SWE x GDSC presents...

EmpowHer

empowering women in stem

We will have **pizza**, **boba**, & a **raffle** for some Google Cloud swag!

industry panel	student panel
Dahlia Garas Mariza Stickle Tonya Kuhl Anchal Lamba	Grisha Bandodkar Aine Keenan Kavya Khare

Come join us for an inspiring conversation with 7 women in STEM who are all UC Davis students, faculty, or alumni!

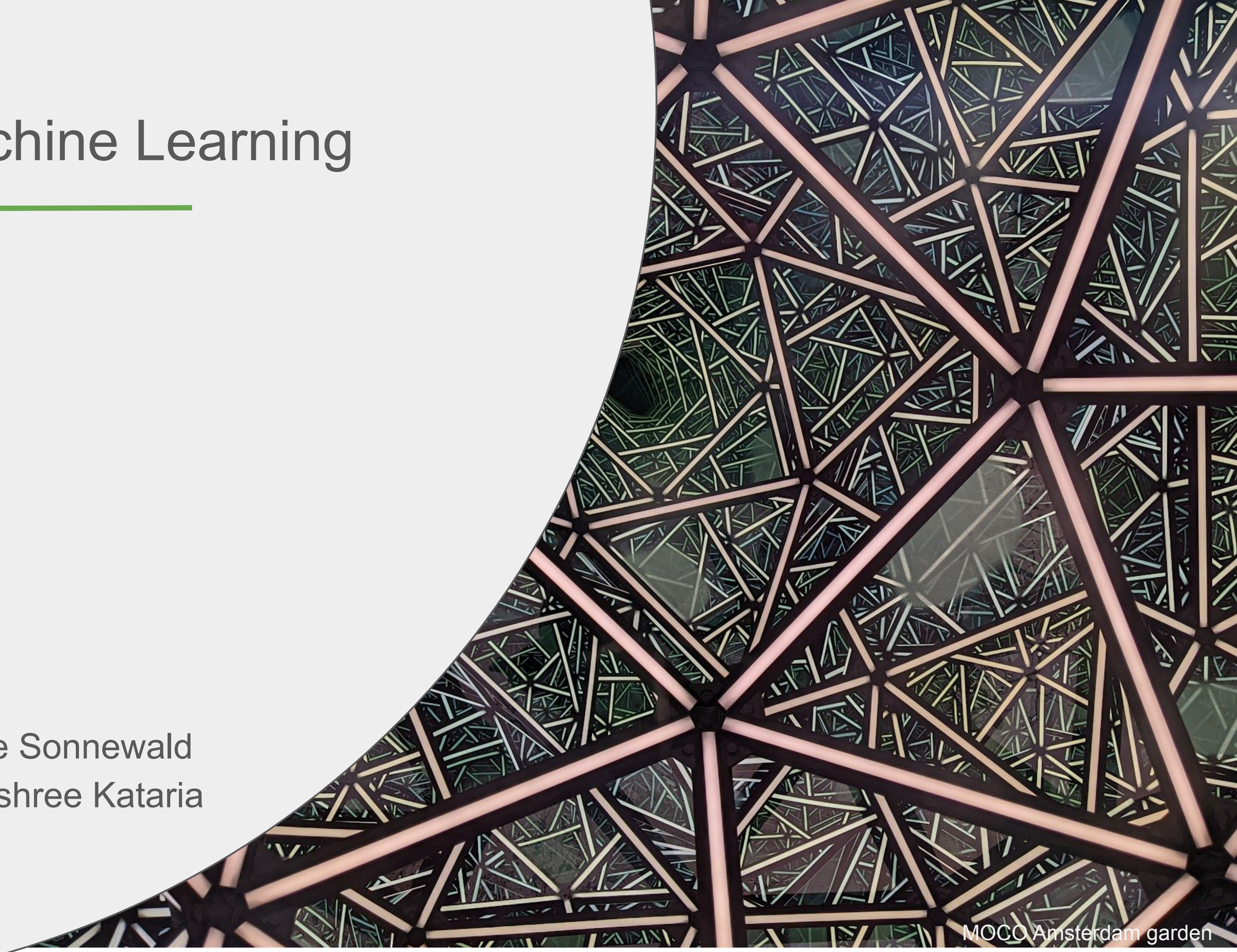
February 6, 2024 | 7:00 - 8:30pm | Walker 1310

UC DAVIS COMPUTER SCIENCE swe

ECS171: Machine Learning

L9:

Instructor: Prof. Maike Sonnewald
TAs: Pu Sun & Devashree Kataria



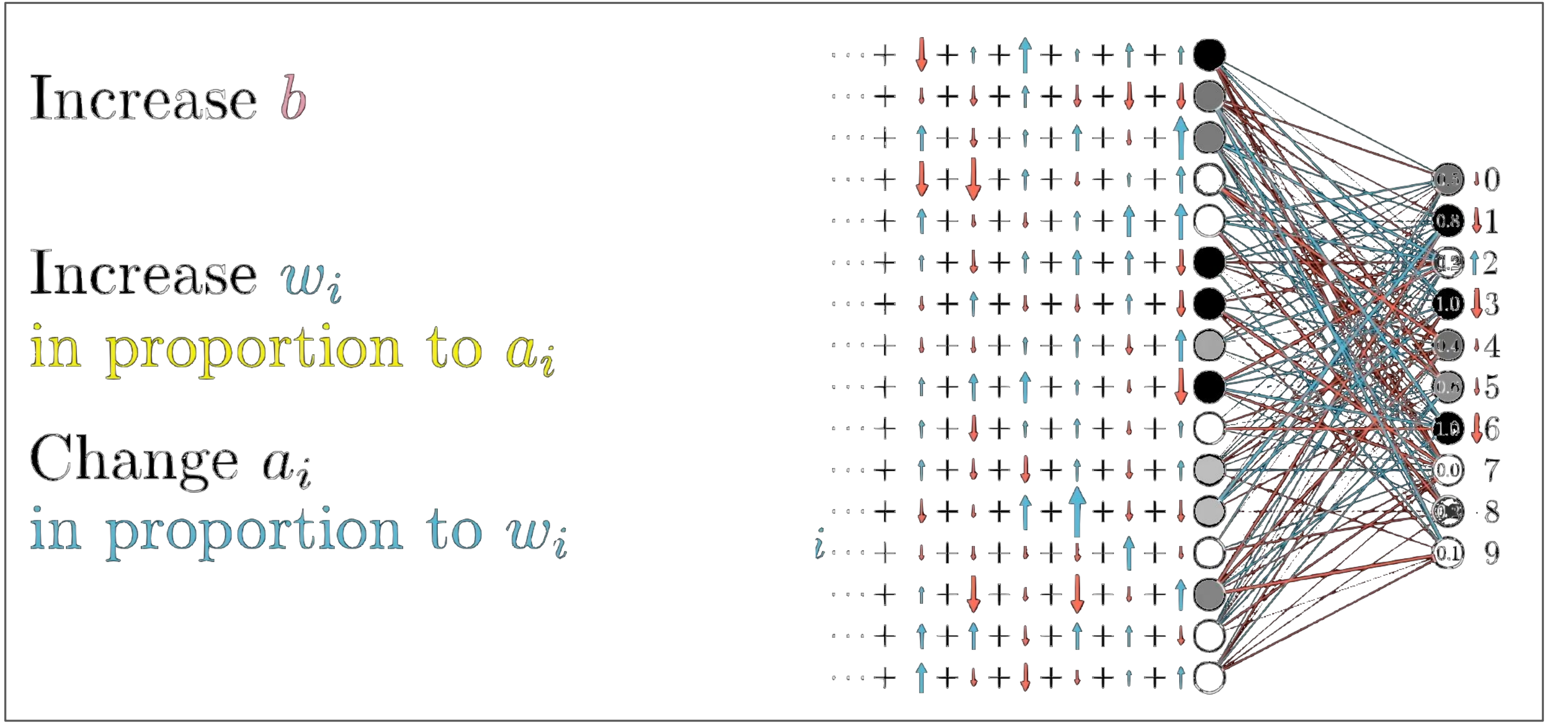
MOCO Amsterdam garden

Intended Learning Outcomes

- Describe how deep neural networks differ from more simple ones
    - Particular attention to the importance of hyperparameters (some similarities with simple networks)
- Be familiar with different gradient updates and when to use them
    - Be cognisant of what they mean in terms of potential issues including oscillations
    - Understand and apply the ‘momentum’ concept
- Be familiar with and apply different search strategies
- Be familiar with and apply different regularisation techniques
- Be familiar with and apply stopping criteria

Training a neural network: Backpropagation

The cost for one data-point x



Increase b

Increase w_i
in proportion to a_i

Change a_i
in proportion to w_i

Training a neural network: Backpropagation

	x_1	x_2	x_3	x_4	x_5	x_6	$\cdots$		
w_0	−0.08	+0.02	−0.02	+0.11	−0.05	−0.14	$\cdots$	→	−0.08
w_1	−0.11	+0.11	+0.07	+0.02	+0.09	+0.05	$\cdots$	→	+0.12
w_2	−0.07	−0.04	−0.01	+0.02	+0.13	−0.15	$\cdots$	→	−0.06
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		$\vdots$
$w_{13,001}$	+0.13	+0.08	−0.06	−0.09	−0.02	+0.04	$\cdots$	→	+0.04

Average over all training examples

Training a neural network: Backpropagation

Recall: Gradient weight update described as: $\mathbf{w}^{t+1} = \mathbf{w}^{t} - \lambda \nabla J(\mathbf{w}^{t})$

$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad \cdots$

$$-\lambda \nabla J(w_1, w_2, ...w_n) = \begin{pmatrix} -0.08 \\ +0.12 \\ -0.06 \\ \vdots \\ +0.04 \end{pmatrix}$$

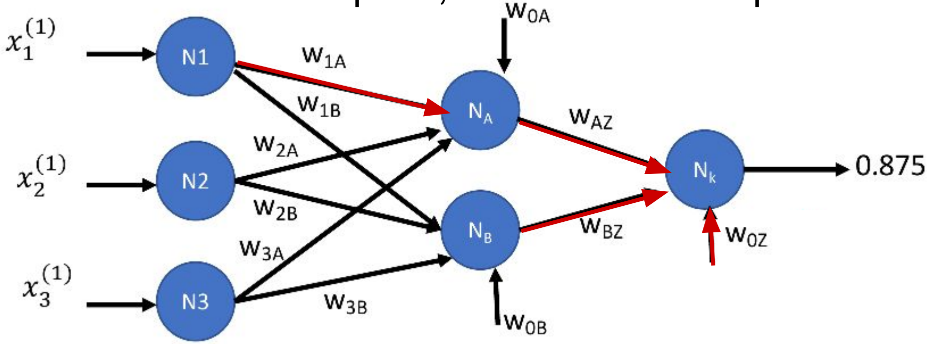
Cost for one update/step down the gradient is considerable

Backward pass: Updating W$_{1A}$

$learning\ rate\ ; 0 \leq \eta \leq 1$
$Assume:\ \eta$ = 0.1

X_i:N$_1$, N$_2$, N$_3$
$\sigma(Z_j)$: N$_A$, N$_B$
$\sigma(Z_k)$: N$_z$

- Feed forward neural network learning in two phases:
  - a forward pass, and a backward pass



x_1=N$_1$ =0.4	N$_A$ =0.7892
x_2=N$_2$ =0.2	N$_B$ =0.8176
x_3=N$_3$ =0.7	N$_z$ =0.875

w$_{0A}$ =0.5	w$_{0B}$ =0.7	w$_{0Z}$=**0.5008**
w$_{1A}$ = 0.6	w$_{1B}$ = 0.9	w$_{AZ}$=0.90067
w$_{2A}$=0.8	w$_{2B}$=0.8	w$_{BZ}$=0.90069
w$_{3A}$=0.6	w$_{3B}$=0.4	

Assume actual y= 0.8 →
residual error = 0.875 – 0.8 = 0.075

$$Z_{N_A} = \omega_{0A} + \omega_{1A}x_1^{(1)} + \omega_{2A}x_2^{(1)} + \omega_{3A}x_3^{(1)} = 1.32$$

$$\Delta w_{hidden}$$
$$= residual\ error * error\ of\ the\ hidden\ layer * weighted\ error\ of\ the\ output\ layer$$

$$\Delta w_{hidden} = (\hat{y} - y) * \frac{\partial\sigma(Z_j)}{\partial W_{ij}} * W_{jk}\frac{\partial\sigma(Z_k)}{\partial W_{jk}}$$

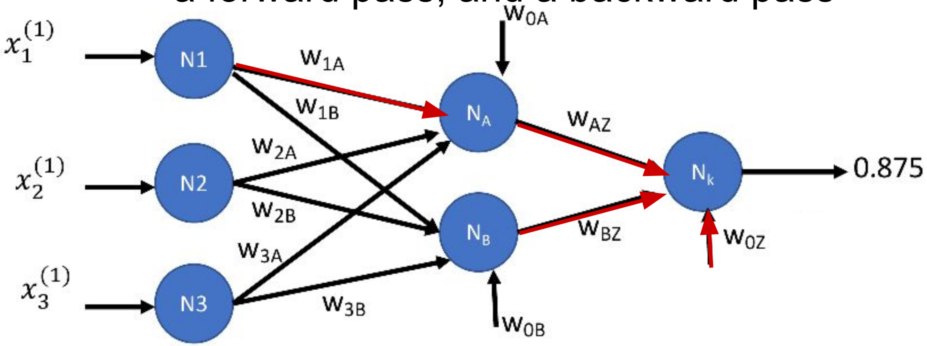
Backward pass: Updating W_{1A}

$learning\ rate\ ; 0 \leq \eta \leq 1$
$Assume: \ \eta = 0.1$

X_i: N_1, N_2, N_3
$\sigma(Z_j)$: N_A, N_B
$\sigma(Z_k)$: N_z

- Feed forward neural network learning in two phases:
  - a forward pass, and a backward pass

$x_1=N_1=0.4$	$N_A=0.7892$
$x_2=N_2=0.2$	$N_B=0.8176$
$x_3=N_3=0.7$	$N_z=0.875$



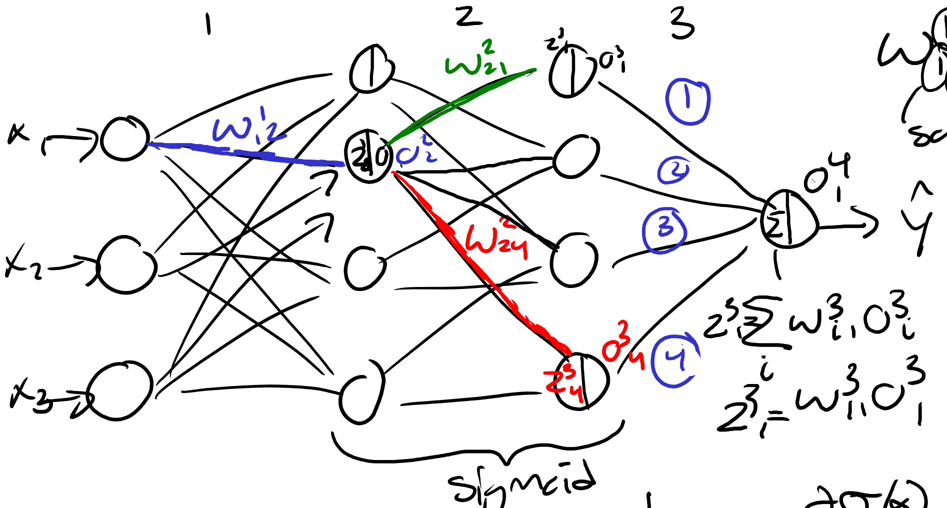
$w_{0A}=0.5$	$w_{0B}=0.7$	w_{0Z}=**0.5008**
$w_{1A}=0.6$	$w_{1B}=0.9$	w_{AZ}=0.90067
w_{2A}=0.8	w_{2B}=0.8	w_{BZ}=0.90069
w_{3A}=0.6	w_{3B}=0.4	

Assume actual y= 0.8 →
residual error = 0.875 – 0.8 = 0.075

$$Z_{N_A} = \omega_{0A} + \omega_{1A}x_1^{(1)} + \omega_{2A}x_2^{(1)} + \omega_{3A}x_3^{(1)} = 1.32$$

$$\Delta w_{hidden} \\ = residual\ error * error\ of\ the\ hidden\ layer * weighted\ error\ of\ the\ output\ layer$$

$$\Delta w_{hidden} = (\hat{y} - y) * \frac{\partial\sigma(Z_j)}{\partial W_{ij}} * W_{jk}\ \frac{\partial\sigma(Z_k)}{\partial W_{jk}}$$



$w^{(k)}_{(i)(j)}$ source dest

$$z^1_2 = \sum_{i=1}^{3} w^1_{i2} o^1_i$$

$$\boxed{z^1_2 \mid o^1_2}$$

$$z^3_1 \sum_i w^3_{i1} o^3_i$$

$$z^3_1 = w^3_{11} o^3_1 + w^3_{41} o^3_4$$

$$z^3_4 = \begin{matrix} \dots \\ + w^2_{24} o^2_4 \\ \dots \end{matrix}$$

$$z^3_1 = \begin{matrix} \dots \\ + w^2_{21} \bigcirc\!\!\!\!\!\; o^2_2 + \\ \dots \end{matrix}$$

$$\sigma(x) = \text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\left[1-\sigma(x)\right]$$

$$\frac{\partial L}{\partial w^2_{21}} = ?, = \underbrace{\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o^3_1} \frac{\partial o^3_1}{\partial z^3_1} \frac{\partial z^3_1}{\partial w^2_{21}}}$$

$$f(x) - 3x + 4x$$

$$\frac{\partial f}{\partial x} = 3 + 4 = 7$$

$$\frac{\partial L}{\partial w^2_{24}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o^3_4} \frac{\partial o^3_4}{\partial z^3_4} \frac{\partial z^3_4}{\partial w^2_{24}}$$

$$\frac{\partial L}{\partial w^1_{12}} = ?, \underbrace{\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o^3_1} \frac{\partial o^3_1}{\partial z^3_1} \frac{\partial z^3_1}{\partial o^2_2}} \frac{\partial o^2_2}{\partial z^2_2} \frac{\partial z^2_2}{\partial w^1_{12}} \text{ (1)} +$$

$$\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o^3} \frac{\partial o^3_4}{\partial z^3_4} \frac{\partial z^3_4}{\partial o^2_2} \frac{\partial o^2_2}{\partial z^2_2} \frac{\partial z^2_2}{\partial w^1_{12}} \text{ (4)} +$$

$$\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o^3_3} \frac{\partial o^3_3}{\partial z^3_3} \frac{\partial z^3_3}{\partial o^2_2} \frac{\partial o^2_2}{\partial z^2_2} \frac{\partial z^2_2}{\partial w^1_{12}} \text{ (3)} + \text{ (2)}$$

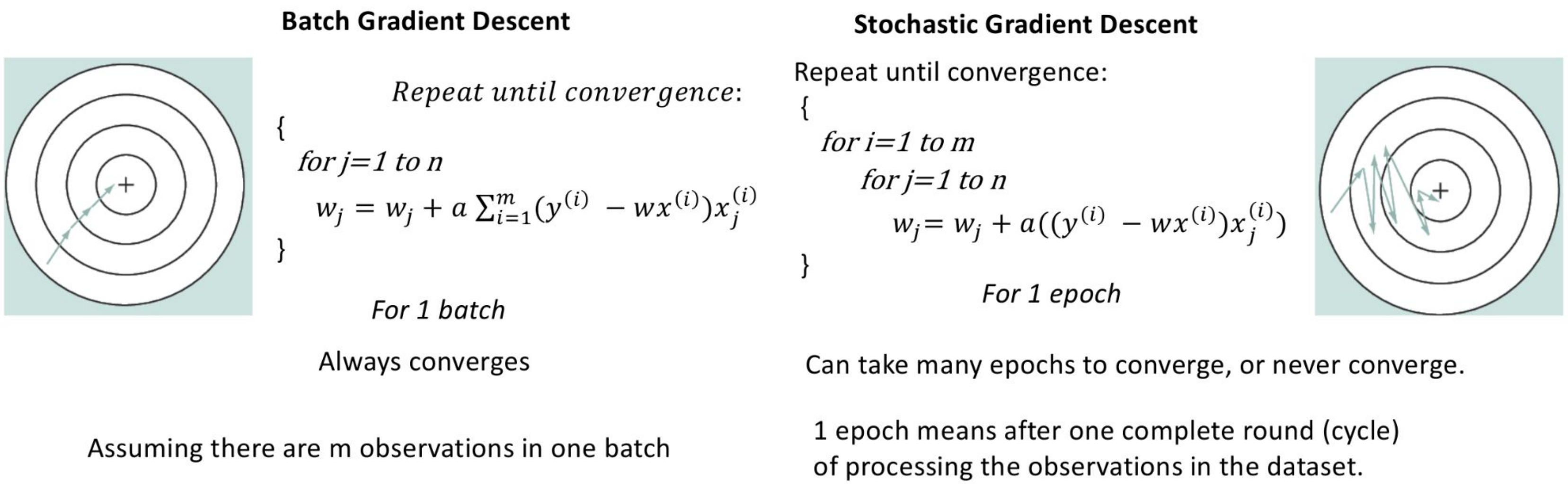
$$z^3_1 = \sum_i w^2_{i1} o^2_i$$

$$\frac{\partial z^3_1}{\partial o^2_2} = w^2_{21}$$

$$\hat{y} = o^4_1 \qquad \frac{\partial \hat{y}}{\partial o^4_1} = 1 \qquad \frac{\partial \hat{y}}{\partial z^4_1}$$

Training a neural network:

Recall from previous lecture: batch and stochastic GD



Batch Gradient Descent

```
Repeat until convergence:
{
  for j=1 to n
    $w_j = w_j + a \sum_{i=1}^{m} (y^{(i)} - wx^{(i)}) x_j^{(i)}$
}
```

For 1 batch

Always converges

Assuming there are m observations in one batch

Stochastic Gradient Descent

```
Repeat until convergence:
{
  for i=1 to m
    for j=1 to n
      $w_j = w_j + a((y^{(i)} - wx^{(i)}) x_j^{(i)})$
}
```

For 1 epoch

Can take many epochs to converge, or never converge.

1 epoch means after one complete round (cycle) of processing the observations in the dataset.

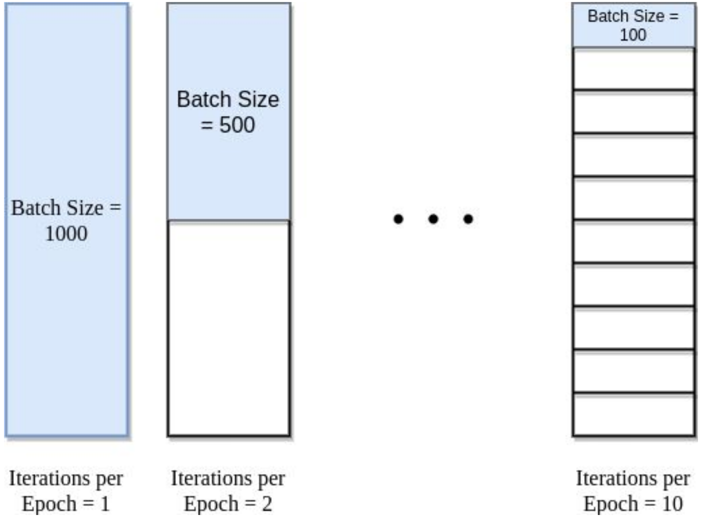
Batch and epochs

Batch: Number of datapoint partitions considered in training

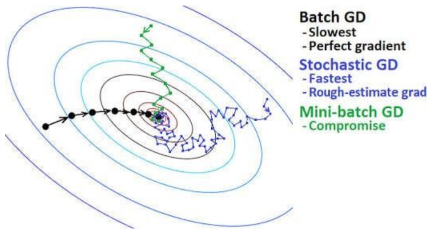
- Need to have equal size
- If batch = 1 we're using all the data

Epoch: Number of batches

- Need to go through all to have one pass through.

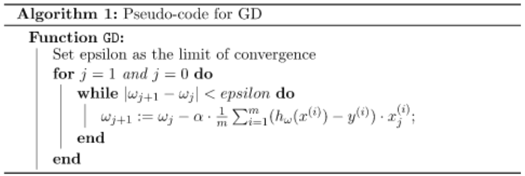


Batch, Stochastic and mini-batch GD

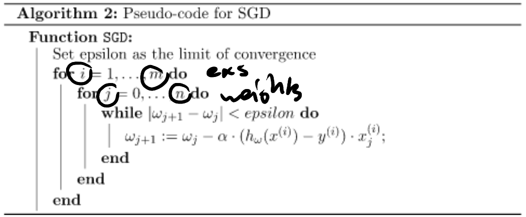


i is iteration index, alpha is the learning ratehere, m is the number of training samples, b are the batches
Hypothesis (aka y hat): $h_\omega(x) = \omega_0 + \omega_1 x$

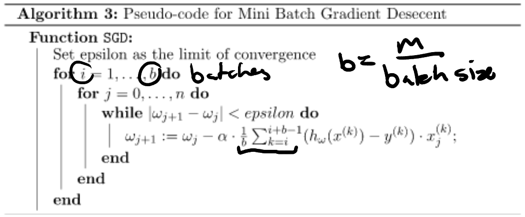
Const function: $J(\omega_0, \omega_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\omega(x^{(i)}) - y^{(i)})^2$



```
Algorithm 1: Pseudo-code for GD
Function GD:
    Set epsilon as the limit of convergence
    for $j = 1$ and $j = 0$ do
        while $|\omega_{j+1} - \omega_j| < epsilon$ do
            $\omega_{j+1} := \omega_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (h_\omega(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$;
        end
    end
```



```
Algorithm 2: Pseudo-code for SGD
Function SGD:
    Set epsilon as the limit of convergence
    for $i = 1, \ldots, m$ do   exs
        for $j = 0, \ldots, n$ do   weights
            while $|\omega_{j+1} - \omega_j| < epsilon$ do
                $\omega_{j+1} := \omega_j - \alpha \cdot (h_\omega(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$;
            end
        end
    end
```



```
Algorithm 3: Pseudo-code for Mini Batch Gradient Desecent
Function SGD:
    Set epsilon as the limit of convergence           $b = \frac{m}{\text{batch size}}$
    for $i = 1, \ldots, b$ do   batches
        for $j = 0, \ldots, n$ do
            while $|\omega_{j+1} - \omega_j| < epsilon$ do
                $\omega_{j+1} := \omega_j - \alpha \cdot \underline{\frac{1}{b} \sum_{k=i}^{i+b-1}} (h_\omega(x^{(k)}) - y^{(k)}) \cdot x_j^{(k)}$;
            end
        end
    end
```

Batch:

- Take all the data and iterate through

Stochastic:

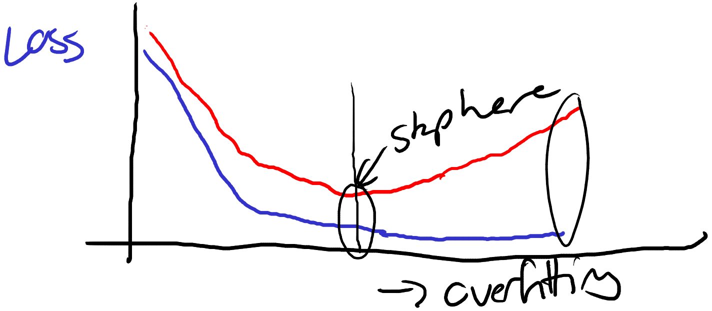
- Take a randomly shuffled subset of the data
- the gradients are calculated on one random shuffled part out of partitions (could be more)

Mini-batch

- The gradients are calculated with subsets of all observations in each iteration

Training a Neural Network using Stochastic GD

- For each training input perform the following to complete 1-epoch:
    - Complete one feed-forward pass
    - Compute the squared loss for the training sample
    - Complete one backward pass (backpropagation) and update the weights using the gradient descent weight update rule
- Measure the mean squared loss (i.e., error) for each epoch and plot (for the training set and the test set)
    - This average loss is plotted over time (epochs) to monitor the progress of the training and to identify when the network has converged to a minimum loss



1. early stopping
if no improvement
stop, save

2 checkpoints
every n steps
save
disk-intensive!

Batch/’mini-batch’ gradient descent

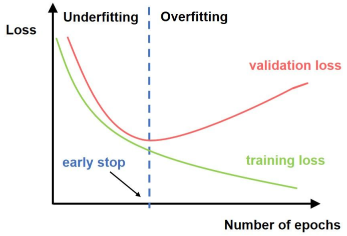
- For the training inputs in each batch, perform the following to complete 1-epoch:
  - Complete one feed-forward pass
  - Compute the average squared loss for the training samples in the batch
  - Complete one backward pass (backpropagation) and update the weights using the gradient descent weight update rule

- Measure the mean squared loss (i.e., error) of the epochs for each epoch and plot (for the training set and the test set)

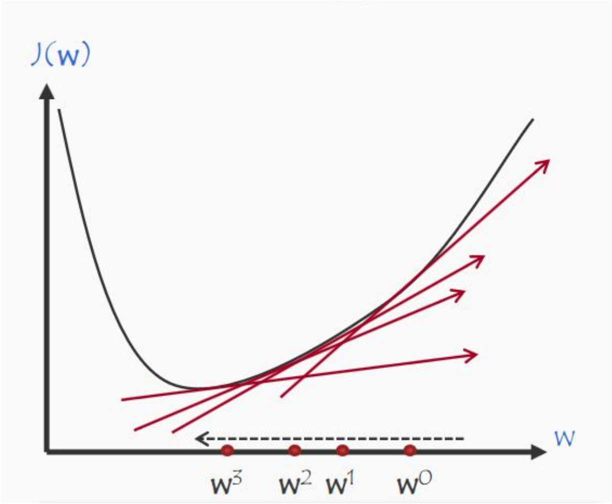
- This average loss is plotted over time (epochs) to monitor the progress of the training and to identify when the network has converged to a minimum loss.

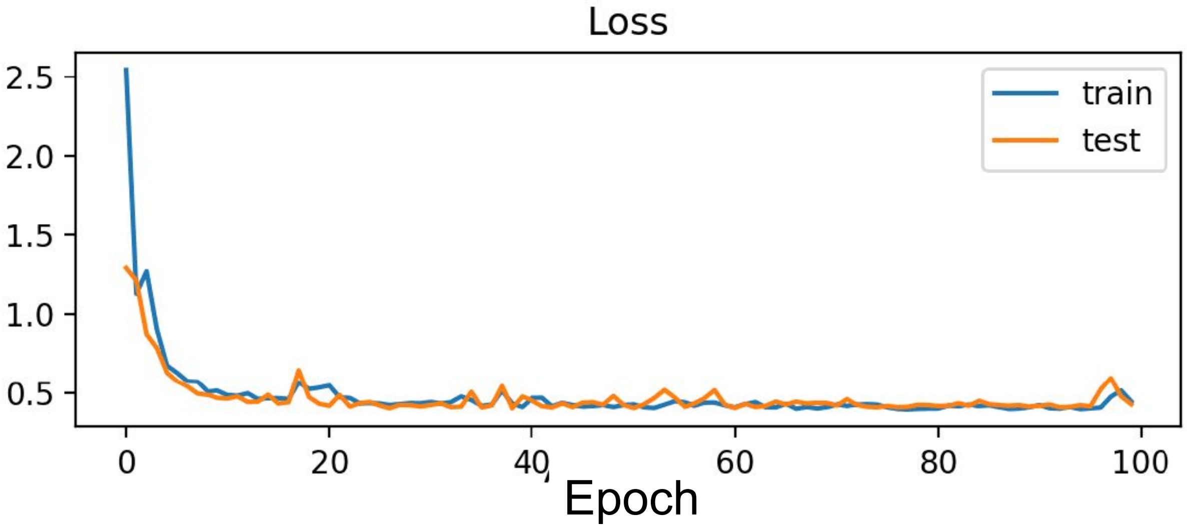
- If all the training data is in one batch, the weight update in neural network training would be done once per epoch, after computing the gradients of the loss with respect to the weights using the entire batch.

Plotting the loss using gradient descent



Recall: Gradient weight update described as: $\mathbf{w}^{t+1} = \mathbf{w}^{t} - \lambda \nabla J(\mathbf{w}^{t})$





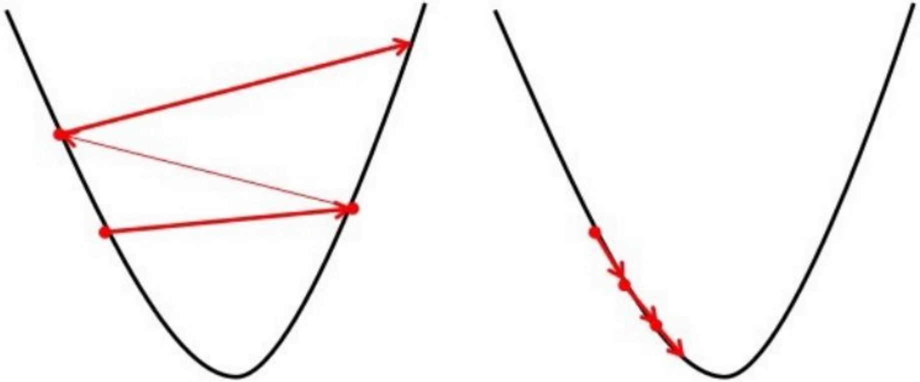
Deep Neural Networks

- Until now we have largely used simple neural networks for our examples
- Deep neural networks (DNNs) have multiple hidden layers between the input and output layers.
- These hidden layers allow the network to learn more complex and abstract representations of the input data, which can lead to better performance on tasks:
  - Example: mage recognition, natural language processing, and speech recognition.
- Basics: each layer consists of multiple neurons that perform a weighted sum of the inputs and apply a non-linear activation function to the result.
- The outputs of the neurons in one layer serve as the inputs to the next layer, and this process is repeated until the final output layer produces a prediction or decision based on the input.

Hyperparameters are important controls for performance

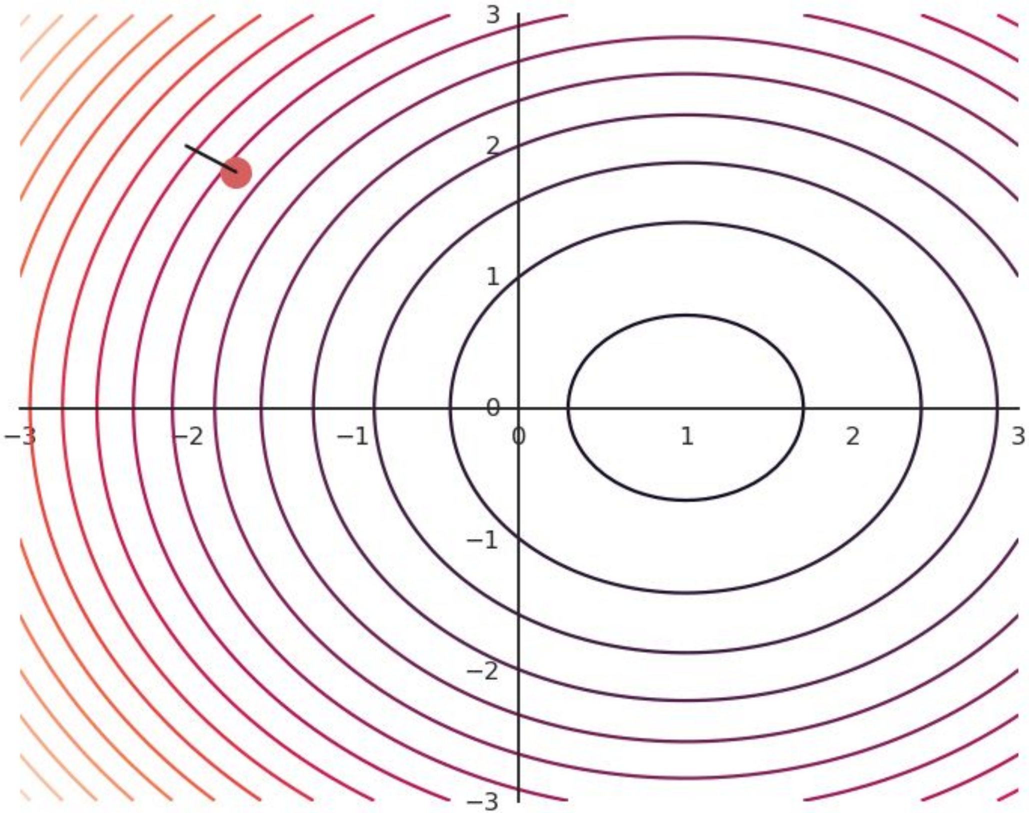
- Hyperparameters are variables that are not learned automatically
- The weights of the model are **not** hyperparameters
- They can increase accuracy and control how the network is trained.
- Examples of Hyperparameters:
    - Number of hidden layers
    - Number of neurons at each hidden layer
    - Hidden layer activation function
    - Momentum term value
    - Learning rate value
    - Number of epochs
    - Batch size in mini-batch
    - Regularization Parameter value

Recall: Oscillations are problematic in training



- There is a trade-off between the learning rate and the momentum term
- This is particularly common when tuning parameters like batch size and using stochastic gradient descent

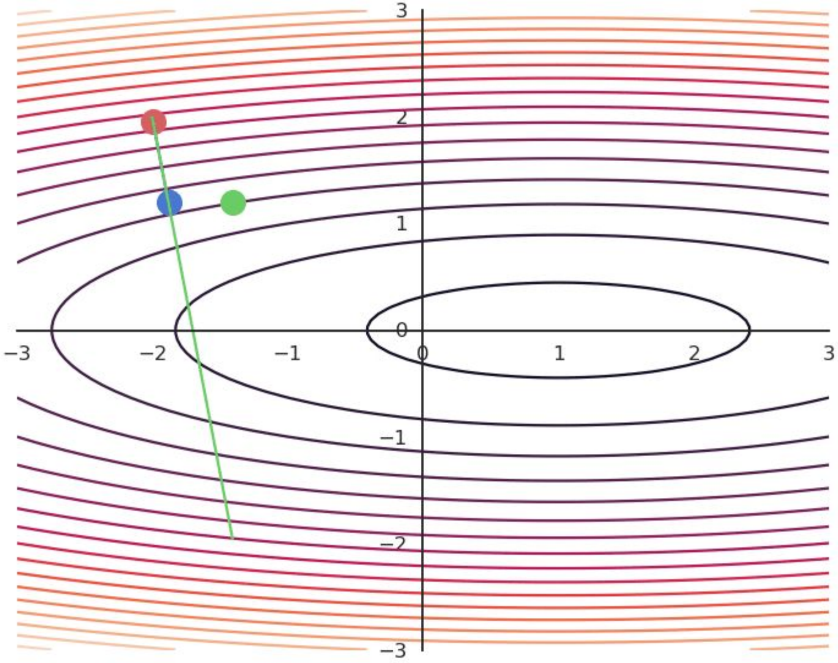
If the loss landscape is simple oscillations are rare



https://julien-vitay.net/lecturenotes-neurocomputing/2-linear/1-Optimization.html

Unfortunately, simple loss landscapes are rare

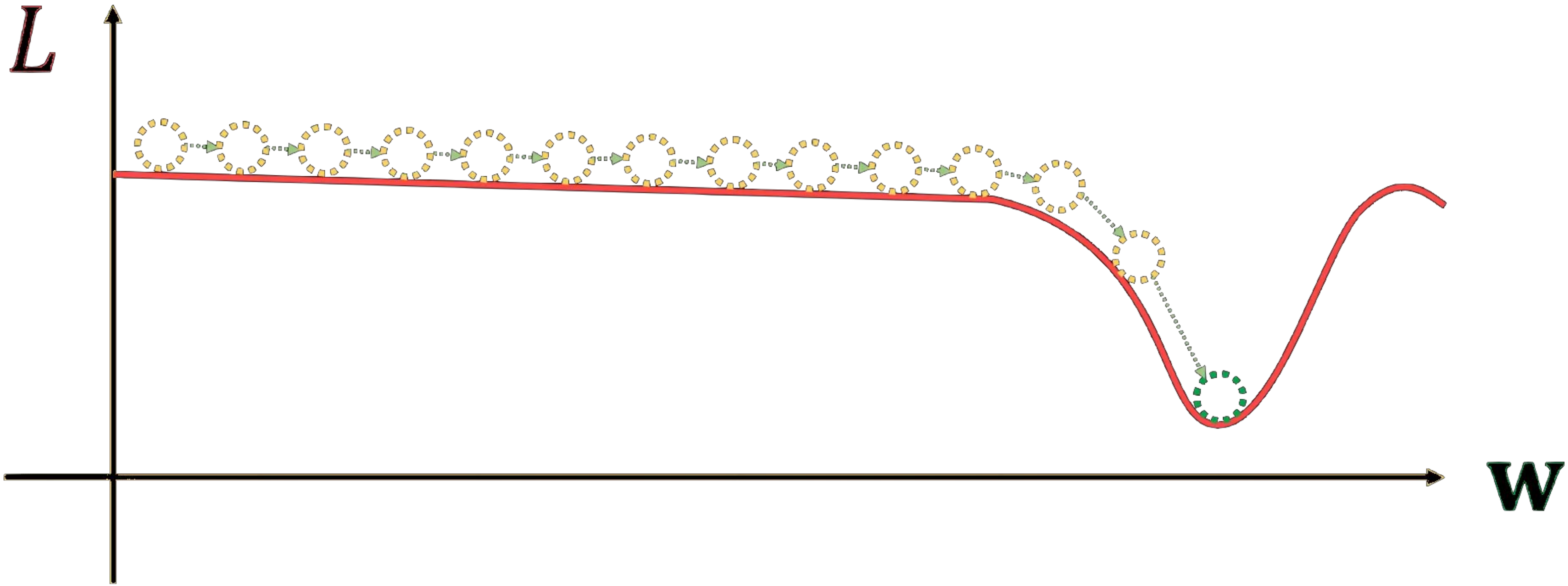
- The learning rate has a big impact on how fast and if a model converges
- Red: small steps and converges slowly
- Green: big steps, could miss
- Blue: in-between
- Not knowing what the ‘landscape’ looks like makes estimating the learning rate an important question



https://julien-vitay.net/lecturenotes-neurocomputing/2-linear/1-Optimization.html

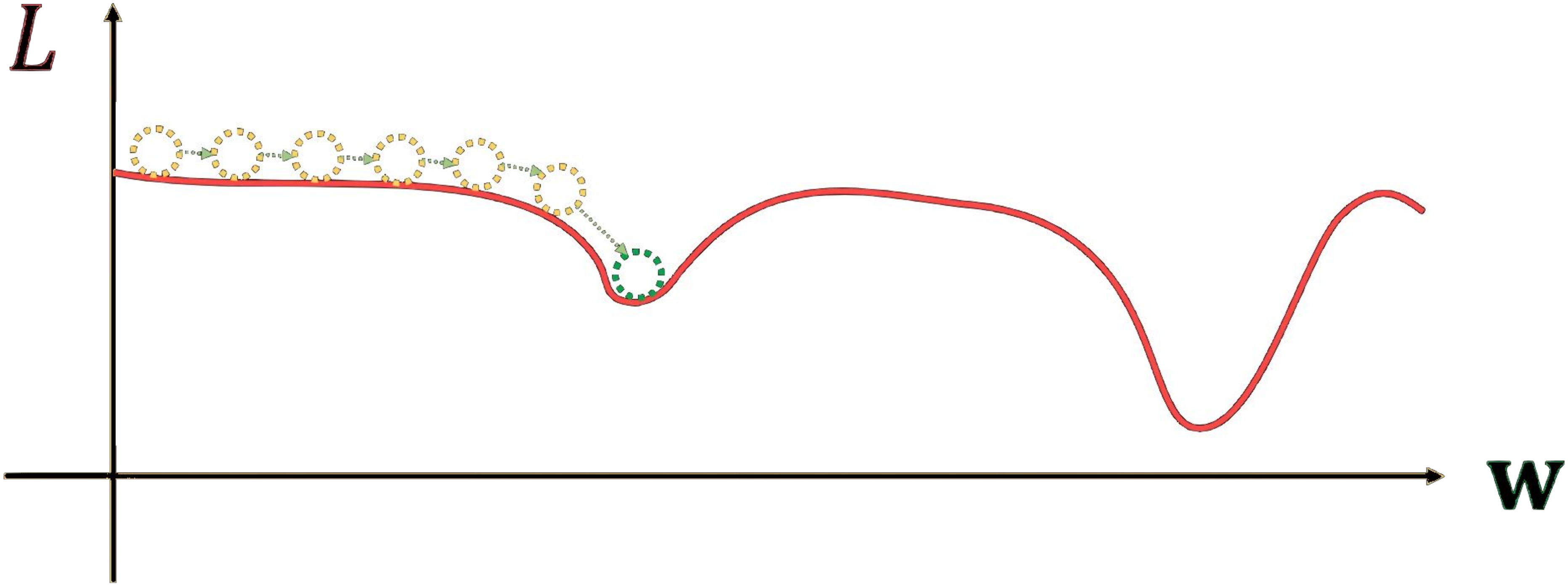
Low learning rate (λ)

$$\mathbf{w}_{t-1} = \mathbf{w}_t - \lambda g(\mathbf{w}_t)$$



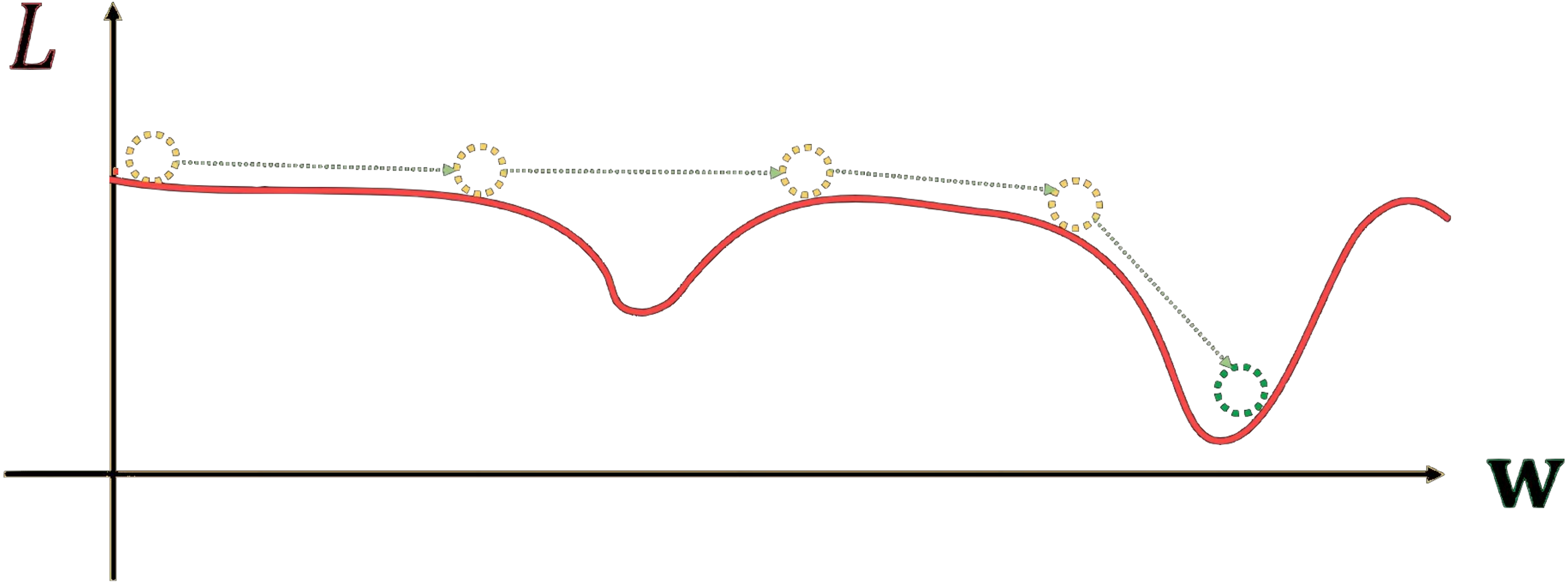
Low learning rate (λ) can get stuck

$$\mathbf{w}_{t-1} = \mathbf{w}_t - \lambda g(\mathbf{w}_t)$$



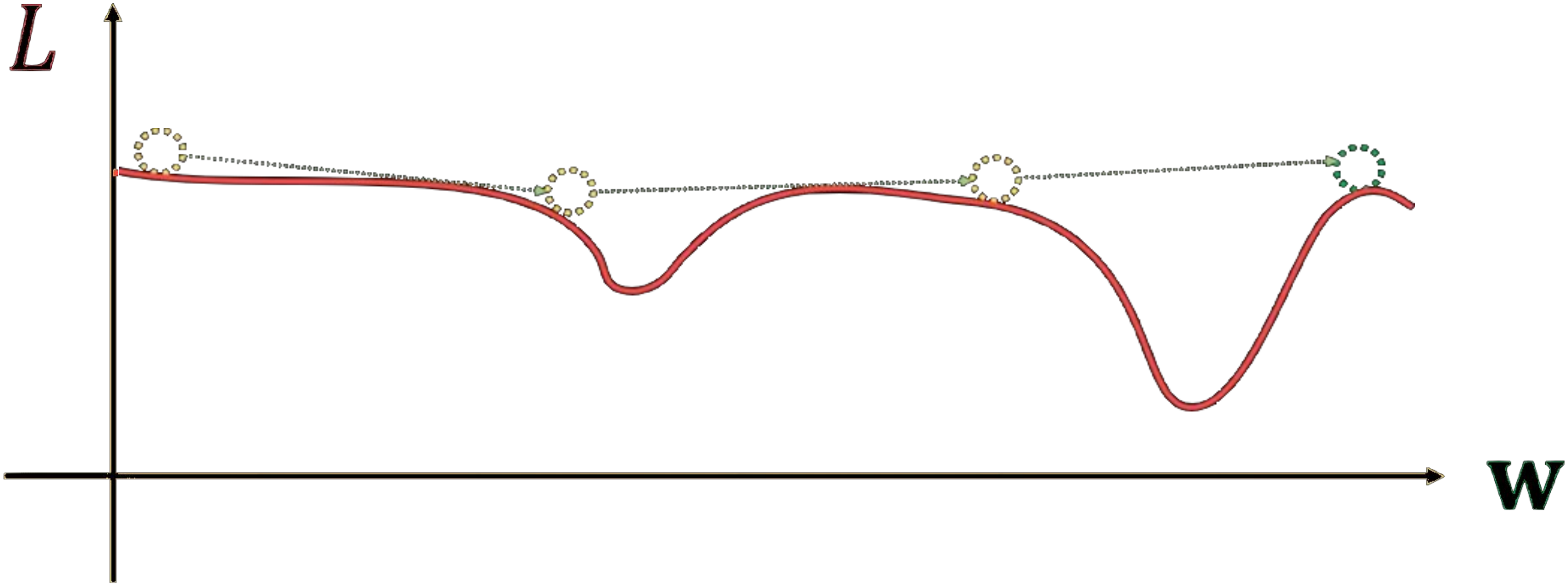
High learning rate (λ) is much faster

$$\mathbf{w}_{t-1} = \mathbf{w}_t - \lambda g(\mathbf{w}_t)$$



High learning rate (λ) can miss

$$\mathbf{w}_{t-1} = \mathbf{w}_t - \lambda g(\mathbf{w}_t)$$



Adding momentum

Take the updates for stochastic gradient descent:

$$\mathbf{w}_{t-1} = \mathbf{w}_t - \lambda g(\mathbf{w}_t)$$

Where we now denote the stochastic part writing g: $g(w) = \frac{\partial J(w_t)}{\partial \mathbf{w}}$

We can introduce ‘velocity’ as: $\mathbf{w}_{t-1} = \mathbf{w}_t - v_{t+1}$

Where the **v** term is:

$$\mathbf{v}_{t-1} = \rho \mathbf{v}_t - \lambda g(\mathbf{w}_t)$$

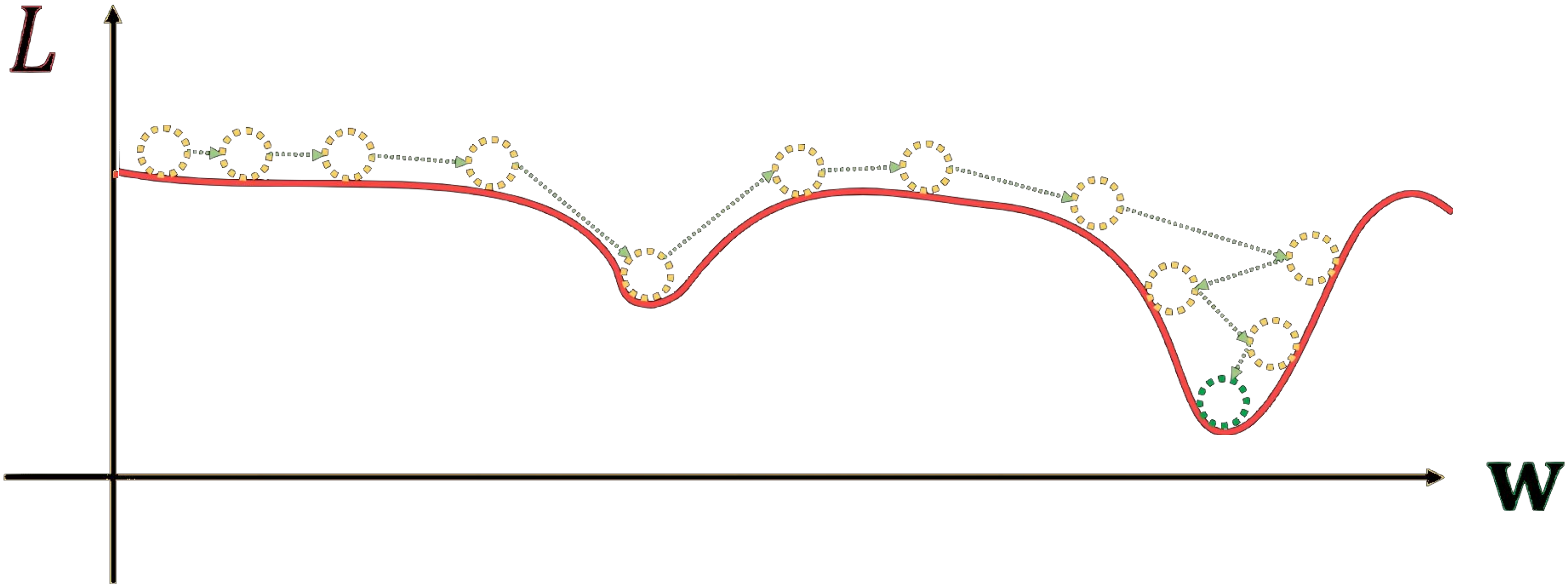
Here, the rho is the ‘momentur

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

Where the second RHS is the ‘velocity jump’ and the third is the ‘gradient jump’

Adding the ‘momentum’ parameter

$$\mathbf{w}_{t-1} = \mathbf{w}_t - \lambda g(\mathbf{w}_t)$$



Momentum

Momentum is a term used in optimization algorithms such as stochastic gradient descent with momentum. It can also be used with mini-batch GD optimization method

It is a technique used to speed up the convergence of the optimization process and prevent the optimization from getting stuck in local minima

Backpropagation is made more powerful with the addition of a momentum term α

Momentum (α) helps to know the direction of the next step with the knowledge of the previous steps.

It helps to prevent oscillations. The momentum term represents inertia.

The momentum term helps to smooth out the oscillations in the gradient updates, which can result in faster convergence to the optimal solution.

It allows the optimization process to continue in the same direction as the previous updates, which can help to navigate around small bumps or ridges in the optimization landscape.

Momentum

Small values of α reduce the inertial effects as well as the influence of the recent adjustments, until α=0.

Larger values of α allows the algorithm to move in the same direction as the previous weight adjustments.

Commonly, momentum is first initialized to 0.9 , but it is then tuned similar to learning rate (learning rate is a value between 0 and 1), during the training process.

A typical choice of momentum is between 0.5 and 0.9.

It is common to start with a low momentum value (e.g., 0.5) at the beginning of training, and then gradually increase it as training progresses (e.g., to 0.9).

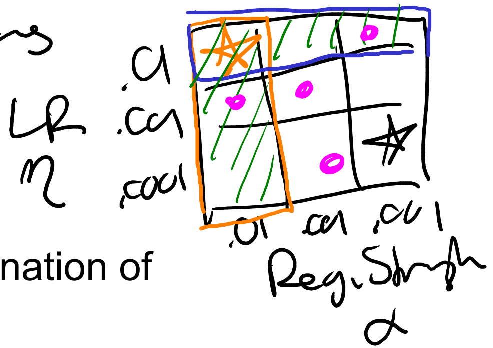
A low momentum value allows the optimization process to explore the optimization landscape more widely in the early stages of training, while a higher momentum value can help the optimization process to converge more quickly towards the global minimum in the later stages of training. Learning w rate value can also be adopted over epochs.

Hyperparameter

• Random /// greedy 9 options

★ global optimum

Optimisation strategies: Greedy Search



- Through controlled experiments where each experiment is one combination of hyperparameter values.

  a) Keep all hyperparameters constant except one. (Greedy optimization)

  b) Analyze the effect and make decision about which hyperparameter to change next.

  c) Repeat.

- Drawback: each experiment requires training the model end-to-end, and can be very slow for Deep Learning algorithms (e.g., CNN)

① Greedy ② Exhaustive ③ Random

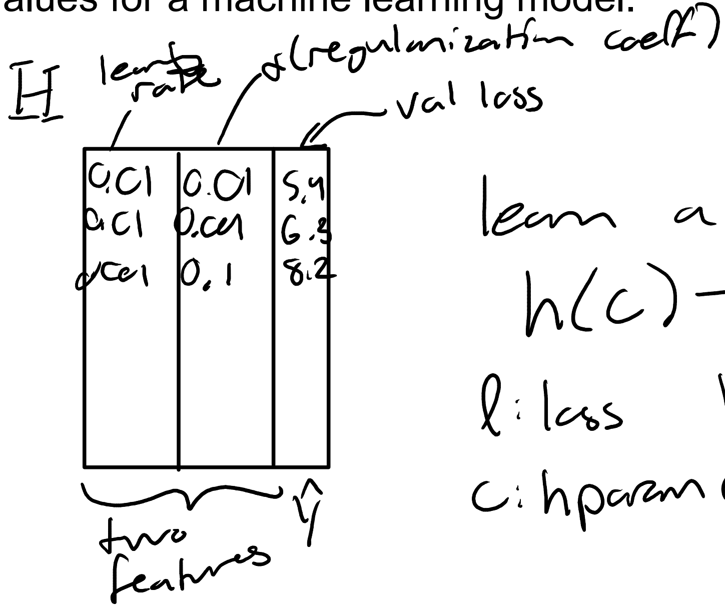
Optimisation strategies: Alternatives

Other model hyperparameter optimization techniques offer ways to automatically find the best possible combination of hyperparameter values for a machine learning model.

Methods include:

- Random search
- Bayesian optimization
- Hyperband

Optimization is a non-trivial task…



learn rate	α (regularization coeff)	val loss
0.01	0.01	5.4
0.01	0.001	6.3
0.001	0.1	8.2

learn a meta-model

$h(c) \rightarrow l$

l: loss h: meta-model

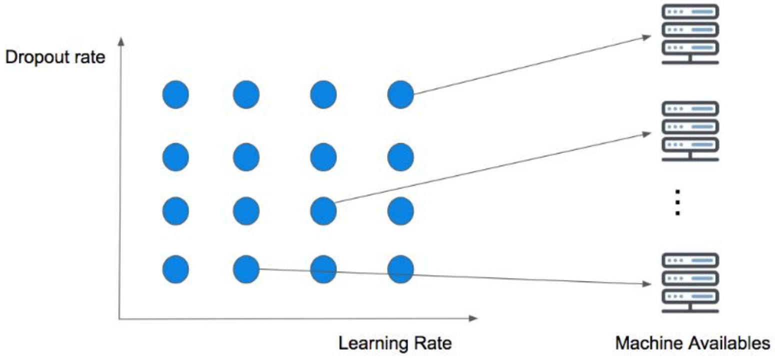
c: hparam configs

Optimisation strategies: Grid Search

- Grid search automates the “trial-and-error” process
- Assume the following hyperparameters in an Artificial Neural Network taking multiple possible values:
    - Number of hidden layers :[1,2,3,4,5]
    - Number of neurons in each layer: [4,5,6,7,8]
    - Number of epochs: [10,30,50,70,100]
        - > **Total number of possible combinations: 5^3**

Grid Search

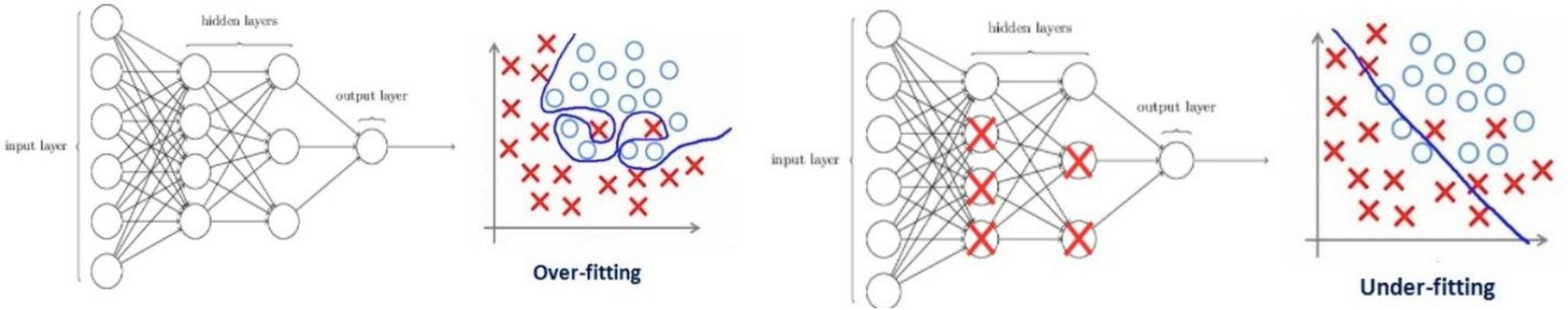
- Grid search is a technique that involves defining a grid of hyperparameters to explore
- For each point on the grid, we evaluate the performance of the model using the combination of hyperparameters on a validation set
- The hyperparameters that result in the best performance on the validation set are then chosen as the optimal hyperparameters



Example grid-search.ipynb

Regularisation: Preventing overfitting

- Regularization in deep neural networks is a technique used to prevent overfitting
- Occurs when a model is trained too well on the training data and does not generalize well to unseen data

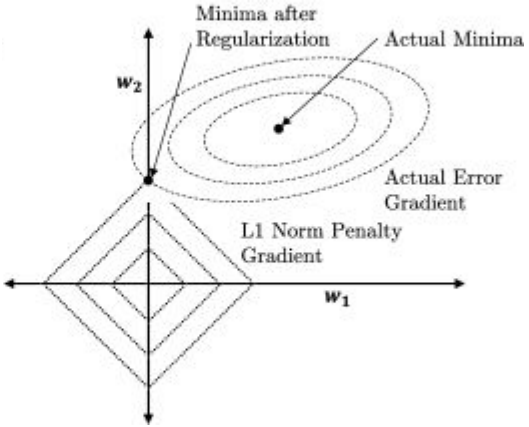


From L5: L1 and L2 regularization are convex functions

L1 Lasso: Sum of absolute values of the coefficient

$$J(\mathbf{w}) = \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}_{\text{MSE}} + \lambda \underbrace{\sum_{i=1}^{n}|w_i|}_{\text{Penalty term}}$$

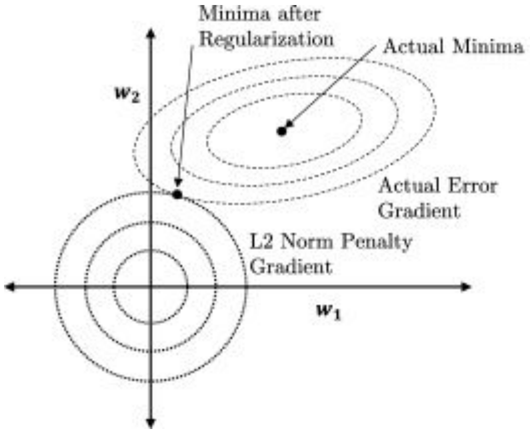
Encourages weights to be **exactly** zero



L2 'Ridge': Squared magnitude of the coefficient

$$J(\mathbf{w}) = \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}_{\text{MSE}} + \lambda \underbrace{\sum_{i=1}^{n} w_i^2}_{\text{Penalty term}}$$

Encourages weights to be **close** to zero

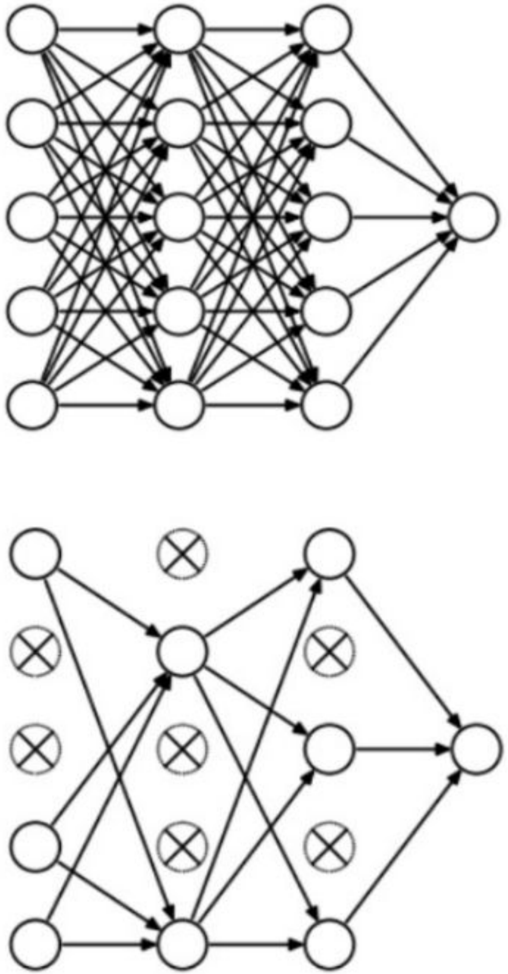


From L5: L1 and L2 regularization: Takeaways

- We can encourage the weights to be small by choosing as our regularizer the L2 penalty.
- Note: To be precise, the L2 norm is Euclidean distance, so we’re regularizing the squared L2 norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.
- If you fit training data poorly, J is large. If your optimal weights have high values, R is large.
- Large λ penalizes weight values more.
- Like M, λ is a hyperparameter we can tune with a validation set.

Dropout regularisation

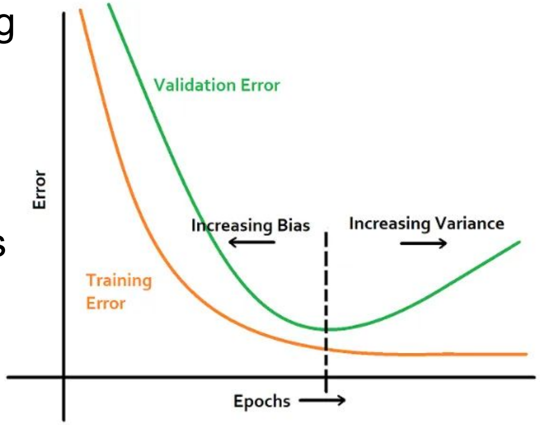
- At every iteration (1 feedforward pass followed by 1 backward pass):
    - Select random fraction of input units to remove along with all of their incoming and outgoing connections
- A Dropout rate of 0.2 means that 20% of the input units are randomly dropped during training.
- Similarly, a Dropout rate of 0.5 means that 50% of the input units are randomly dropped during training.
- The probability of choosing the number of nodes to be dropped is the hyperparameter of the dropout function.



Termination Criteria: When to stop training

Epoch: number of times the learning algorithm works through the training data
Batch: Epochs have one or more batches

- Criteria for terminating the training process can be dictate by:
  - Time: Risk degradation in model performance
  - Threshold of prediction error/minimum accuracy with training data: Risk overfitting
- Cross-validation procedure to determine when to stop training
  - Save a portion of the data not used for training and testing
  - For example: with k-fold cross-validation, the training data is divided into k subsets (or "folds")
  - The model is trained on k-1 folds and validated on the remaining fold. This process is repeated k times, each time with a different fold used for validation. The model's performance is then averaged over these k iterations.

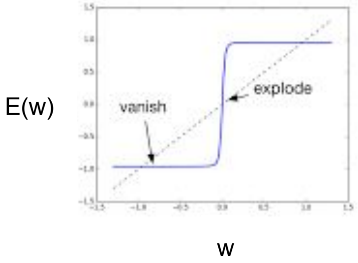


medium.com

Regardless of the termination criteria used, the NN is not guaranteed to arrive at the global minimum for the SSE as it may become stuck in a local minimum which still represents a good, if not optimal solution

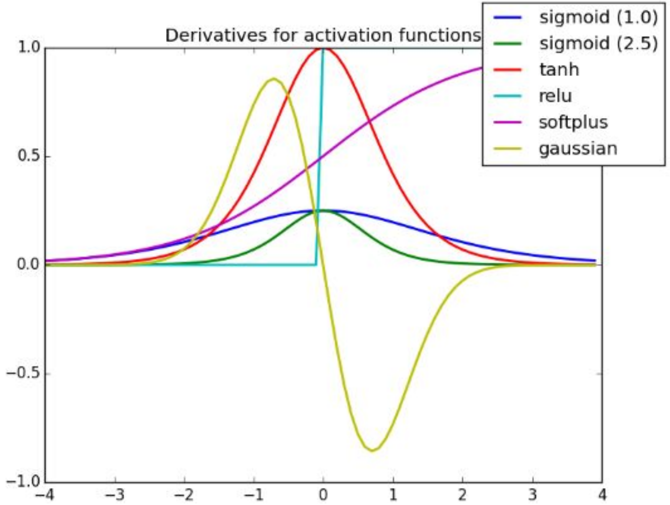
Vanishing Gradient Problem

- Vanishing Gradient Problem can occur during the training of NN when the gradient of the loss function wrt the weights in the lower layers of the network become very small
- Small gradient do not contribute much to the weight updates during training. As a result, it can slow or even halt learning those layers as the weights are not being updated effectively
- The problem is a result of the multiplicative effect of small derivatives of activation functions in deep networks



Examples of Activation Functions and Their Impact

- **Sigmoid Function**: It squashes its input to a range between 0 and 1. Its derivative is maximal at 0.25 and decreases toward 0 as the input moves away from 0. In deep networks, multiplying these small values can quickly lead to vanishing gradients.
- **Tanh Function**: Similar to sigmoid, but squashes input to a range between -1 and 1. It suffers from the same problem as sigmoid for high absolute values of input.
- **ReLU (Rectified Linear Unit)**: Introduced as a solution to the vanishing gradient problem. It does not saturate in the positive input range and has a derivative of either 0 (for negative inputs) or 1 (for positive inputs). However, ReLU can lead to another issue called the "dying ReLU problem," where neurons only output negative values and thus have a derivative of zero, effectively "dying."



Wrapping up

- Once all the weights are updated, we complete one round of backpropagation
  - One forward pass followed by a backward pass are counted as one full pass
- With the updated weights, we then obtain the predicted output for the next data point, in another forward pass and compute the prediction error
- We repeat this until termination/stop criteria is reached which is termed that the model has converged