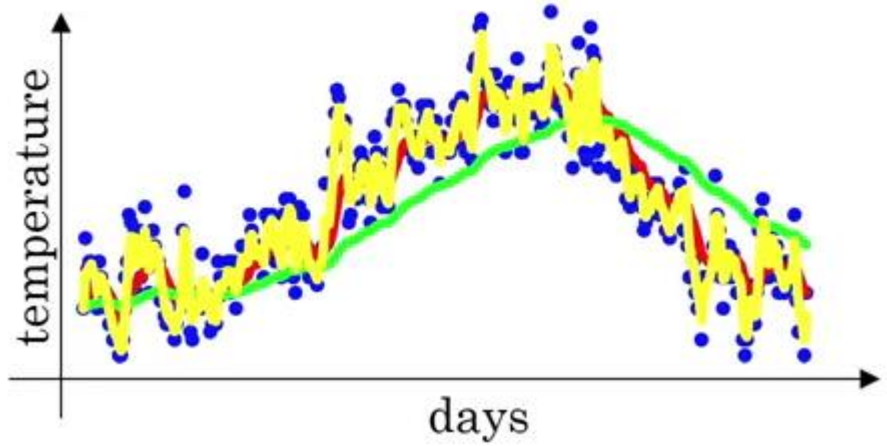
$V_t = \beta V_{t-1} + (1 - \beta)\theta_t$

Using this equation for trend, the data will be generalized as:



Exponentially weighted averages

- The β value in this example is 0.9, which means Vt is an approximation of average over 1/(1-β) days, i.e., 1/(1-0.9) = 10 days temperature. Increasing the value of β will result in approximating over more days, i.e., taking the average temperature of more days. If the β value is small, i.e., we use only 1 day's data for approximation, the predictions become much more noisy:

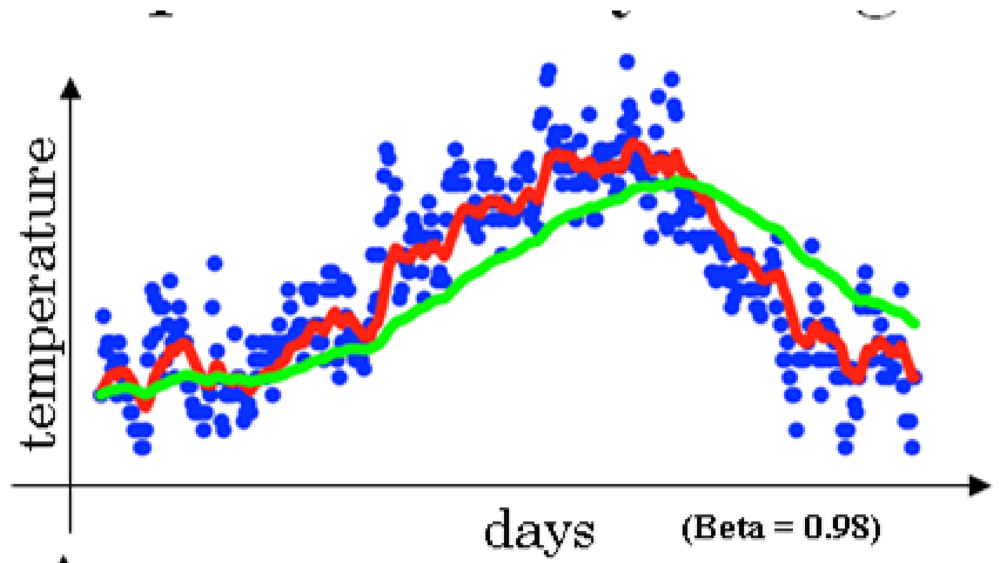


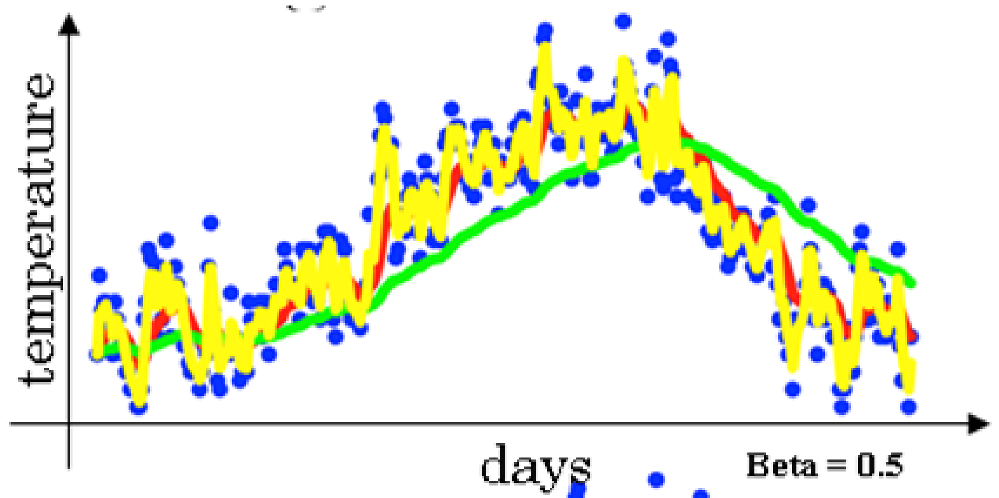
- Here, the green line is the approximation when β = 0.98 (using 50 days) and the yellow line is when β = 0.5 (using 2 days). It can be seen that using small β results in noisy predictions.

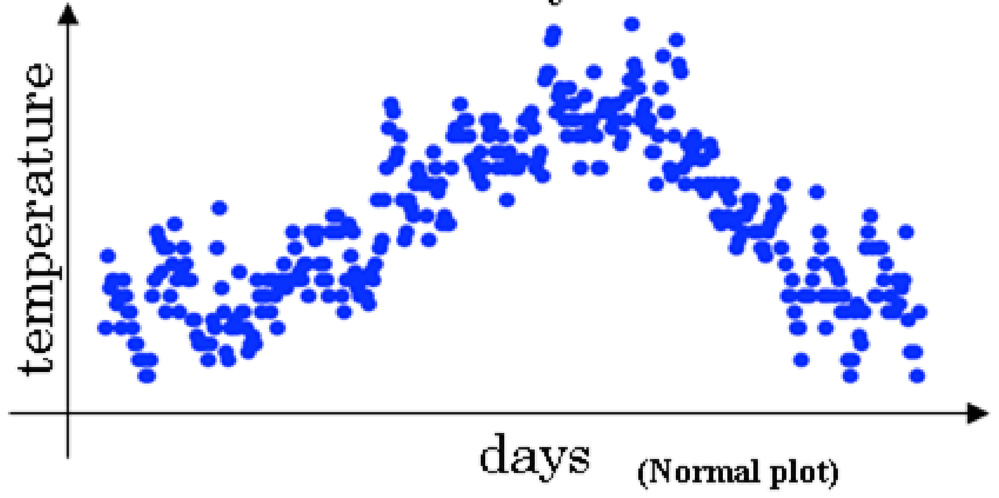
Exponentially weighted averages

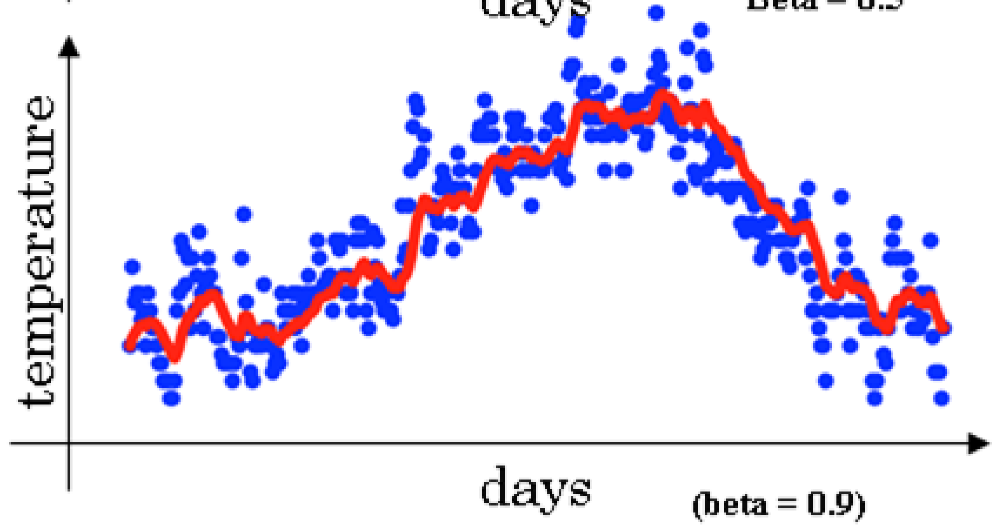
- The choice of β depends on the nature of the data and the desired level of smoothing.
- By varying the value of β, one can obtain different levels of smoothing and adaptation to changes in the data.
- The exponentially weighted average is a widely used technique in signal processing, finance, and machine learning, where it is used in optimization algorithms that are faster than gradient descent.

Exponentially weighted averages









Exponentially weighted averages

- The equation of exponentially weighted averages is given by:
- $\mathbf{V_t = \beta V_{t-1} + (1 - \beta)\boldsymbol{\theta}_t}$
- Let's look at how we can implement this:

Initialize $V_0 = 0$

- Repeat

```
        {
        get next $\boldsymbol{\theta}_t$
        $V_0 = \beta V_0 + (1 - \beta)\boldsymbol{\theta}_t$
        }
```

This step takes a lot less memory **as we are overwriting the previous values**. Hence, it is a computational, as well as memory efficient, process.

FPT Education

FPT UNIVERSITY

Optimization Algorithms

Understanding exponentially weighted averages

Exponentially weighted averages (EWA)

FPT Education
FPT UNIVERSITY

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

$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$

Also, if beta = 0.9 the weight decays to about a third by 10th iteration.

Proof: (1− epsilon)$^{(1/\text{epsilon})}$ ≈ 1/e , where e is 2.718281828459045…
(Euler's number)

epsilon = 1 - beta
of beta = 0.9, epsilon = 1 - beta = 0.1

(1-0.1)$^{(1/0.1)}$ = 0,9^{10} = 0.35 ≈ 1/e
Interpretation: It takes about 10 days for height to decay to 1/3rd

Generalizing,

$$v_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9v_{97})) + \cdots$$
$$= 0.1\theta_{100} + 0.1 \times 0.9\theta_{99} + 0.1 \times (0.9)^2\theta_{98} + 0.1 \times (0.9)^3\theta_{97} + \cdots$$

v_{100} is basically an element-wise computation of two metrics/functions — one an exponential decay function containing diminishing values (0.9, 0.9^{22}, 0.^{93}, ……and another with all the elements of θt.

Exponentially weighted averages

- The EWA algorithm computes a time series moving average with exponentially decaying weights, adapting faster with smaller beta and slower with larger beta.
- Its efficiency lies in minimal memory usage, making it suitable for computing averages across numerous variables.

FPT Education

FPT UNIVERSITY

Bias correction in exponentially weighted average

Optimization Algorithms

Bias correction in exponentially weighted average

- Exponentially weighted moving averages play a vital role in optimizing neural networks during training. They involve calculating a weighted average that prioritizes recent values while smoothing out noise

- To enhance accuracy, a bias correction technique is often applied, adjusting the initialization and computation using the formula.

$$V_0 = 0$$

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

Bias correction in exponentially weighted average

- When we calculate V_t using this formula, it can result in a bias towards lower values, especially during the initial phase of learning. This is because the first few values of V_t are calculated using a small number of $\boldsymbol{\theta_t}$ values, which can cause the moving average to underestimate the true value of the parameter.
- To correct for this bias, we can modify the estimate of V_t by dividing it by a correction factor, which is **$1-\beta^t$** :
- $\mathbf{V_t = [\beta V_{t-1} + (1-\beta)\theta_t\,] / (1-\beta^t)}$
- This correction factor ensures that the estimate is normalized and takes into account the fact that we are using a smaller number of $\boldsymbol{\theta_t}$ values during the initial phase of learning. By doing this, we can obtain a more accurate estimate of the parameter during the initial phase of learning.

Bias correction in exponentially weighted average

- Bias correction in machine learning isn't always applied due to computational costs, and some initial bias may be acceptable. However, when early-phase accuracy is crucial, bias correction becomes valuable.
- These concepts are essential for optimizing neural network training and enhancing model accuracy and efficiency.

FPT Education

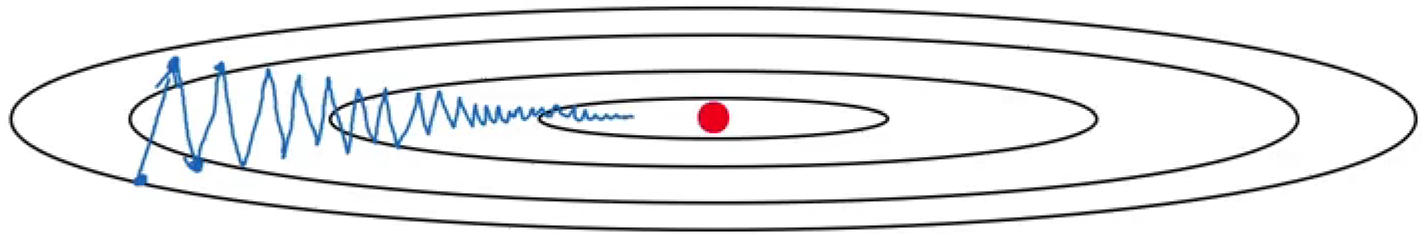
FPT UNIVERSITY

Optimization Algorithms

Gradient descent with momentum

Gradient descent with momentum

- An algorithm called momentum, or gradient descent with momentum that almost always works faster than the standard gradient descent algorithm.
- The underlying idea of gradient descent with momentum is to calculate the exponential weighted average of gradients and use them to update weights. Suppose we have a cost function whose contours look like this:



- The red dot is the global minima, and we want to reach that point. Using gradient descent, the updates will look like:

Gradient descent with momentum

- One more way could be to use a larger learning rate. But that could result in large upgrade steps, and we might not reach global minima. Additionally, too small a learning rate makes the gradient descent slower. We want a slower learning in the vertical direction and a faster learning in the horizontal direction which will help us to reach the global minima much faster.
- Let’s see how we can achieve it using momentum:

```
On iteration t:
Compute dW, dB on current mini-batch using momentum
$V_{dW} = \beta * V_{dW} + (1 - \beta) * dW$
$V_{db} = \beta * V_{db} + (1 - \beta) * db$
Update weights
$W = W - \alpha * V_{dW}$
$b = b - \alpha * V_{db}$
```

Gradient descent with momentum

- The hyperparameters α and β in the given equation control momentum and friction, respectively, in the gradient descent algorithm. The terms dW and db contribute momentum, while Vdw and Vdb represent velocity. β acts as friction, preventing excessive acceleration.
- The algorithm ensures that steps are smaller oscillations in the vertical direction, emphasizing quick movement horizontally.
- The terms (1–β) resemble acceleration, and β resembles velocity on a specific point.
- While bias correction can be applied, its impact diminishes quickly, making it less practical.
- β becomes a new hyperparameter influencing the learning rate α.

FPT Education

FPT UNIVERSITY

Optimization Algorithms

RMSprop

RMSprop (Root Means Square Propagation)

- RMSprop is an optimization algorithm that can speed up gradient descent. It works by keeping an exponentially weighted average of the squares of the derivatives for each parameter. This average is then used to adjust the learning rate for each parameter during the update step.

- RMSprop is particularly effective at reducing oscillations in the vertical direction, where the function is sloped more steeply. This is achieved by dividing the update for each parameter by a larger number in the vertical direction than in the horizontal direction.

RMSprop

```
Compute $dW$ and $db$ on current mini-batch
$S_{dW} = \beta S_{dW} + (1-\beta) dW^2$ <- element-wise operation
$S_{db} = \beta S_{db} + (1-\beta) db^2$ <- element-wise operation
$W = W - \alpha \frac{dW}{\sqrt{S_{dW}+\varepsilon}}$
$b = b - \alpha \frac{db}{\sqrt{S_{db}+\varepsilon}}$
```

- Intuition is that in dimensions where you're getting these oscillations, you end up computing a larger sum. A weighted average for these squares and derivatives, and so you end up dumping out the directions in which there are these oscillations.
- The ε is present in the denominator to avoid division by 0

RMSprop

- In practice, RMSprop is used in high-dimensional parameter spaces, where the oscillations may be present in a subset of parameters. By taking the squared derivatives of these parameters, RMSprop can dampen the oscillations and speed up the learning process.

- To ensure numerical stability, a very small epsilon value is added to the denominator to prevent division by zero or very small numbers.

- RMSprop was first introduced in a Coursera course taught by Jeff Hinton and has since become a widely used optimization algorithm in deep learning.

Optimization Algorithms

Adam optimization algorithm

Adam optimization algorithm

- The field of deep learning has seen many optimization algorithms proposed over time, but few have been shown to work well across a wide range of neural networks. Gradient descent with momentum has been a popular and effective algorithm, and RMSprop and Adam have been two algorithms that have also been shown to work well across a wide range of deep learning architectures.
- Adam (stands for Adaptive Moment Estimation) is an optimization algorithm that combines the effect of gradient descent with momentum and gradient descent with RMSprop. The algorithm has several hyperparameters, including the learning rate α, β_1, β_2, and epsilon. The default values for β_1 and β_2 are 0.9 and 0.999, respectively, and the recommended value for ϵ is 10^{-8}.

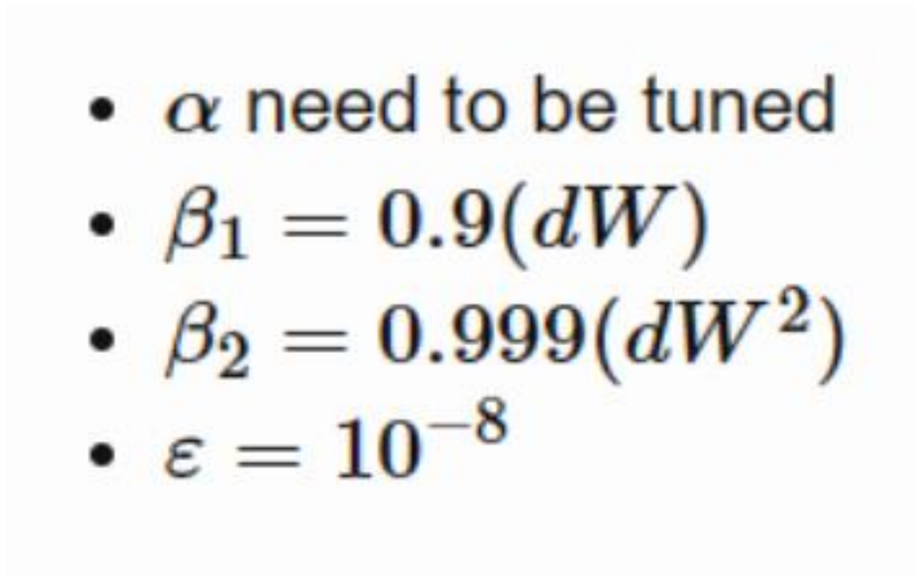
Adam optimization algorithm

```
$V_{dW} = 0, S_{dW} = 0, V_{db} = 0, S_{db} = 0$
On iteration t:
*Compute dW, dB on current mini-batch using momentum and RMSprop*
$V_{dW} = \beta_1 * V_{dW} + (1 - \beta_1) * dW$
$V_{db} = \beta_1 * V_{db} + (1 - \beta_1) * db$

$S_{dW} = \beta_2 * S_{dW} + (1 - \beta_2) * dW^2$
$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * db^2$

*Apply bias correction*
$V_{dW}^{corrected} = V_{dW} / (1 - \beta_1^t)$
$V_{db}^{corrected} = V_{db} / (1 - \beta_1^t)$

$S_{dW}^{corrected} = S_{dW} / (1 - \beta_2^t)$
$S_{db}^{corrected} = S_{db} / (1 - \beta_2^t)$

*Update weights*
$W = W - \alpha * (V_{dW}^{corrected} / S_{dW}^{corrected} + \varepsilon)$
$b = b - \alpha * (V_{db}^{corrected} / S_{db}^{corrected} + \varepsilon)$
```

Hyperparameters

- There are a range of hyperparameters used in Adam and some of the common ones are:
- Learning rate α: needs to be tuned
- Momentum term β1: common choice is 0.9
- RMSprop term β2: common choice is 0.999
- ε: 10^{-8}
- Adam helps to train a neural network model much more quickly than the techniques we have seen earlier.

Hyperparameters



- α need to be tuned
- $\beta_1 = 0.9(dW)$
- $\beta_2 = 0.999(dW^2)$
- $\varepsilon = 10^{-8}$

Adam Cotes

Adam optimization algorithm

- β_1 is used to compute the mean of the derivatives (called the first moment) and β_2 is used to compute the exponentially weighted average of the squares (called the second moment). The bias correction is also implemented in Adam to improve numerical stability.

- Hyperparameter tuning is an important step in optimizing the performance of deep learning models. In practice, it is common to use default values for β_1, β_2, and ϵ, and to tune the learning rate α by trying a range of values to see what works best.

FPT Education

FPT UNIVERSITY

Optimization Algorithms

Learning rate decay

Learning rate decay

- Learning rate decay is a technique used to gradually reduce the learning rate α over time during the training process of a neural network.
- The intuition behind this technique is that as learning approaches convergence, having a slower learning rate allows for smaller and more precise steps towards the minimum instead of oscillating around it.
- Initially, when the learning rate is not very small, training will be faster. If we slowly reduce the learning rate, there is a higher chance of coming close to the global minima.
- Learning rate decay can be given as:
- $\alpha = [1 / (1 + decay\_rate * epoch\_number)] * \alpha_0$
- The *decay rate* becomes another hyperparameter that you may need to tune. There are several other ways to implement learning rate decay, such as exponential decay or discrete staircase.

Learning rate decay

- Let’s understand it with an example. Consider:
- α0 = 0.2
- decay_rate = 1
- α_1 = [1/(1+1)]*0.2 = 0.1
- α_2 = [1/(1+2)]*0.2 = 0.067
- α_3 = 0.05
- α_4 = 0.04

Learning rate decay

- This is how, after each epoch, there is a decay in the learning rate which helps us reach the global minima much more quickly. There are a few more learning rate decay methods:

1. **Exponential decay:** $\alpha = (0.95)\text{epoch\_number} * \alpha_0$
2. $\alpha = k / \text{epochnumber}^{1/2} * \alpha_0$
3. $\alpha = k / t^{1/2} * \alpha_0$

- Here, t is the mini-batch number.

Learning rate decay

- While learning rate decay isn't always essential and should be a secondary consideration among hyperparameters, setting a well-tuned fixed value for Alpha can significantly impact training. Learning rate decay may aid training speed in certain scenarios.
- In neural network training, optimization algorithms strive to find a global minimum but may encounter challenges like local optima or saddle points. Recognizing these optimization issues enhances intuition about the training process.

FPT Education

FPT UNIVERSITY

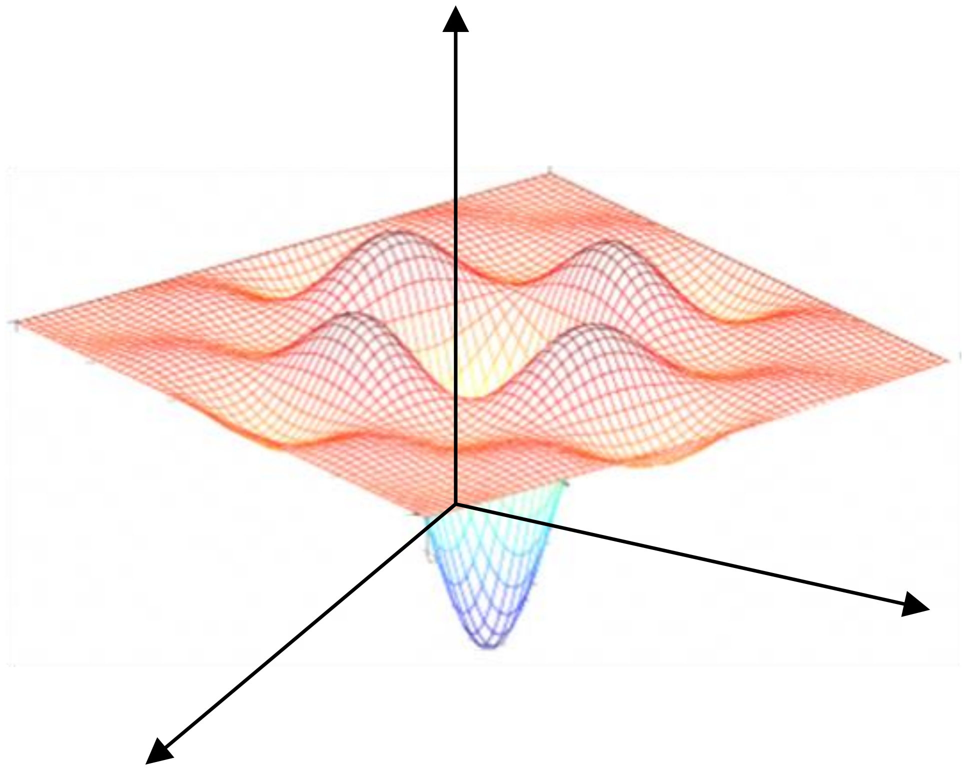
Optimization Algorithms

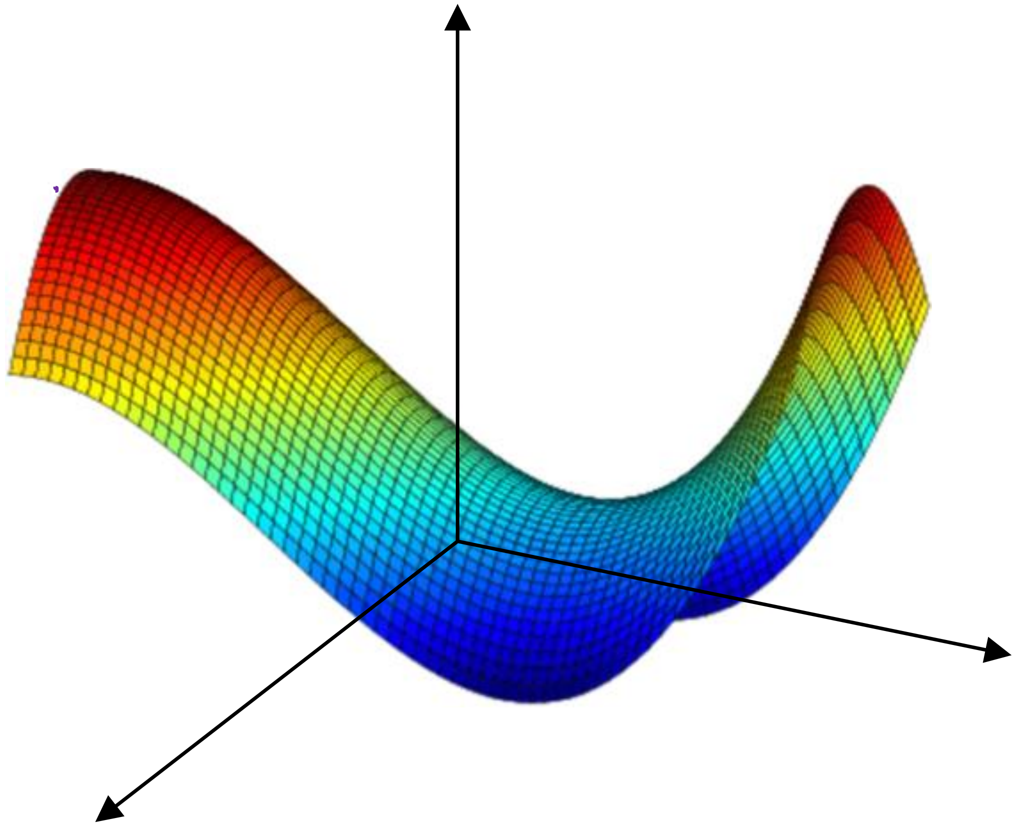
The problem of local optima

The problem of local optima

- In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in bad local optima. However, as the theory of deep learning has advanced, our understanding of local optima has also changed.
- It turns out that most points of zero gradient in a cost function are not local optima but saddle points. In a very high-dimensional space, for a point to be a local optima, all directions must look like a convex light function or a concave light function, which is very unlikely.

Local optima in neural networks

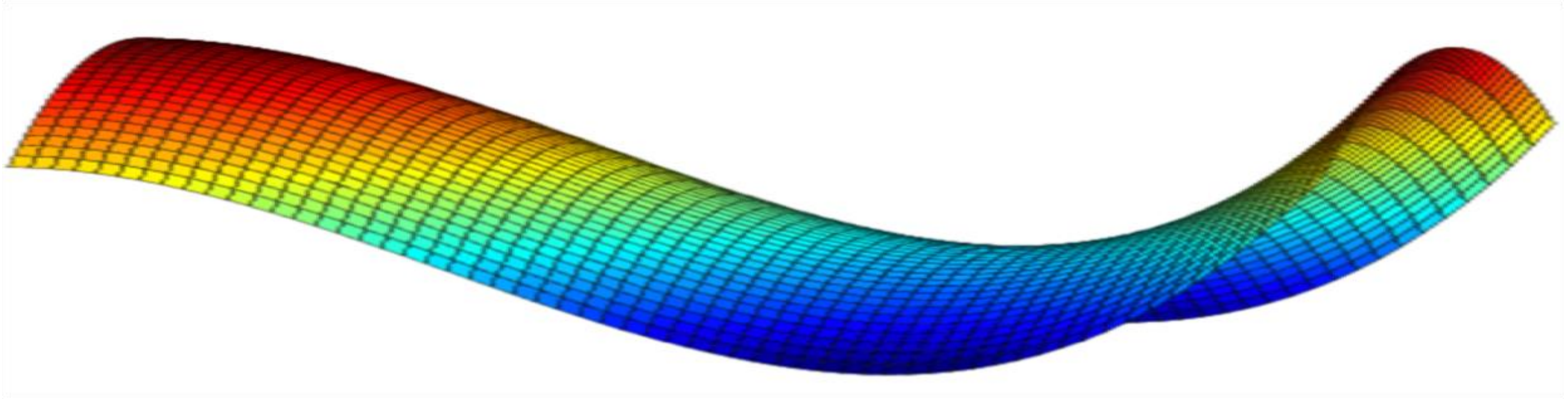




The problem of local optima

- Saddle points, where the gradient is zero in various directions, and plateaus, where the derivative remains near zero, are common challenges in optimization. These hinder learning, especially on plateaus, causing slow convergence.
- Algorithms like momentum, RMSProp, or Adam are beneficial in such scenarios. Due to the high-dimensional nature of neural network optimization, traditional intuition from low-dimensional spaces doesn't directly apply, and our understanding is still evolving.

Problem of plateaus



- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Summarization

- Mini-batch gradient descent divides training data into mini-batches for efficient updates.
- Exponentially weighted averages smooth noisy data by assigning more weight to recent observations.
- Bias correction improves the accuracy of early moving averages.
- Gradient descent with momentum adds a fraction of the previous update to the current gradient, aiding optimization in flatter regions.
- RMSprop adjusts learning rates for each parameter to handle varying gradients.
- The Adam optimization algorithm combines RMSprop and momentum for efficient and adaptive updates.
- Learning rate decay reduces the learning rate over time for refined updates.
- Local optima, suboptimal solutions during optimization, are addressed by modern techniques like stochastic gradient descent.
- Understanding these topics is crucial for effective neural network training and optimization.

Questions

1. Statement about mini-batch gradient descent?
2. Why mini-batch size not 1 or m?
3. How does the cost function plot differ between mini-batch and batch gradient descent?
4. β=0.9 EWMA. β change effect (right shift/oscillation)?
5. Which techniques help if batch GD is too slow?
6. What is Adam?