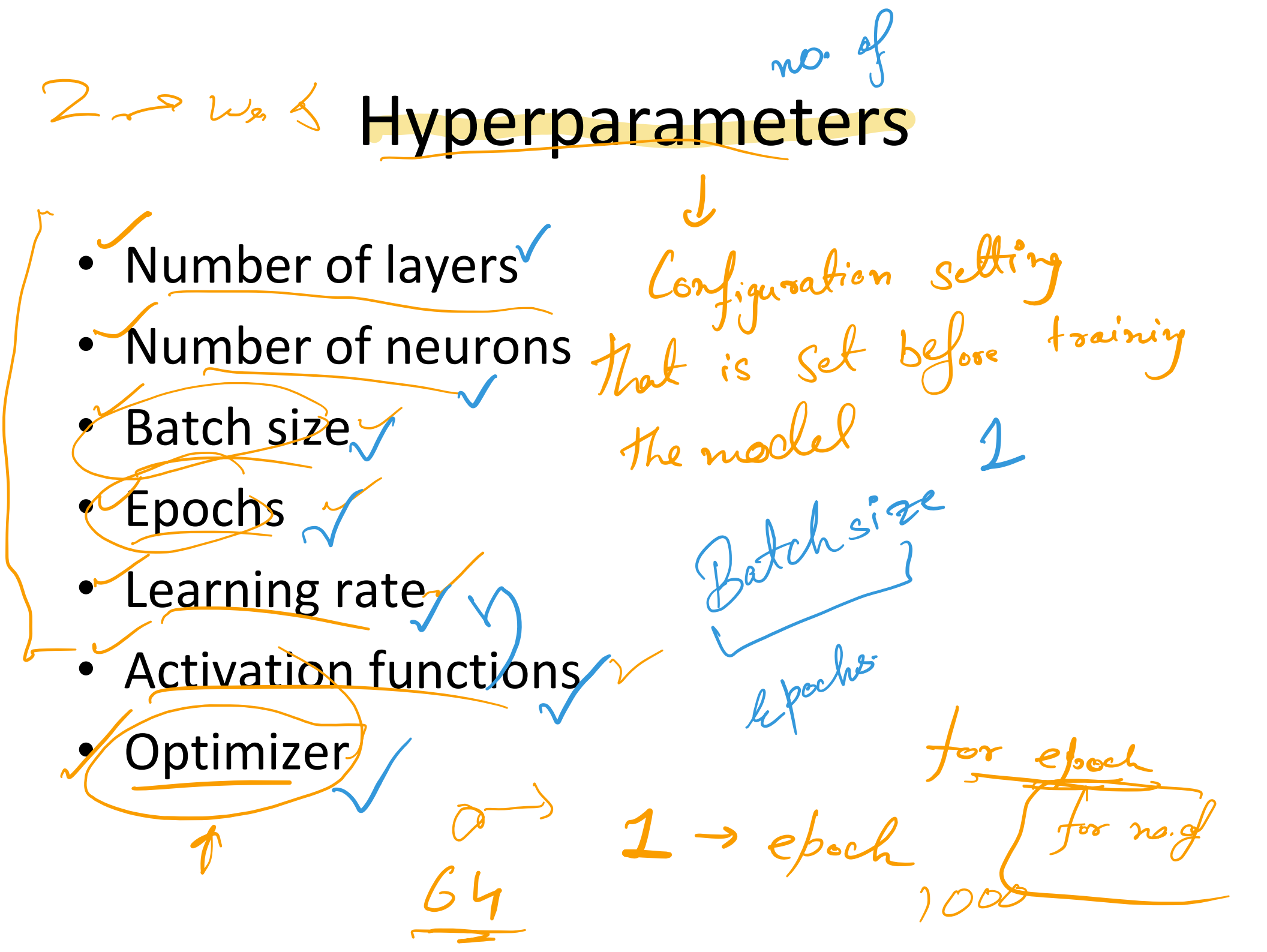
Parameters, Hyperparameters, Optimization

Hyperparameters

- Number of layers
- Number of neurons
- Batch size
- Epochs
- Learning rate
- Activation functions
- Optimizer



- Learning rate
- Momentum

Step decay

$$v_t = \gamma v_{t-1} + \eta \nabla_w L$$

$$w_{t+1} = w_t - \eta \nabla_w L$$

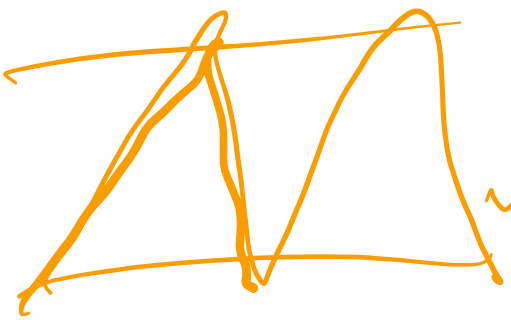
$$\eta = \frac{\eta_0}{1 + kt}$$

t: iteration, k: decay

$$\eta = \eta_0 e^{-kt}$$

- RMS

$$S w_t = \beta S w_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w}\right]^2$$



$$w_t = w_{t-1} + \frac{\eta}{\sqrt{S w_t + \epsilon}} \frac{\partial L}{\partial w}$$

Batch, Epoch, Iteration

- A training dataset can be divided into one or more batches
- **Batch size** - hyperparameter that defines number of samples to work through before updating internal model parameters
  - At end of batch, predictions are compared to expected output variables and error is calculated
  - From this error, the update algorithm is used to improve model, e.g. move down along error gradient

Batch, Epoch, Iteration

- Batch size - number of samples processed before model is updated
  - 1≤ Batch size ≤ number of samples in training dataset
- **If dataset does not divide evenly by batch size?**
  - Final batch has fewer samples than other batches
  - Alternately, remove some samples from dataset / change batch size such that number of samples divide evenly by batch size

Batch, Epoch, Iteration

- Small batch size:
  - Unlikely that a small sample is a good representation of the entire dataset
  - Will introduce a high degree of variance (noisiness) within each batch
- Large batch size:
  - May not fit in memory
  - Will have the tendency to overfit data
- A common heuristic – use square root of size of dataset

ex. 1000 images batch size=1

Epoch 1: iteration 1: Takes 1st image forward & backward propagation

:

1000 iteration

then epoch 2:

$y = \tanh x$

$y' = 1 - y^2$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial f(x)} \times \frac{\partial f(x)}{\partial z} \times \frac{\partial z}{\partial t} \times \frac{\partial t}{\partial w}$$

Batch, Epoch, Iteration

- **Epoch** - hyperparameter that defines number times learning algorithm will work through entire training dataset
  - Comprises of one or more batches
  - One epoch: each sample in training dataset has had a chance to update internal model parameters
  - A *for*-loop over number of epochs - each loop proceeds over complete training dataset
  - Within this *for*-loop: another nested *for*-loop that iterates over each batch of samples, of specified "batch size"
  - Number of **iterations** is equivalent to number of batches needed to complete one epoch

Batch, Epoch, Iteration

- Number of epochs ~ hundreds or thousands
  - Allows learning algorithm to run until error from model is sufficiently minimized
  - Common to create line plots (sometimes called **learning curves)** with error against epochs
  - Help to diagnose whether model has over learned, under learned, or is suitably fit to training dataset
  - $1 \leq$ No. of epochs $< \infty$
  - Can stop it using other criteria besides a fixed number of epochs, such as a change (or lack of change) in model error over time

Batch, Epoch, Iteration

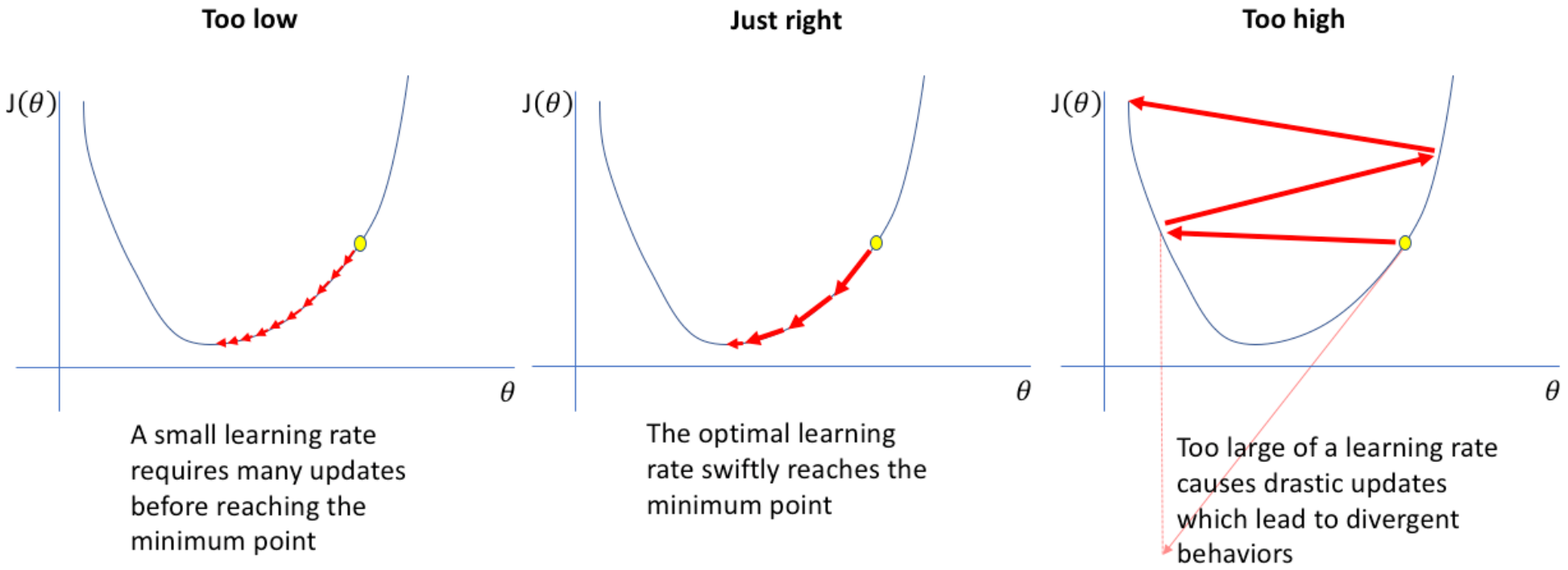
- Assume dataset with 200 samples
  - Ex. Batch size = 5, Epochs = 1,000
  - Dataset will be divided into 40 batches, each with five samples
  - Model weights will be updated after each batch of five samples
  - One epoch will involve 40 batches or 40 updates to model (40 iterations)
  - 1,000 epochs: model will be passed through whole dataset 1,000 times
    - Total of 40,000 batches during entire training process

Learning Rate

- Learning rate (step size)
  - Configurable hyperparameter
  - Controls how much to change model in response to estimated error when model weights are updated
  - Small positive value, often in range between 0.0 and 1.0
- Large learning rate
  - May cause model to learn a sub-optimal set of weights too fast or an unstable training process
- Small learning rate
  - Requires more training epochs
  - May result in a long training process

Learning Rate

- How to set the learning rate, η?
  - Small η converges slowly; gets stuck in local minima
  - Large η overshoots, becomes unstable and diverges
  - Stable η converges smoothly and avoids local minima



https://www.jeremyjordan.me/nn-learning-rate/

Learning Rate

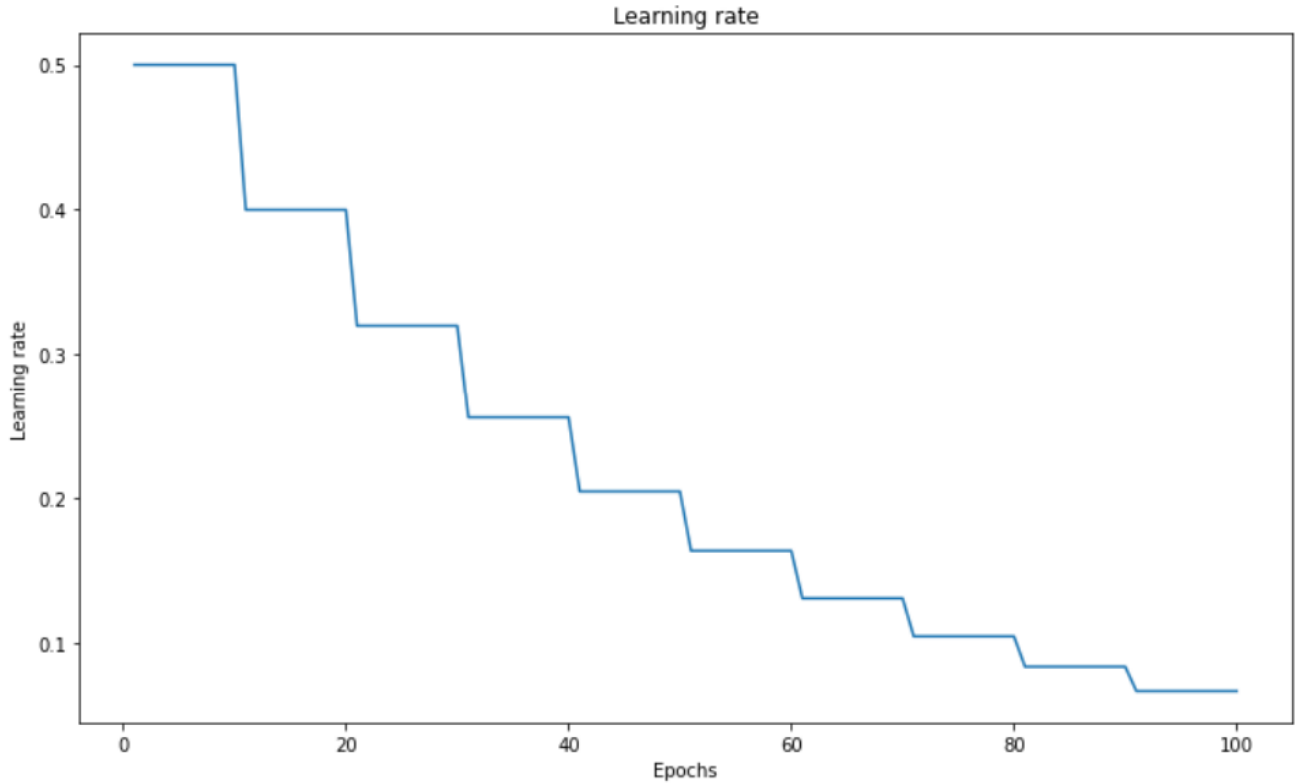
- How to set η?
  - Trial and error: Try different rates and see what works right
  - Learning rate decay
  - Cyclical learning rate: Rate varies between two bounding values
  - Adaptive learning: Rate adapts to the landscape
    - Learning rates are not fixed
    - Can be made larger or smaller depending on:
      - How large is the gradient
      - How fast is the learning
      - Size of particular weights …

Learning Rate Schedule

- A predefined framework that adjusts learning rate between epochs or iterations as training progresses
- Two common techniques:
  - Constant learning rate: Initialize a learning rate and do not change it during training
  - Learning rate decay (**learning rate annealing)**: Select an initial learning rate, then gradually reduce it in accordance with a scheduler
    - Early in training, learning rate is set to be large in order to reach a set of weights that are good enough
    - Over time, weights are fine-tuned to reach higher accuracy by leveraging a small learning rate

Step Decay Learning Rate Schedule

- Most popular form of learning rate annealing:
  - *Step decay* - learning rate is reduced by some percentage after a set number of training epochs



https://neptune.ai/blog/how-to-choose-a-learning-rate-scheduler

Step Decay Learning Rate Schedule

- $\eta = \eta_0/(1+ \text{decay\_rate}*\text{epoch\_number})$
- Ex. η_0 = initial learning rate = 0.2, decay_rate = 1

Epoch	η
1	0.1
2	0.067
3	0.05
4	0.04

$$\eta = \frac{\eta_0}{1 + \text{decay-rate} * \text{epoch no.}}$$

X[1]	X[2]	X[3]	…	X[m]

Epoch 1

Epoch 2

…….

Epoch n

Decay Learning Rate

- Time-based decay:

  $\eta = \eta/(1 + k*t)$

  k: decay rate

  t: iteration_number

- Exonential decay:

  $\eta = \eta_0 * e^{-k*t}$

https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

Cyclical Learning Rate (CLR)

- Intuition of CLR - Let learning rate vary within a range of values rather than adopting linearly/exponentially decreasing value
  - Set a definite range of learning rates
  - Instead of linear or exponential variation - cyclically vary learning rates from defined range
  - Leslie* considered following function forms:
    - Triangular window (linear) – preferred due to simplicity
    - Welch window (parabolic)
    - Hann window (sinusoidal)

Smith, Leslie. (2017). Cyclical Learning Rates for Training Neural Networks. 464-472. 10.1109/WACV.2017.58.

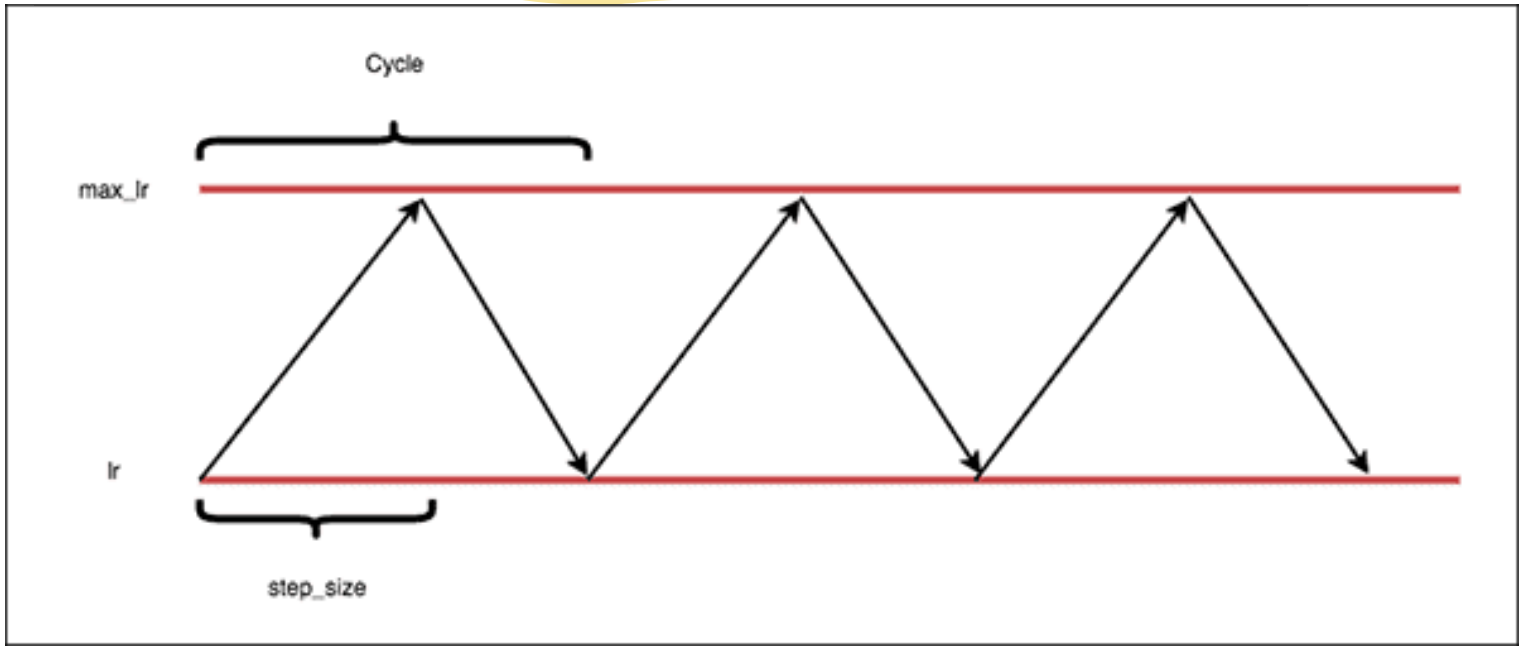
Cyclical Learning Rate (CLR)

- Instead of monotonically decreasing learning rate:
  - Define lower bound on learning rate, called '*base_lr*'
  - Define upper bound on learning rate, called '*max_lr*'
  - Allow learning rate to oscillate back and forth between these two bounds when training
    - Slowly increasing and decreasing the learning rate *after every batch update*
  - *Step_size:* In how many epochs, learning rate will reach from one bound to another

- Cyclical learning rate schedule - varies between two bound values
  - Main learning rate schedule is a triangular update rule
  - Can also use of a triangular update in conjunction with a fixed cyclic decay or an exponential cyclic decay

Why it works?

  - Periodic higher learning rate within each epoch helps to come out of any saddle point or local minima

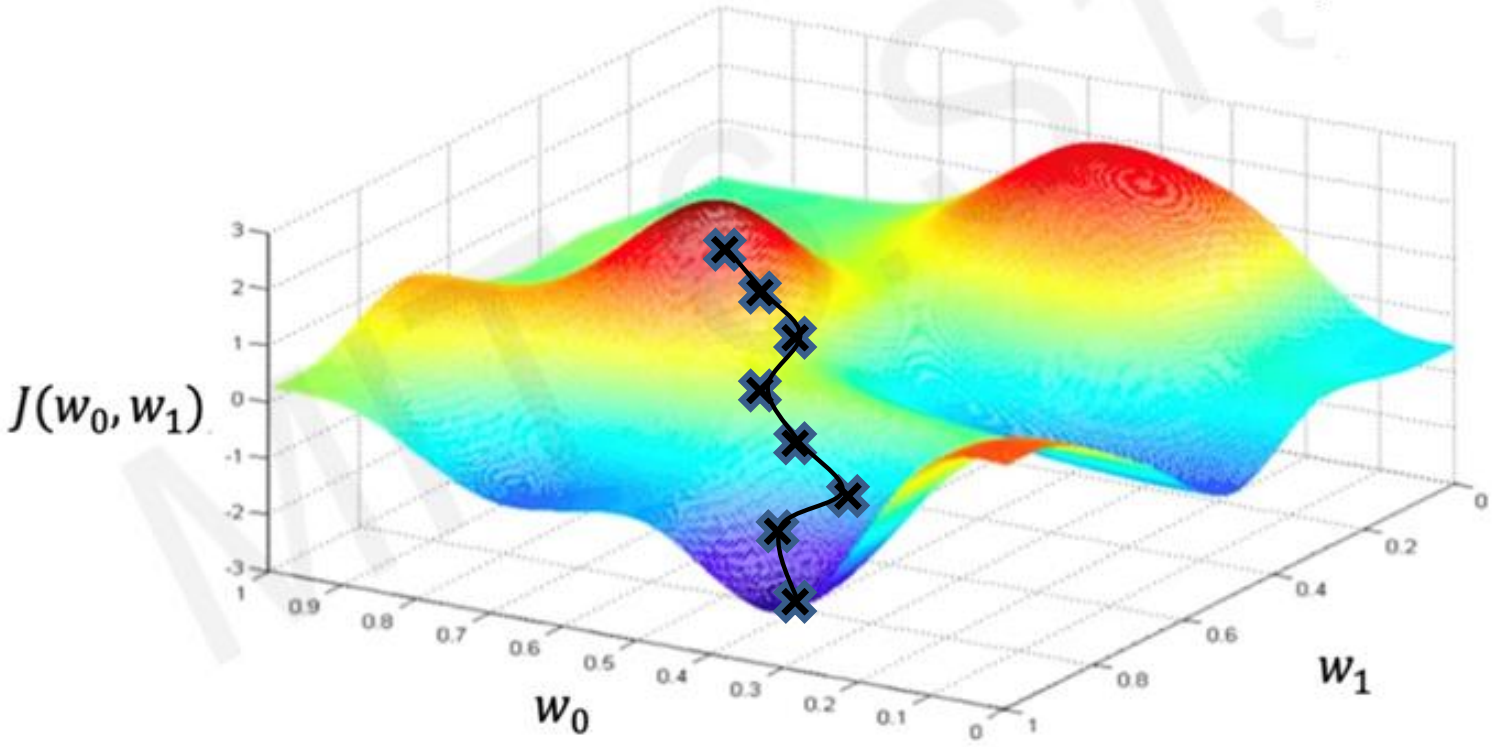


Adaptive Learning Rate

- Challenge of using learning rate schedules:
  - Hyperparameters have to be defined in advance
  - Depend heavily on type of model and problem
  - Same learning rate is applied to all parameter updates
- Adaptive gradient descent algorithms: ex. Adagrad, Adadelta, RMSprop, Adam
  - Per-parameter learning rate methods
  - Provide heuristic approach without requiring expensive work in tuning hyperparameters for the learning rate schedule manually

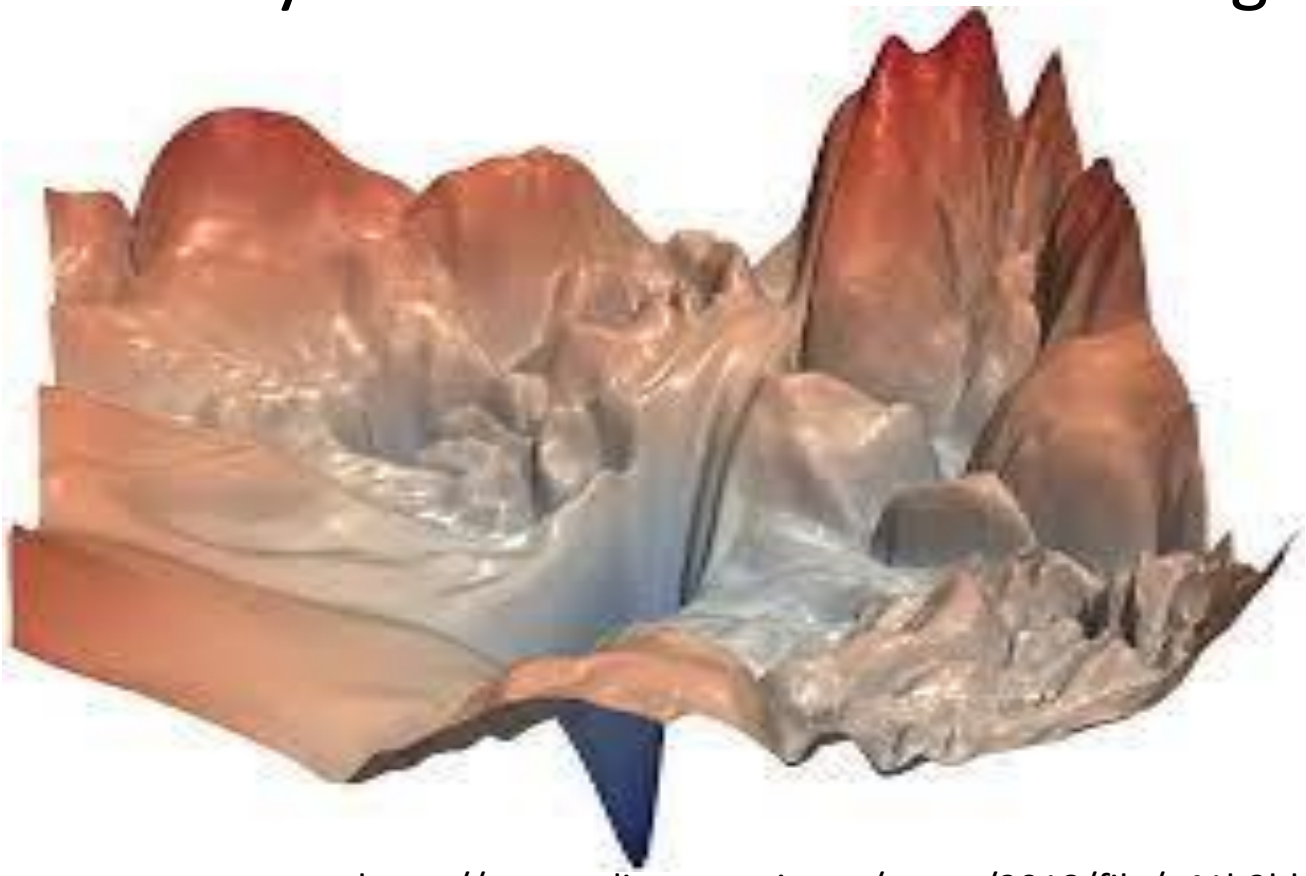
Loss Optimization

Repeat until convergence → Gradient Descent



Training Neural Networks

- Extremely difficult and complex
- Computationally intensive
- Many local minima – how to find global minima?



Visualizing the loss landscape

Non-convex problem: Has a global minima and local minima

https://proceedings.neurips.cc/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf

Batch Gradient Descent

- Batch Size = Size of Training Set
- Batch gradient descent computes gradient of cost function w.r.t. to parameters (*w,b*) for entire training dataset
  - Computationally intensive
  - Summation over all data points in dataset in each iteration
  - Can be very slow
  - Difficult to deal with datasets that do not fit in memory
  - Converges to global minimum for convex error surfaces and to a local minimum for non-convex surfaces

Stochastic Gradient Descent (SGD)

- Batch Size = 1
- Performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$
- Algorithm:
  1. Initialize weights randomly
  2. Loop until convergence:
     1. Pick single data point ***i***
     2. Compute gradient $\partial J(\boldsymbol{W})/\partial \boldsymbol{W}$
     3. Update weights $\boldsymbol{W} := \boldsymbol{W} - \eta * \partial J(\boldsymbol{W})/\partial \boldsymbol{W}$
  3. Return weights

Stochastic Gradient Descent

- Easy to compute and much faster
- Performs frequent updates with a high variance that cause the objective function to fluctuate heavily
  - Batch gradient descent converges to minimum of basin the parameters are placed in
  - SGD's fluctuation enables it to jump to new and potentially better local minima
    - Complicates convergence as keeps overshooting
    - However, if learning rate is slowly decreased: SGD converges to a local or global minimum for non-convex/convex optimization respectively
  - Shuffle training data at every epoch

Mini Batch Gradient Descent

- 1 < Batch Size < Size of Training Set
  - Popular batch sizes include 32, 64, and 128 samples
- Performs an update for every mini-batch of *n* training examples
- Algorithm:
  1. Initialize weights randomly
  2. Loop until convergence:
     1. Pick batch of ***n*** data points
     2. Compute gradient $\partial J(W)/\partial W = (1/n) * \sum \partial J_k(W)/\partial W$ $k = 1 \ldots n$
     3. Update weights $W := W - \eta * \partial J(W)/\partial W$
  3. Return weights

Mini Batches Gradient Descent

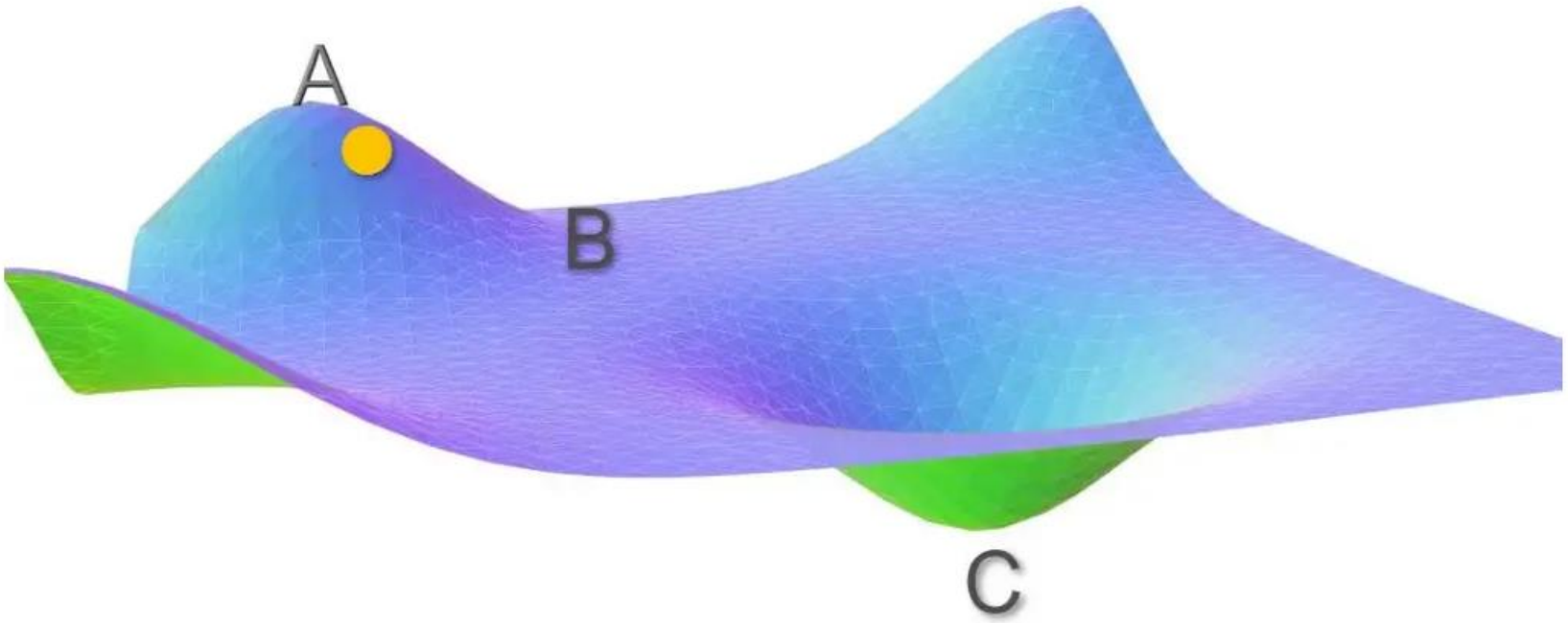
- *n*: 30~300 of samples (32/64/128)
- Fast to compute
- Reduces variance of parameter updates
  - Smoother convergence
- Better estimate of true gradient
- Leads to faster training
  - Can parallelize computation

Gradient Descent - Problems

- Gradient descent algorithm - progression of search can bounce around non-convex search space based on gradient
  - Ex., search may progress downhill towards minima
  - But during progression, it may move in another direction
  - Even uphill, depending on gradient of specific points (sets of parameters) encountered during search
- Can slow down progress of search

Gradient Descent

- Gentler the slope, smaller the gradient
  - Smaller the gradient, slower the movement
- Larger the slope, higher the gradient
  - Larger the gradient, higher the movement
- Takes a lot of time to navigate regions with gentler slopes


A
B
C

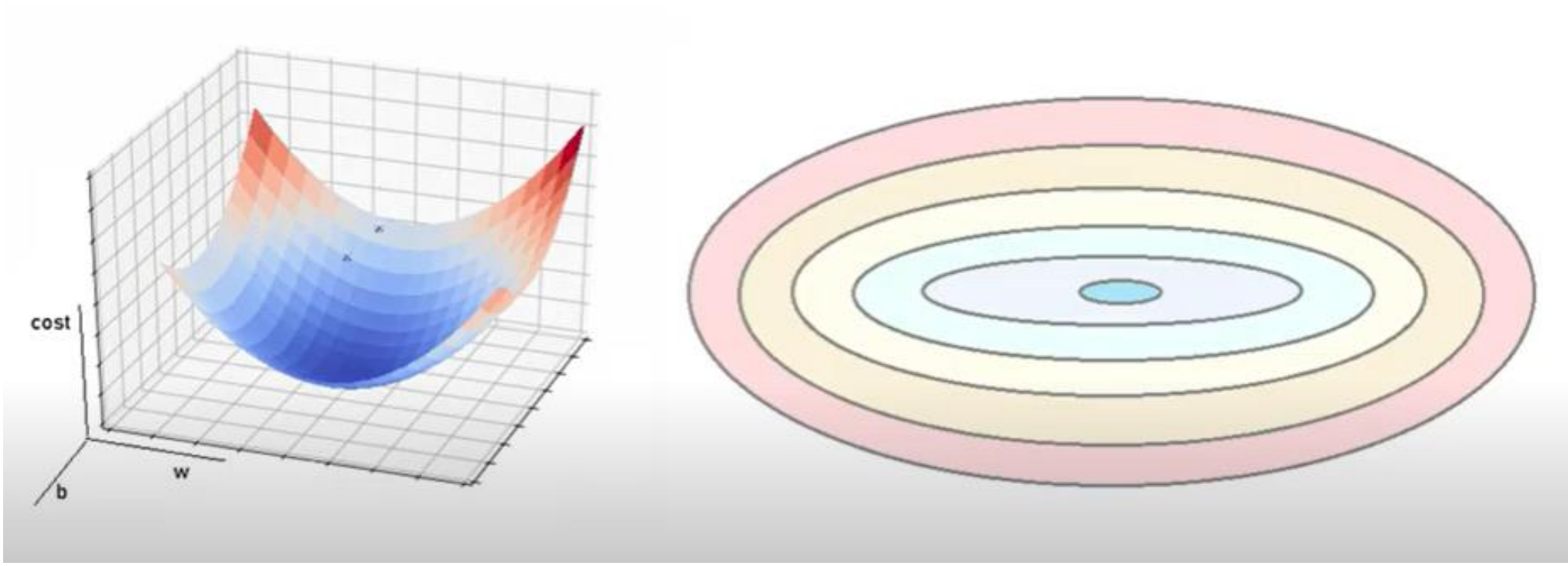
OPTIMIZERS

Momentum

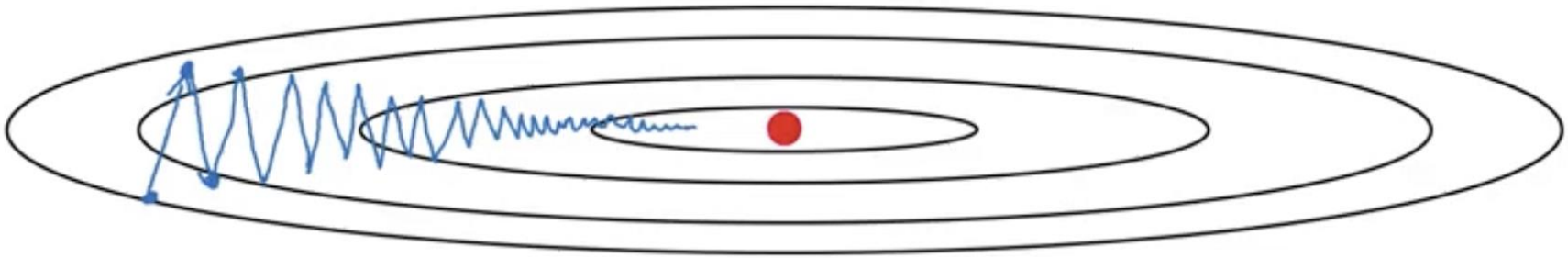
- **Momentum**: extension to gradient descent optimization algorithm
  - Often referred to as **gradient descent with momentum**
  - Most useful in optimization problems where objective function has a large amount of curvature (e.g. changes a lot)
    - Gradient may change a lot over relatively small regions of search space
  - Helpful when search space is flat or nearly flat, e.g. zero gradient
    - Allows search to progress in same direction as before the flat spot and helpfully cross the flat region

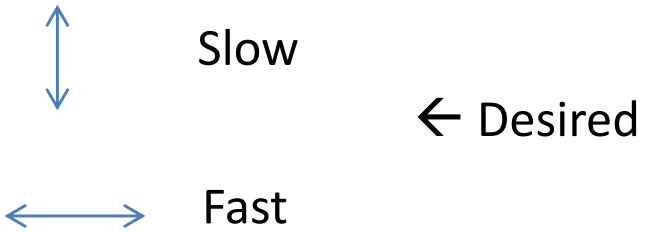
Polyak, Boris. (1964). Some methods of speeding up the convergence of iteration methods. Ussr Computational Mathematics and Mathematical Physics. 4. 1-17. 10.1016/0041-5553(64)90137-5.

Contour Plot

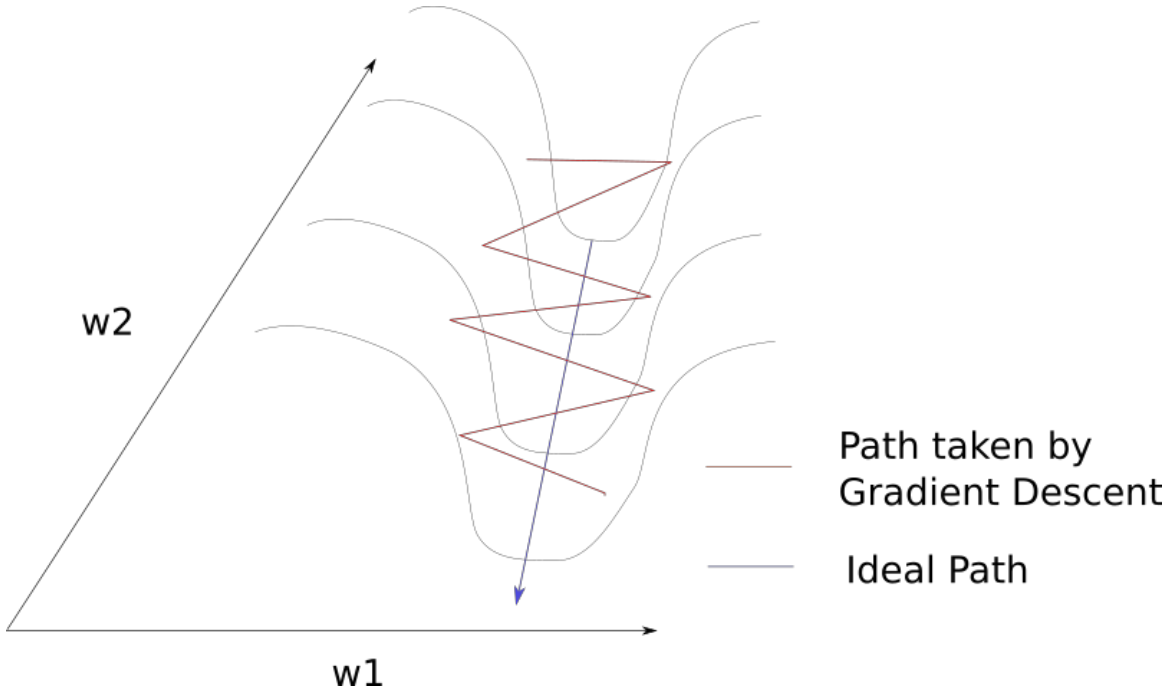


Momentum





Momentum



- Gradient descent is bouncing along ridges of the ravine and moving a lot slower towards minima
- This is because surface at the ridge curves much more steeply in the direction of *w1*

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

Momentum

- Can use a slow learning rate to deal with this bouncing between the ridges problem
  - Makes sense to slow down when nearing a minima, and we want to converge into it
  - If we use a slower learning rate, it might take so too much time to get to minima
  - Might lead user to believe that loss is not improving at all, and abandon training
- **Momentum -** Instead of using only gradient of current step to guide search
  - Also accumulate gradient of past steps to determine direction to go

Gradient descent

- Gradient descent update equation:
  - Calculation of change to position
  - Update of old position to new position
- Change in parameters is calculated as gradient for the point scaled by step size

$$change = \eta * \nabla(w)$$

$$change_t = \gamma\, change_{t-1} + \eta \nabla w$$

- New position:

$$w = w - change$$

Gradient descent with Momentum

- Involves maintaining change in parameter and using it in subsequent calculation of change in position
- Update at current iteration or time (t) will add change used at previous time (t-1) weighted by **momentum hyperparameter**, as follows:

$$change_t = \gamma * change_{t-1} + \eta * \nabla(w_t)$$

Update to position performed as before:

$$w_{t+1} = w_t - change_t$$

Gradient descent with Momentum

$$change_t = \gamma * change_{t-1} + \eta * \nabla(w_t)$$

$$w_{t+1} = w_t - change_t$$

$change_0 = 0$

$change_1 = \gamma * change_0 + \eta \nabla(w_1) = \eta \nabla(w_1)$

$change_2 = \gamma * change_1 + \eta * \nabla(w_2) = \gamma \eta \nabla(w_1) + \eta \nabla(w_2)$

$change_3 = \gamma * change_2 + \eta * \nabla(w_3) = \gamma [\gamma \eta \nabla(w_1) + \eta \nabla(w_2)] + \eta \nabla(w_3)$

$= \gamma^2 \eta \nabla(w_1) + \gamma \eta \nabla(w_2) + \eta \nabla(w_3)$

.....

$change_t = \gamma^{t-1} \eta \nabla(w_1) + \gamma^{t-2} \eta \nabla(w_2) + ... + \eta \nabla(w_t)$

Exponentially weighted averages

$\theta_1 = 3° C$

$\theta_2 = 5° C$

$\theta_3 = 7° C$

$\theta_4 = 2° C$

....

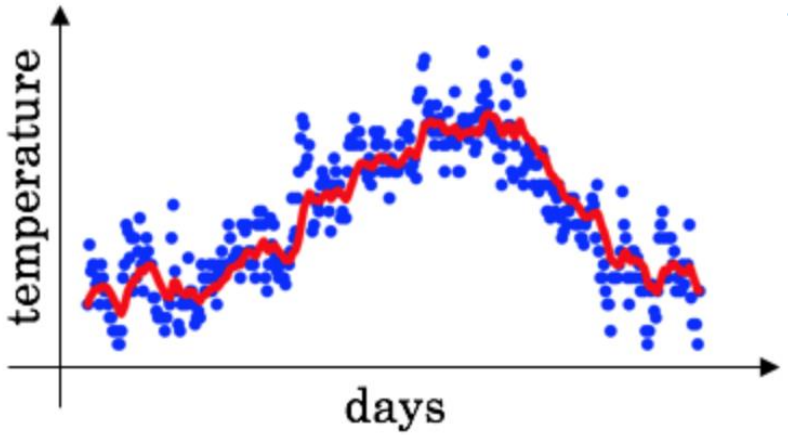
$\theta_{198} = 30° C$

$\theta_{199} = 37° C$

$\theta_{200} = 28° C$

....

....



$v_0 = 0$

$v_1 = 0.9*v_0 + 0.1*\theta_1$

$v_2 = 0.9*v_1 + 0.1*\theta_2$

$v_3 = 0.9*v_2 + 0.1*\theta_3$

.....

$v_t = 0.9*v_{t-1} + 0.1*\theta_t$

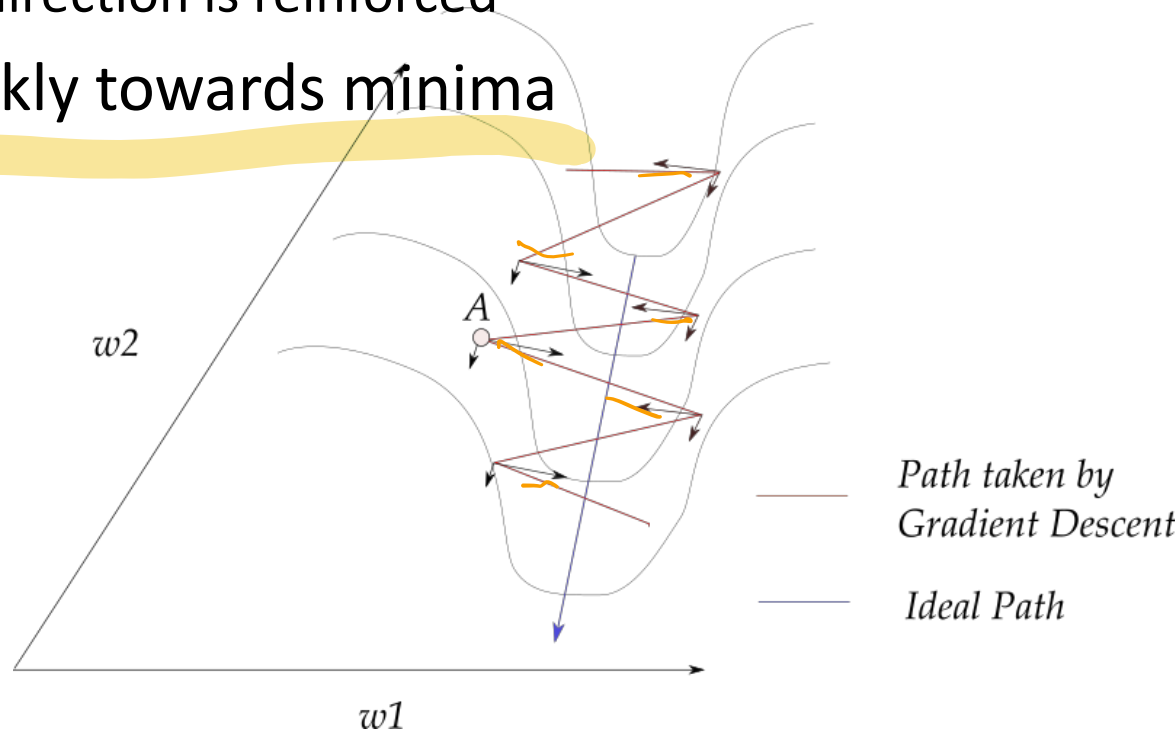
v_t approximately averages over (1/(1-β) days of temperature

If β = 0.9 → average over ~10 days

If β is large, curve becomes more smoother as averaging is over more number of days

Momentum

- Most gradient updates are in *zig-zag* direction
  - Each gradient update resolved into components along *w1* and *w2* directions
  - If these vectors are individually summed up:
    - Components along direction *w1* cancel out
    - Component along *w2* direction is reinforced
  - Helps move more quickly towards minima



Momentum

$\gamma \, change_{t-1} + \eta \Delta w$

- Momentum based gradient descent able to take large steps because momentum carries it along
  - **Will there be a situation when momentum causes to run past our goal?**
- Momentum adds an additional hyperparameter
  - Controls amount of history (momentum) to include in the update equation
  - Value for hyperparameter: range 0.0 to 1.0
  - Often value close to 1.0, such as 0.8, 0.9, or 0.99
  - Large momentum (e.g. 0.9) means update is strongly influenced by previous update

RMSProp

- Root Mean Square Propagation
- Algorithm for **adaptive learning rate**
  - Decay in learning rate as gradient changes over iterations
- Adjusts learning rate automatically
  - Chooses **a different learning rate for each parameter**
  - Update is done separately for each parameter

https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

RMSProp

$sw_t = \beta_2 sw_{t-1} + (1-\beta_2) * (\nabla w_t)^2$

- Take s_t : exponential average of squares of gradients
- For each parameter:

$$sw_t = \beta_2 sw_{t-1} + (1 - \beta_2) * [\partial L/\partial w_t]^2$$

$$w_{t+1} = w_t - \eta/(sw_t + \varepsilon)^{1/2} * \partial L/\partial w_t$$

$$sb_t = \beta_2 sb_{t-1} + (1 - \beta_2) * [\partial L/\partial b_t]^2$$

$$b_{t+1} = b_t - \eta/(sb_t + \varepsilon)^{1/2} * \partial L/\partial b_t$$

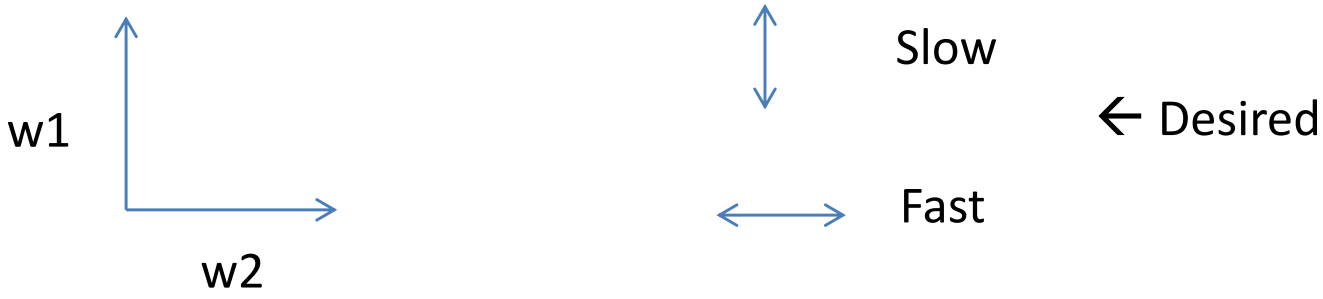
RMSProp

s initialised to 0

Common values:

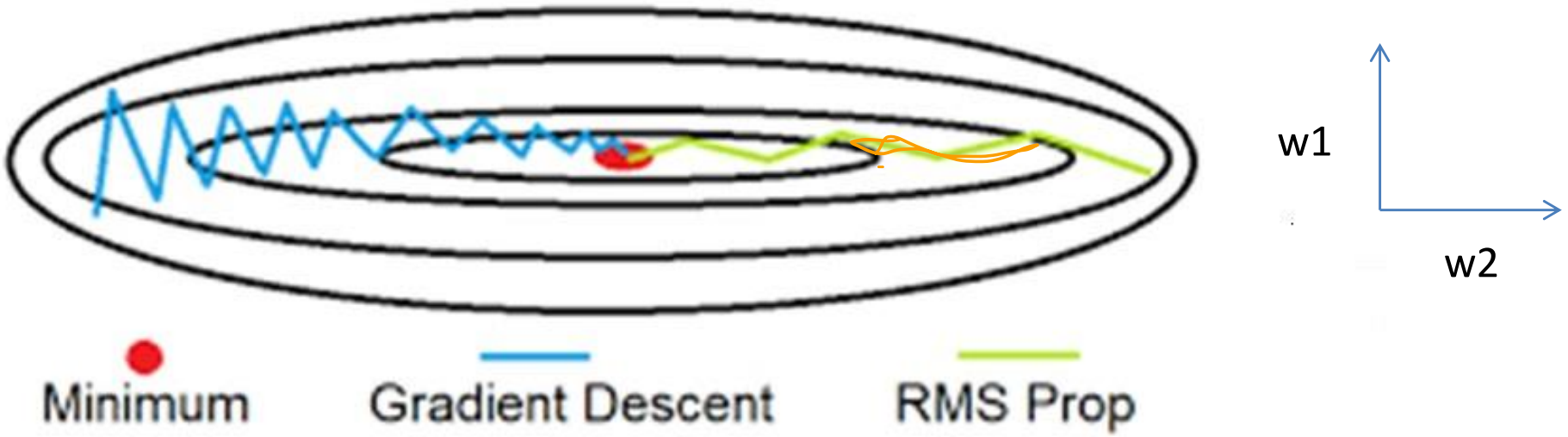
$\eta = 0.001$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$

(ε to ensure that we do not end up dividing by zero)



If $\partial L/\partial w_t$ for w2 is small, so s_t will be small and hence dividing η by a relatively small number will increase learning rate. For w1, if $\partial L/\partial w_t$ is large, so s_t will be large. w1 should be divided by a large number to slow down learning rate

RMSProp



In a high-dimensional space, w1 could represent a set of parameters and w2 could represent another set of parameters

https://blog.jaysinha.me/train-your-first-optimized-machine-learning-model/

RMSProp

$$\frac{\eta}{\sqrt{s_{w_t} + \epsilon}}$$

- Done separately for each parameter:
  - Gradient corresponds to projection, or component of gradient along direction of parameter being updated
- Why exponential average?
  - Helps weigh more recent gradient updates more than less recent ones
  - "exponential" - weightage of previous terms falls exponentially

$$s_{w_t} = \beta_2 w_{t-1} + (1-\beta_2)(\nabla w)^2$$

Adam

adaptive moment optimization Algo.

- Adam - Adaptive Moment Optimization algorithms
  - Combines heuristics of both Momentum and RMSProp
  - Momentum accelerates search in direction of minima (smoothening)
  - RMSProp impedes search in direction of oscillations (change learning rate)

Diederik P. Kingma and Jimmy Ba, “Adam: A Method for Stochastic Optimization, 2017, arXiv

Adam

- For each parameter w:

  v_t: Exponent average of gradients along w

  s_t: Exponent average of squares of gradients along w

  β_1 β_2 hyperparameters

At iteration t, computed for each mini-batch:

$v_t = \beta_1 v_{t-1} - (1 - \beta_1) * \partial L/\partial w_t$ ← momemtum

$s_t = \beta_2 s_{t-1} - (1 - \beta_2) * [\partial L/\partial w_t]^2$ ← RMSprop

$\Delta w_t = \eta * v_t/(s_t + \varepsilon)^{1/2} * \partial L/\partial w_t$

$w_{t+1} = w_t - \Delta w_t$

s and v initialized to 0

Commonly used values: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$

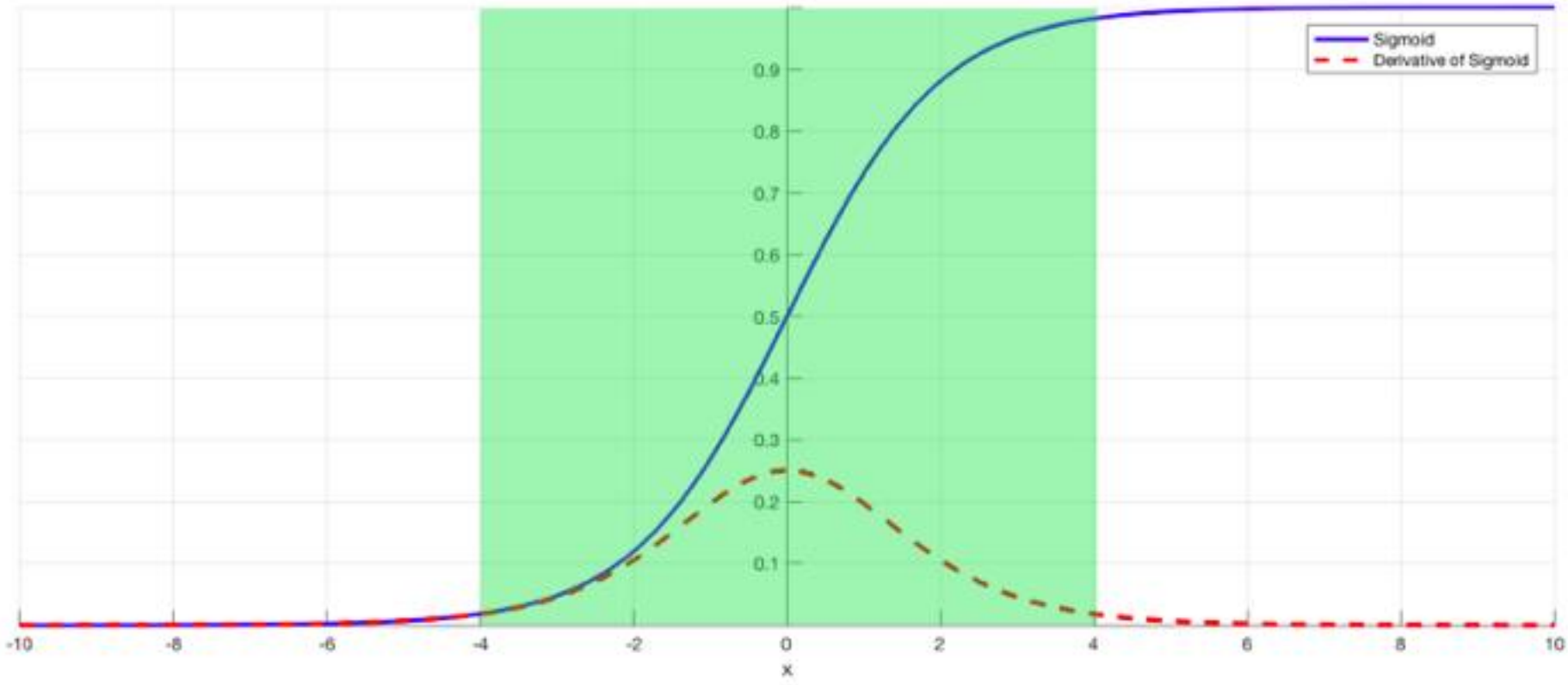
Adam

- Compute exponential average of gradient as well as squares of gradient for each parameter
- To decide learning step – multiply learning rate by average of gradient and divide it by root mean square of exponential average of square of gradients
  - First moment normalized by second moment gives direction of the update
- Add the update

Vanishing Gradient

- In networks using certain activation functions, adding more layers may lead to:
  - Gradients of loss function approaching zero
- Why?:
  - Certain activation functions (eg. sigmoid) squeeze a large input space into a small input space between 0 and 1
  - A large change in input of sigmoid function will cause a small change in output - derivative becomes small
  - For shallow network, not a big problem
  - With more layers - cause more activation function with them, can cause gradient to be too small

Vanishing Gradient



Vanishing Gradient

- Gradients found using backpropagation
  - Finds derivatives of network by moving layer by layer from final layer to initial one
  - By chain rule, derivatives of each layer multiplied down the network to compute derivatives of initial layers
- When *n* hidden layers use an activation like sigmoid
  - *n* small derivatives are multiplied together
  - Gradient decreases exponentially as they propagate down to initial layers
  - Networks are unable to backpropagate gradient information to input layers of model
  - Difficult to train network

Vanishing Gradient

- Small gradient means:
  - Weights and biases of initial layers will not be updated effectively with each training session
  - Since initial layers are crucial to recognize core elements of input data, it can lead to overall inaccuracy of whole network
- Simplest solution
  - Use other activation functions, such as ReLU – does not cause a small derivative
  - Other solutions – residual networks, batch normalization

Exploding Gradient

- In some cases, gradients keep on getting larger and larger as backpropagation algorithm progresses
  - Causes very large weight updates: causes gradient descent to diverge; known as ***exploding gradients*** problem
- Why?
  - Suppose initial weights assigned to network generate some large loss
  - Gradients can accumulate during an update and result in very large gradients
  - Eventually results in large updates to network weights and leads to an unstable network.
  - Parameters can sometimes become so large that they overflow and result in NaN values

Solutions: Exploding Gradient

- Re-design Network Model
  - Redesign network to have fewer layers
  - Can use smaller batch size while training
- Use Long Short-Term Memory Networks
- Use Gradient Clipping
  - Values of error gradient checked against a threshold value
  - Clipped/set to threshold value if error gradient exceeds threshold
- Weight Regularization
  - Check size of network weights and apply a penalty to loss function for large weight values

Gradient Clipping

- A technique that tackles exploding gradients
- Involves normalizing error gradient vector such that vector norm (magnitude) equals a defined value
- If the gradient **g** gets too large, rescale it to keep it small

  If $\|\mathbf{g}\| \geq c$, then, $\mathbf{g} \leftarrow\!\shortmid\ c \cdot \mathbf{g}/\|\mathbf{g}\|$
- c is a hyperparameter, $\|\mathbf{g}\|$ is norm of **g**
- Since $\mathbf{g}/\|\mathbf{g}\|$ is a unit vector, after rescaling, new **g** will have norm c

How to know?

Vanishing Gradient	Exploding Gradient
Parameters of higher layers change significantly whereas parameters of lower layers would not change much	Exponential growth in model parameters
Model weights may become 0 during training	Model weights may become NaN during training
Model learns very slowly and perhaps training stagnates at a very early stage just after a few iterations	Model experiences avalanche learning

Weight Initialization

- Weight initialization used to define initial values for parameters in neural network models prior to training models on a dataset
- Crucial to initialize weights appropriately to ensure a model with high accuracy
  - If weights not correctly initialized, may give rise to Vanishing Gradient or Exploding Gradient problem

Weight Initialization

- Historically, weight initialization involved using small random numbers
- Over last decade, more specific heuristics have been developed that use information, like activation function being used and number of inputs to node
- Rules of thumb:
  1. The *mean* of activations should be zero
  2. The *variance* of activations should stay same across every layer

Weight Initialization

- Zero initialization: all weights assigned zero as initial value
- Result - derivative w.r.t. loss function is same for every weight in W^l
  - All weights have same value in subsequent iterations
  - Makes hidden layers symmetric and this process continues for all *n* iterations
  - Highly ineffective as neurons learn same feature during each iteration
  - During any kind of constant initialization, same issue happens to occur
  - Thus, constant initializations are not preferred

Weight Initialization

- **Random initialization:** assigns random values except for zeros as weights to neuron paths
  - However, problems such as Overfitting, Vanishing Gradient Problem, Exploding Gradient Problem might occur
- **Random Normal:** Weights are initialized from values in a normal distribution
- **Random Uniform:** Weights are initialized from values in a uniform distribution

Weight Initialization

- **Glorot or Xavier initialization**: current standard approach for initialization of weights of nodes that use Sigmoid or TanH activation function
  - Named for Xavier Glorot, currently a research scientist at Google DeepMind
- Initialization method calculated as a random number with uniform probability distribution between range $(-\frac{1}{\sqrt{n}})$ and $(+\frac{1}{\sqrt{n}})$, *n* is no. of inputs to node
- Found to have problems when used to initialize networks that use ReLU activation function

Weight Initialization

- **He initialization**: current standard approach for initialization of weights of neural network layers and nodes that use ReLU activation function
  - Named for Kaiming He, currently a research scientist at Facebook
- Calculated as a random number with a Gaussian probability distribution with a mean of 0.0 and a standard deviation of $\sqrt{(\frac{2}{n})}$, where *n* is number of inputs to node

$$\text{weight} = G\,(0.0, \sqrt{(\frac{2}{n})})$$

Activation functions

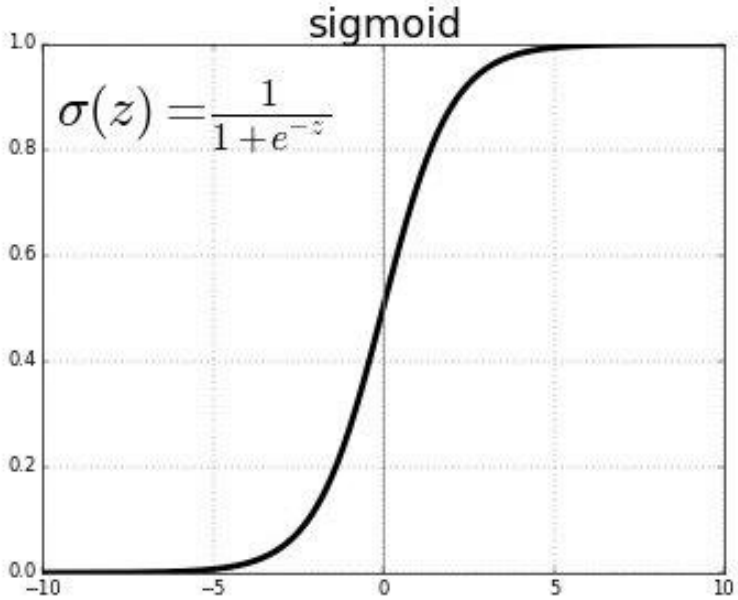
- Defines how weighted sum of input is transformed into output from nodes in a layer of network
- Also called:
  - Transfer function
  - Squashing function
- All hidden layers typically use same activation function
- Output layer will typically use a different activation function
  - Dependent upon type of prediction required by model

Activation functions

- Sigmoid
  - Output ranges from 0 – 1
  - Used in output layers where output is predicted as a probability
  - Used commonly in hidden layers in RNN

Cons:

  - Non-zero centered output
  - Slow convergence
  - Vanishing gradient problem



https://colab.research.google.com/drive/1munoUkJc-N-y_xvin_YSul0KDuaiYkNy#scrollTo=dEXZKP4EanSa

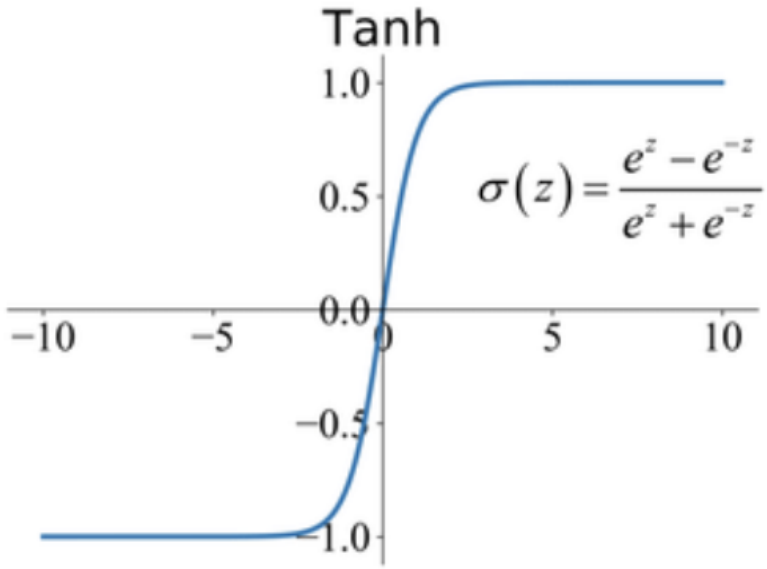
https://www.quora.com/What-is-a-sigmoid-function-in-neural-networks

Activation functions

- Tanh
  - Output ranges from -1 to +1; Zero centered
  - Easier to model inputs which have strong positive, negative, neutral values
  - Used commonly in hidden layers in RNN

Cons:

  - Vanishing gradient problem



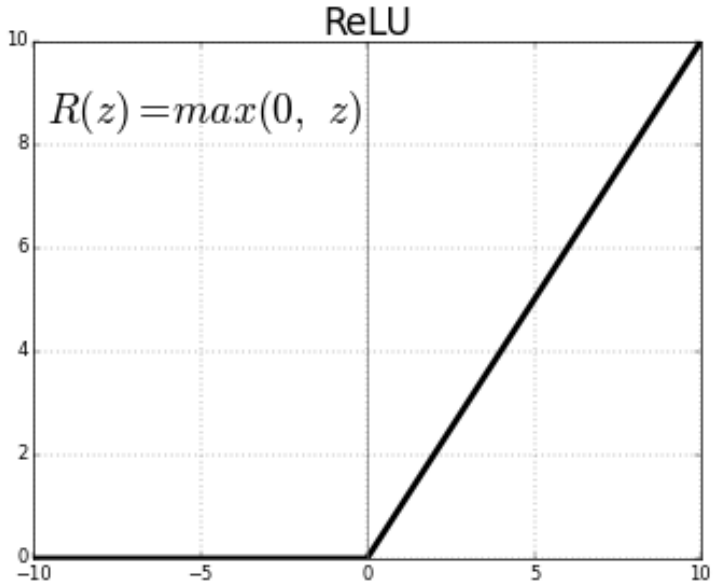
https://paperswithcode.com/method/tanh-activation

Activation functions

- ReLU
  - Most widely used
  - Provides better generalization
  - Faster computation
  - Less susceptible to vanishing gradient

Cons:

  - Easily overfits (techniques like dropout have to be used to avoid this)
  - Results in saturated or dead units



https://medium.com/

Activation functions

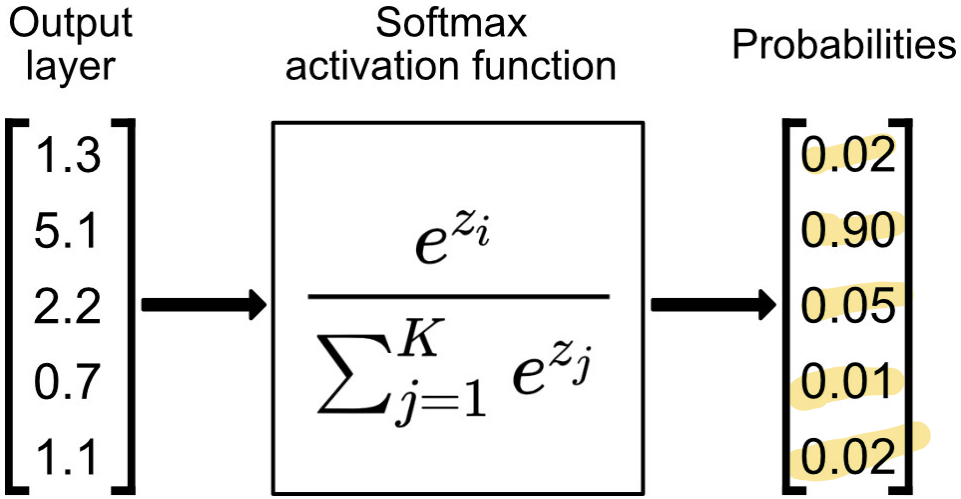
- Use of ReLU to address vanishing gradient
  - Has only two gradients 0 or 1
  - Gradient one when output of function is > 0
  - Gradient zero when output of function is < 0
  - These derivatives give either 0 or either 1 when multiplying together
  - Backpropagation equation will have only two options:
    - Either being 1 or being 0
    - Update is either nothing or takes contributions entirely from other weights and biases

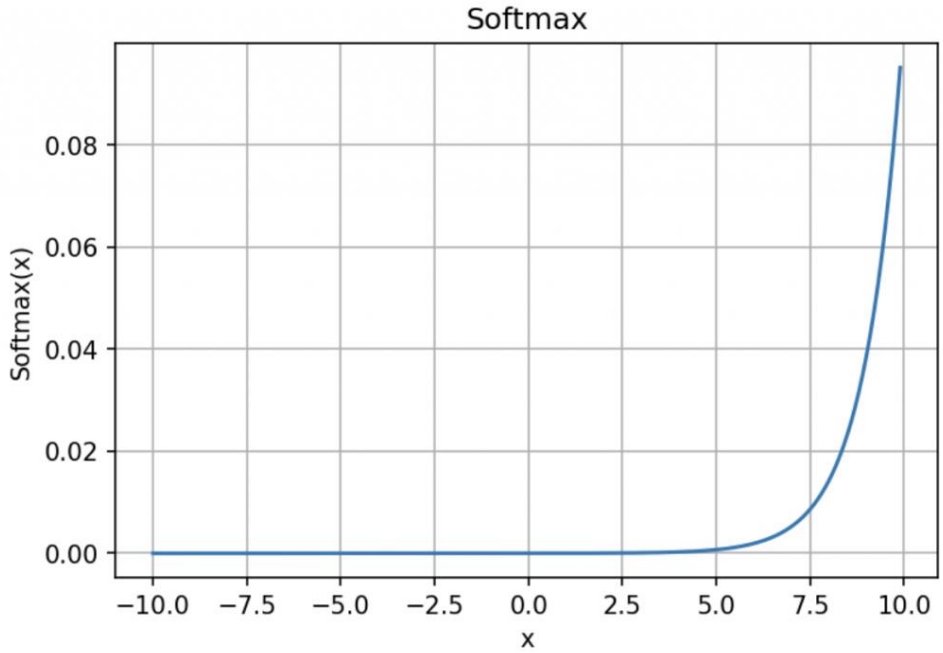
https://medium.com/

Activation functions

- Softmax
  - Outputs a vector of values that sum to 1.0 (can be interpreted as probabilities of class membership – related to argmax function)
  - Computed as: e^x /sum(e^x)
  - Target labels in output layer are vectors with 1 for target class and 0 for all other classes

Activation functions





https://towardsdatascience.com/

https://blog.robofied.com/

Choosing an activation function

- Points to be considered
  - Choose ReLU in hidden layers only
  - In binary classification – use sigmoid in output layer
  - In multi-class classification – use softmax in output layer

https://medium.com/

Multi-class classification

- More than two class labels
- Examples:
  - Face classification
  - Fruits and vegetables recognition
  - Optical character recognition
  - Hand written digit recognition
- Examples are classified as belonging to one among a range of known classes
- Create one hot encoding for the output:
  - creating dummy variables from a categorical variable

One hot encoding

		Tiger	Cat	Aeroplane
Tiger	→	1	0	0
Cat	→	0	1	0
Aeroplane	→	0	0	1
Cat	→	0	1	0

Categorical Cross-entropy Loss

$\hat{y}$ = [0.2 0.7 $\hat{y}_i$ 0.1]

y = [0 1 y_i 0]

- Loss is computed as follows:

  Loss = - $\sum (y_i * \log \hat{y}_i)$

for i = number of scalar values in model output

  - Usually, equal to no. of classes
  - $\hat{y}_i$ is i^{th} scalar value in model output
  - y_i is corresponding target value