EECS 498-007 Deep Learning for Computer Vision

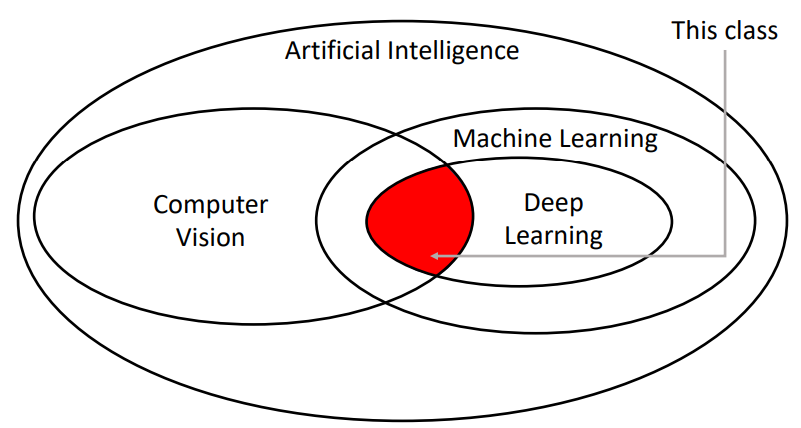
# Lecture 1 – Introduction

Computer vision: building artificial systems that process, perceive, and reason about vision data.

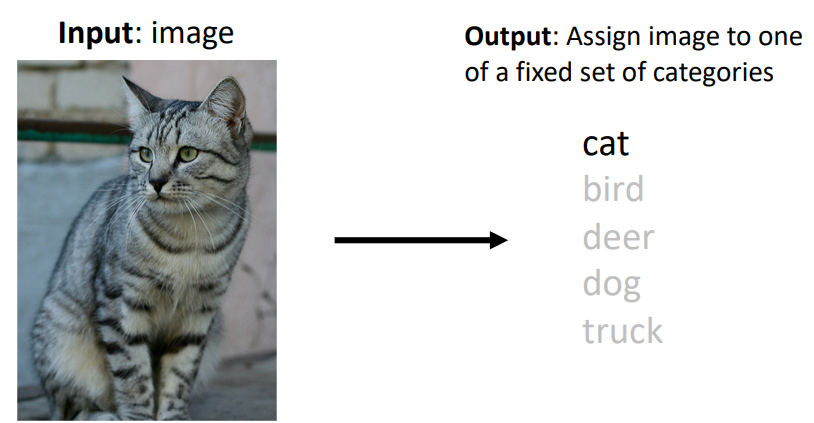
Learning: building artificial systems that learn from data and experience.

Deep Learning: hierarchical learning algorithms with many “layers”, (very) loosely inspired by the brain.

AI: a set of technologies that allow computers to perform tasks that typically required human intelligence.

Not all AI is based on machine learning, “symbolic AI” or “Good Old-Fashioned AI (GOFAI)” mainly relies on explicitly programmed rules and logic to behave like a human.

# Lecture 2 – Image Classification

Image classification is a core computer vision task, where the goal is to assign a label from a fixed set of categories to an input image. An RGB colour input image is simply some 3D array of red, green, and blue intensity values.

## Challenges

Whilst image classification is relatively trivial for people, there are many challenges involved with developing an image classification algorithm, including but not limited to:

* Viewpoint variation: Even a slight shift in the camera’s position will change all the pixel values in the image.
* Fine-grained categories: e.g. There are many different breeds of cats with different appearances.
* Intraclass variation: e.g. cats of the same breed still look different to one another due to their unique markings.
* Background clutter: Objects may blend into their background, making them hard to identify.
* Illumination conditions: The exact same scene with different lighting conditions produces a completely different image array.
* Deformation: Many objects are not rigid bodies and can be deformed in extreme ways.
* Occlusion: Sometimes only a portion of the object is visible.
* Scale variation: Objects exist as different sizes both in the real-world and in terms of their extent in the image.

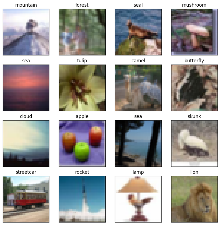
## Data-driven approach

Whilst it may technically be possible to write an algorithm that encodes some of the techniques humans use to identify cats for example (e.g. extract edges, then search edges for pointy ears, whiskers, etc.), this approach is brittle and not easily adapted to classify different classes of images. Instead, we can use a **data-driven approach**, where we:

1. Collect a dataset of images and labels
2. Use machine learning to train a classifier
3. Evaluate the classifier on new images

### *Collect a dataset of images and labels*

Some commonly used image classification datasets include:

* MNIST – a dataset of handwritten digits
  + Classes: 10, digits 0 to 9
  + Image size: 28x28 grayscale
  + Num. training: 50k
  + Num. test: 10k
  + Sometimes called the “drosophila (fruit fly) of computer vision” due to it being a simple baseline. **Results from MNIST often do not hold for more complex datasets.**
* CIFAR10
  + Classes: 10, e.g. airplane, automobile, cat, deer, dog
  + Image size: 32x32 RGB
  + Num. training: 50k
  + Num. test: 10k
* CIFAR100
  + Classes: 100, with 20 “superclasses” e.g. aquatic mammals, trees, etc.
  + Image size: 32x32 RGB
  + Num. training: 50k
  + Num. test: 10k
* ImageNet
  + Classes: 1000
  + Image size: Variable, often resized to 256x256 for training
  + Num. training: ~1.3M
  + Num. validation: 50k
  + Num. test: 100k
  + Due to the large number of classes, the **Top 5 accuracy** is often used as the performance metric, where the algorithm predicts 5 labels for each image and one of them needs to be right
  + Often still used as the standard for benchmarking a classification model, however, it can be expensive to train on.
* MIT Places
  + Classes: 365, scenes like “bedroom”, “bar”, “rainforest”
  + Image size: Variable, often resized to 256x256 for training
  + Num. training: ~8M
  + Num. validation: 18.25k
  + Num. test: 328.5k
  + Like ImageNet but with scenes rather than objects
* Omniglot
  + Classes: 1623, characters from 50 different alphabets
  + Image size: 105x105
  + Num.: **Only 20 images per category**
  + Used to test few-shot learning

### *Use machine learning to train a classifier – E.g.* ***Nearest Neighbour Classifier***

The Nearest Neighbour Classifier simply takes a test image and compares it to every single one of the training images, and returns the label of the closest training image. This makes training very quick (O(1) for simply copying pointers) and testing very slow (O(n) to compare against every image in the training set). This is **bad**.

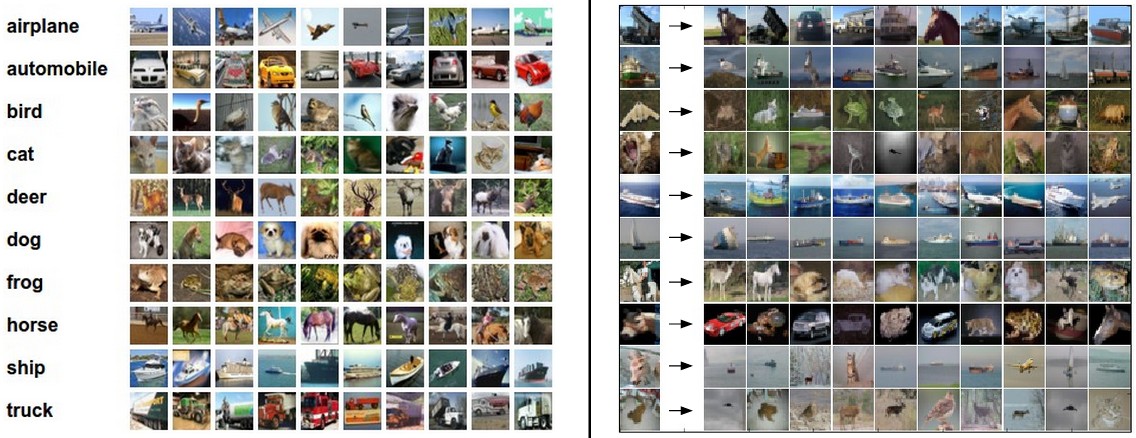
There are many methods for fast / approximate nearest neighbours, [example](https://github.com/facebookresearch/faiss).

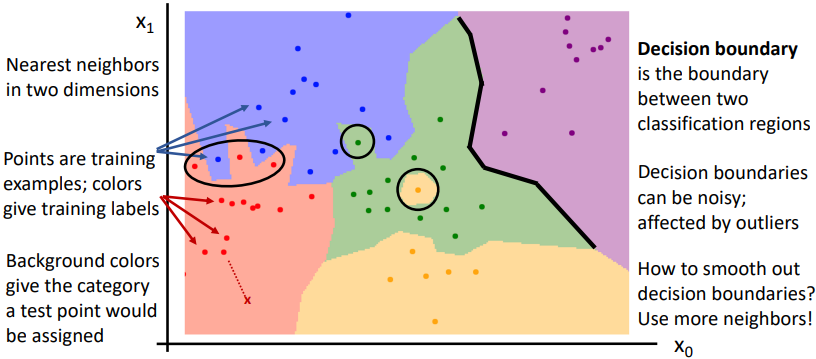
We can use different distance formulas to compare two different images, to provide different results. For example, the L1 distance is:

Where the sum is taken over all of the pixels .

A grid of numbers and equal to the number

Description automatically generated with medium confidenceVisually, the L1 distance procedure is:

An example is given below for test images in the left-hand column, the top 10 nearest neighbours are returned. Notice that while the nearest neighbours are visually similar, they’re not the correct class most of the time.

If you plot the regions which are closest to the training examples and then colour them according to their training label, you can visualize the **decision boundaries**.

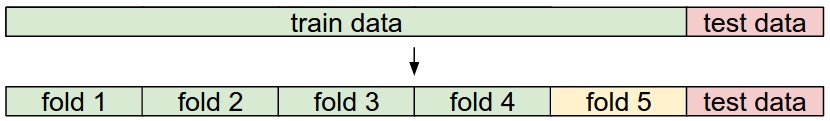
Instead of finding the nearest single neighbour, you can take a majority vote from the K closest points. For example, K=3 means you find the closest 3 images in the training set to the test image and then take the majority vote to classify the test image. This is referred to generally as **k Nearest Neighbour** (**kNN**). Using more neighbours helps to smooth decision boundaries and reduces the effect of outliers.

The **L2 (Euclidean) distance** is another choice for the distance calculation:

#### Hyperparameters – What is the best K value, what distance metric do I use?

Selecting a K value and the distance metric for the Nearest Neighbour Classifier Hyperparameters are very problem dependent. Outlined below is the common train-validation-test method for choosing hyperparameters and then evaluating the model:

1. The model is trained on the training set
2. Its performance is tested on the validation set
3. Steps 1 and 2 are repeated with differing hyperparameters
4. The model that performs the best on the validation set is then tested on the test set. No changes should be made based on the performance on the test set (to avoid contamination of the data).

**K-fold cross validation** extends this idea by first separating the test set, and then splitting the remaining data into training and validation sets in k different ways. The steps are then the same as before, except the model’s performance is the average of its performance on each validation fold.

#### The Problem with the Nearest Neighbour Classifier

As the number of training samples goes to infinity, the Nearest Neighbour Classifier can represent any continuous function! (If the number of training samples was infinite, any test sample would also be in the training set). *However, the number of training samples that you need for a dense coverage of the sample space increase exponentially with dimension*. For example, the number of possible 32x32 binary images is:

Clearly, having enough training samples to give good results for even very small binary images is impossible.

Therefore, Nearest Neighbour Classifiers are seldom used on raw pixels. **They do however work well on ConvNet features**.

# Lecture 3 – Linear Classifiers

Many machine learning models take a parametric approach to classifying images by defining some function that takes in the **image** and a matrix of **parameters** to output a vector of **scores** that represent the classifier’s prediction for each class. Arguably the most simple function, as used in a **linear classifier**, is a linear mapping:

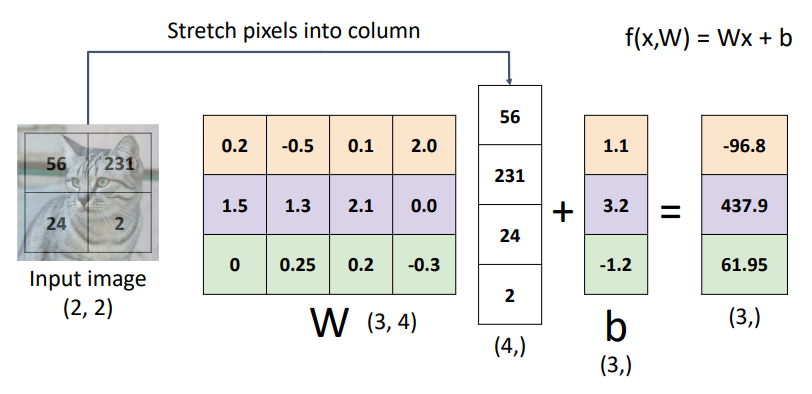
Where, using the CIFAR10 dataset with 32x32x3 sized images as an example:

* outputs a vector equal in size to the number of classes
* The weights matrix therefore, has size
* And the **bias** vector has size

There are three common different viewpoints we can use to view the linear classifier:

* The algebraic viewpoint
* The visual viewpoint
* And the geometric viewpoint

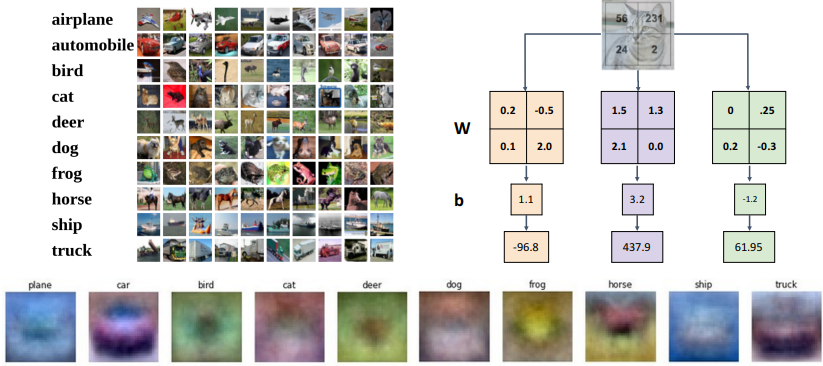
## The Algebraic Viewpoint

Using a simple grayscale image to be classified as one of three classes as an example. The weights matrix has size and the bias vector has size . We can see below that each row of and are effectively their own classifier in parallel, one for each class.

A screenshot of a graph

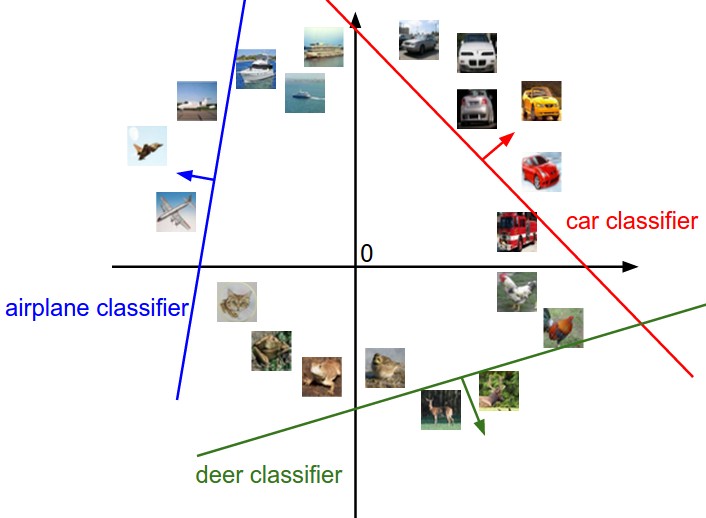
AI-generated content may be incorrect.Note: a common trick known as the **bias trick** can be used to combine the bias terms into the weights matrix as an additional column, as long as the input vectors are appended with a , as shown below:

## The Visual Viewpoint

Instead of stretching the pixels of the image into a column, we can imagine reshaping each row of the weights matrix into an image! After training the linear classifier, each of these rows is essentially a learned “template” to be used for matching against its corresponding class. You can actually plot each row of the weights matrix as an image to visualize what the learned “templates” look like, as seen at the bottom of the image below.

In this way, a linear classifier can be thought of in a similar manner to a kNN Classifier, except the test image is compared to only a single “template” learned from the dataset instead of every dataset image directly, and the negative of the inner product is used as the distance instead of the L1 or L2 distance.

## The Geometric Viewpoint

Since CIFAR10 images are represented by 3072 values, we can image each image in our dataset as a point in a 3072-dimensional space. Each class score of a linear classifier is then a linear function over this space, providing some score for each possible 3072-dimensional image.

The above is an imagining of this 3072-dimensional space as a 2D plot. The red line shows all of the points in the plot which get a score of zero for the car class, whilst the red arrow shows the direction of increase, so all points to the right have linearly increasing scores.

As we saw above, every row of  is a classifier for its corresponding class. The geometric interpretation of these rows is that as we change them, the corresponding line in the pixel space will rotate. The biases , on the other hand, allow our classifiers to translate the lines. In particular, if we do not include any bias terms, plugging in  would always give a score of zero regardless of the weights, so all lines would be forced to cross the origin.

## Loss Functions

So far, we have described the weights matrix and the bias vector , but how do we quantify how good our and are for our classification task? A loss function (also called an objective function or cost function) tells us how good our current classifier is. A high loss indicates poor performance.

For a single sample in our dataset where is the image and is the known label, the loss is some function :

And the loss for the dataset is the average of the per-example losses:

### Cross-Entropy Loss

Cross-entropy (CE) loss is used in multinomial (multiple classes) logistic (i.e. classification) regression when we want to interpret raw classifier scores as probabilities. We use the **softmax** function to take our output score vector of arbitrary real-valued numbers and transform them to an output vector of values between 0 and 1 that sum to 1. These output values can be interpreted as probabilities, representing the classifier’s confidence in each class being the correct classification.

For the output scores of a linear classifier , softmax for class is defined as:

The **cross-entropy loss** is then defined as the negative log of the softmax’s output for the correct class, i.e;

* What is the minimum and maximum possible loss ?
  + Minimum: when
  + Maximum as
* If all scores are small random values upon initialization, what loss can we expect?
  + We expect the softmax output probabilities to be approximately equal. Therefore, for number of classes:
  + For 10 classes in the CIFAR10 dataset, we can expect the initial CE loss to be

### Multiclass Support Vector Machine (SVM) Loss

SVM loss “wants” the correct class for each image to have a score higher than the incorrect classes by some fixed margin . The multiclass SVM loss for the -th image in the dataset is defined as:

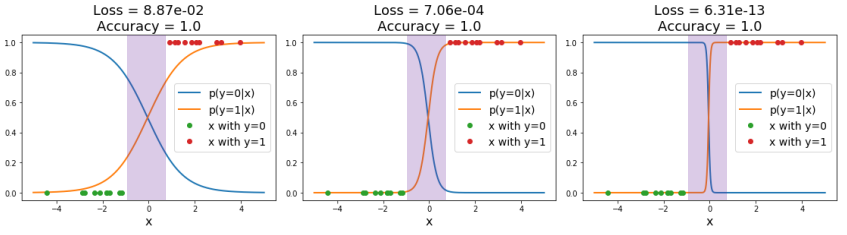
A graph with words and a line

AI-generated content may be incorrect.This type of loss function is also referred to as **hinge loss** and can be visualized as:

* What is the minimum and maximum possible loss ?
  + Minimum: when
  + Maximum
* If all scores are small random values upon initialization, what loss can we expect?
  + We expect the scores to also be random, so the differences between them should be approximately zero. Therefore, for number of classes, we sum lots of :

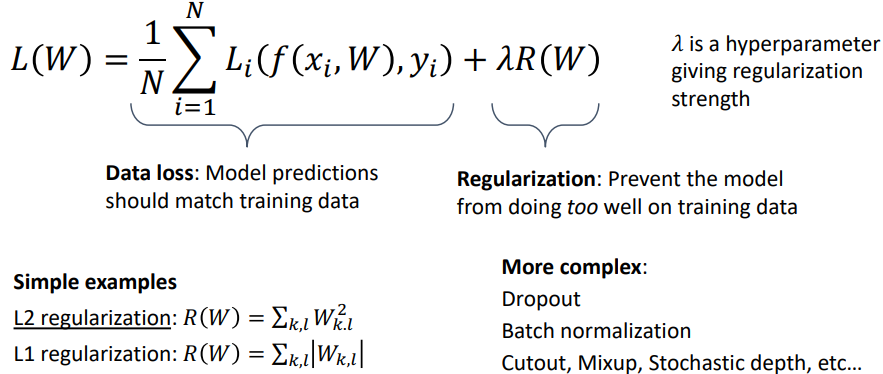
## Regularisation

### Motivation

Loss functions encourage good performance on *training data* but ultimately, we really care about *test data*. We say a model is **overfit** when it performs too well on the training data and has poor performance on unseen data. For example, the below three plots show the same training data in green and red, and the models’ predicted scores for the two classes in orange and blue respectively. The rightmost model has the lowest loss, but it is way too confident in the purple region with no training data, which can lead to poor generalisation on unseen data.

Additionally, if using SVM loss it is possible to have many different weights matrices and bias vectors which give the exact same loss. How do we differentiate between these models?

### Implementation

We can add an additional term to our loss function to express preferences about the weights and to prevent our model from doing *too* well on the training data:

#### Expressing Preferences

Regularisation allows us to penalise model behaviours that we consider undesirable. For example:

* If we have the input
* and two different weights matrices and
* both models give the same prediction: , so data loss will always be the same.
* However, model 1 only uses the first input dimension and completely disregards the other dimensions.
* If we use L2 regularisation, which sums the square of each weight and adds this to the loss while training, we can encourage the weights to be smaller and more spread out.

#### Preventing overfitting

Penalising large weights also tends to improve generalisation, as no input dimension can have a disproportionately large influence on the scores.

A graph of a graph of accuracy

AI-generated content may be incorrect.For example, using L2 regularisation on the previous classification example causes loss to actually *increase* for the overfit model.

Note that biases do not have the same effect since, unlike the weights, they do not control the strength of influence of an input dimension. Therefore, it is common to only regularize the weights  but not the biases . However, in practice this often turns out to have a negligible effect.

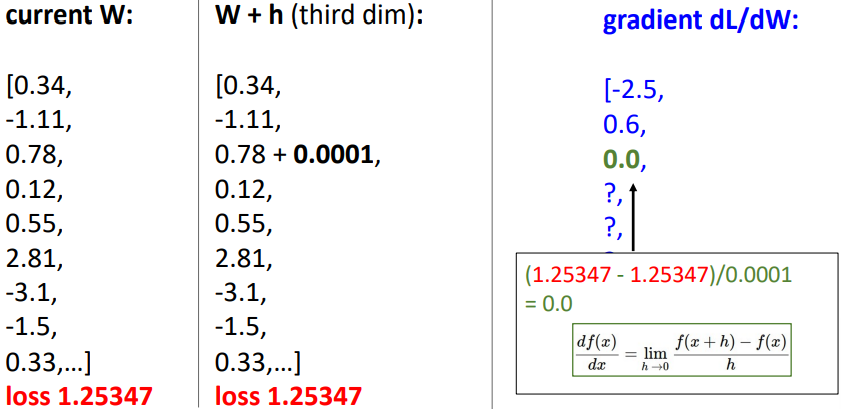
# Lecture 4 – Optimisation

Very generally, the process of optimisation usually refers to finding the weights that minimize the loss function , i.e. the optimal weights are:

And a common way to visualize the search for these optimal weights is traversing a high dimensional landscape to find the lowest valley. We are doing this with no sight, i.e. no way to determine from afar, but we can feel around our local position for information.

By feeling around our local position, we can determine the direction of steepest decent and then traverse down that direction and then repeat for each step we take. Functionally this is equivalent to calculating the derivative of the loss function with respect to the weights , and then moving in the negative direction of the gradient.

## Numeric Gradient

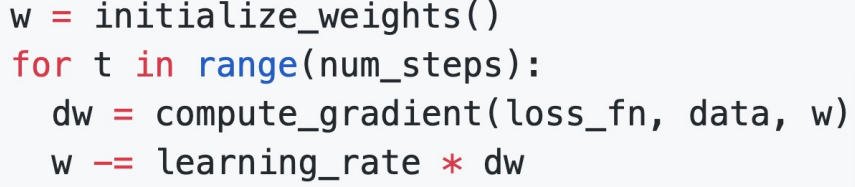
A naïve approach to calculating is to perturb each element in by a small amount and then determine the change in loss, then use rise-over-run to approximate the derivate for that element of , repeating for all elements in . This approach is **slow** (O(n-dimensions)) and **approximate**.

## Analytic Gradient

Alternatively, we can use calculus to actually compute the analytic gradient based on the model and the formula of our selected loss function and regularisation technique. This method is **exact**, **fast**, but can be **error-prone** due to needing to calculate the derivatives analytically.

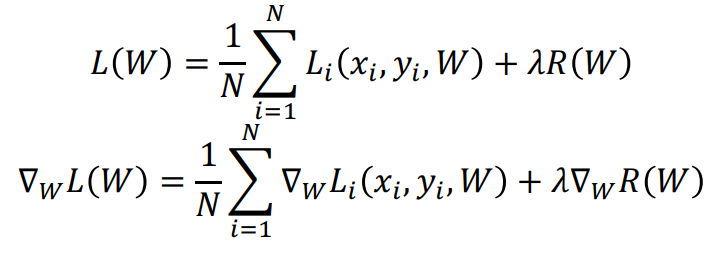
In practice, we should always use analytic gradient, but we should check our implementation using the numerical gradient. This is called a **gradient check**.

## Gradient Descent

The method described above of “feeling around our local position” and then iteratively stepping in the direction of steepest descent (i.e. the negative gradient) is known as **gradient descent**. The basic vanilla function looks like:

Where the weight initialisation method, number of steps, and learning rate are all hyperparameters.

### Batch Gradient Descent (or Full Batch Gradient Descent)

Recall that our loss at each step is the average of the loss of the model on the entire training set. Then the gradient of the loss with respect to the weights is the average of the gradients on the entire training set. That is:

This can be very expensive and impractical when the training set is very large. Do we need to calculate the gradient on every single training data point?

### Minibatch Gradient Descent / Stochastic Gradient Descent

Stochastic Gradient Descent approximates the gradient at each step by using a **minibatch** of examples, commonly 32, 64, 128, or 256 (technically SGD only refers to the case where the minibatch is a single example, however people now commonly use it to refer to minibatch A close-up of a code

AI-generated content may be incorrect.gradient descent). The basic function looks like:

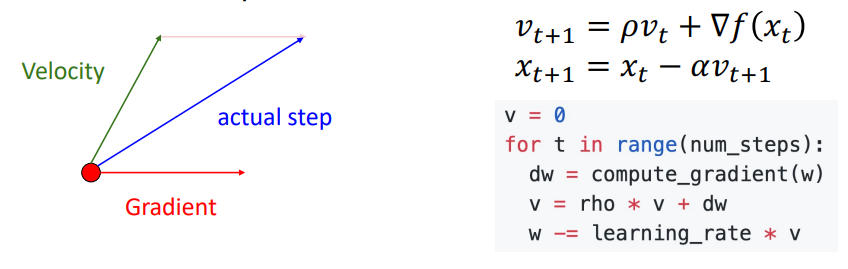
Where the batch size and sampling method are additional hyperparameters.

Training is generally not super sensitive to batch size, and in practice you generally make the batch size as big as you can fit on the GPU(s).

There are some problems with the above implementation of SGD however:

* If the loss changes quickly in one dimension and slowly in another, there is unlikely to be an optimum step size which doesn’t cause very slow progress along the shallow dimension and oscillations along the steep direction. Note: this is usually indicated by a high **condition number**, the ratio of largest to smallest singular value of the Hessian matrix.
* The SGD algorithm can get stuck at points where the gradient is zero, at:
  + Local minimums, where the loss function curve curves upwards in all directions. These are rare in very high dimensional space.
  + Saddle points, where the loss function curve curves upwards in some directions and downwards in others. Much more common in very high dimensional space.
* The gradients are approximations from minibatches so they can be noisy

#### Stochastic Gradient Descent + Momentum

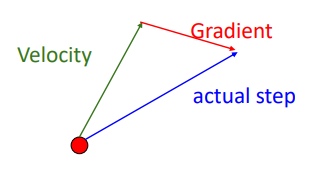
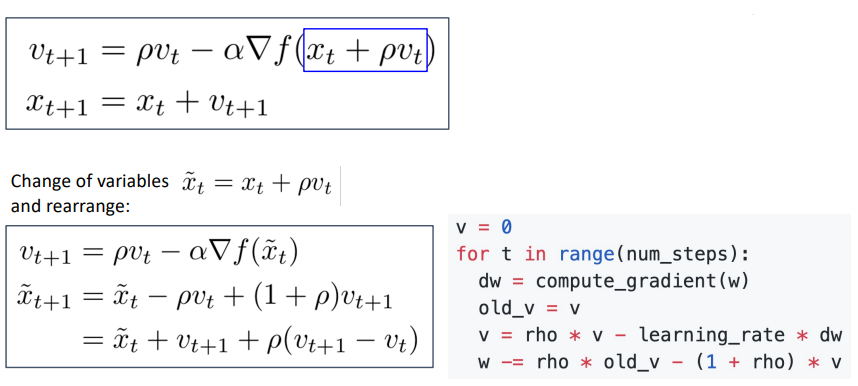
Momentum is the idea of “building up a velocity” as the running mean of previous gradients. The step used to update the weights then comes by combining the gradient at the current step with the velocity. This gives similar behaviour to a marble rolling down a hill. The velocity is usually reduced by some “friction” value (typically 0.9 or 0.99). The formula for SGD+Momentum is:

SGD+Momentum addresses some of the problems with SGD outlined above:

* The velocity acts to smooth out oscillations by acting as a moving average.
* The velocity helps to carry the weights over local minima and saddle points, thanks to the momentum.
* The velocity acts to smooth out the gradient noise that comes from using minibatches.

#### Nesterov Momentum

Nesterov momentum uses a slightly different method to calculate the “actual step” from the gradient at the current point, and the velocity. The gradient is computed at the point where updating using only the velocity would take us, and it is then mixed with the velocity to give the actual update direction.



#### AdaGrad

AdaGrad is not a momentum-based optimisation algorithm like SGD+Momentum and Nesterov Momentum. Rather than tracking the historical sum of gradient values, we keep track of the historical sum of squared gradient values. Then, when updating our weights matrix, we divide the current gradient by the square root of the historical squared gradient values. This effectively gives an adaptive learning rate (AdaGrad = adaptive gradient):

This technique addresses the problem of having large gradients in one direction and small gradients in another direction. Large gradients divide through to cause small step sizes, and small gradients divide through to cause large step sizes. Progress along “steep” directions is damped, progress along “flat” directions is accelerated.

As the historical sum of squared gradients keeps accumulating (due to the squared gradients always being positive), learning can slow dramatically over time as you divide the step by larger and larger numbers.

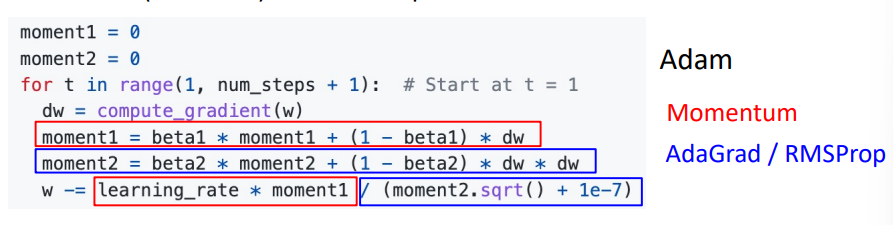
#### RMSProp: “Leaky AdaGrad”

A close-up of a computer screen

AI-generated content may be incorrect.To fix the problem of AdaGrad halting over time due to the squared gradients becoming too large, RMSProp (AKA. Leaky AdaGrad) adds a decay rate term to not accumulate squared gradients so fast.

#### Adam): RMSProp + Momentum + bias correction

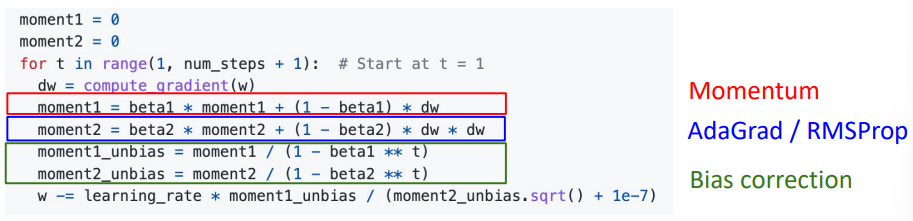
Adam (adaptive moment estimation) is a very common optimisation algorithm that effectively combines the idea of momentum and adaptive learning rate from AdaGrad/RMSProp. In statistics, the first moment of a distribution is simply the mean, and the second moment is the variance.

Combing RMSProp and Momentum gives:

Where:

* The first moment​ is the running average of the gradients. It smooths out the fluctuations in the gradient and captures the general direction in which the gradient is pointing over time.
* The second moment is the running average of the squared gradients. This moment captures how large or small the gradients are over time, helping the optimizer adjust the step size (learning rate) based on how "volatile" the gradients are.

The above form has the problem that the first and second moment estimates start at zero and and are generally , so the very first step size can be very large (if the second moment is near zero after the first update, you divide by the square root of near zero, taking a very large step). Essentially, the estimates for the first and second moments are biased towards zero at the start of training.

A **bias correction** is used to counter this. Giving the full form of:

#### AdamW: Decoupled Weight Decay

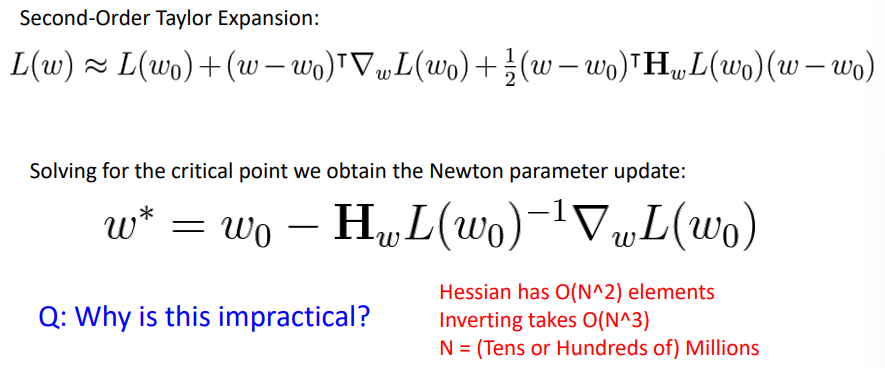
Recalling L2 regularisation from the previous lecture, in practice instead of adding a term to the loss function, it is much more common to add a term to the gradient. The term **weight decay** refers to adding a term (where is the learning rate) to the update step instead. In some optimisers, L2 regularisation and weight decay are equivalent (SGD, SGD+Momentum), so weight decay is often implemented by adding L2 regularisation. However, **L2 regularisation and weight decay** **are not the same for adaptive methods** (AdaGrad, RMSProp, Adam, etc.).

AdamW is the Adam optimiser with the weight decay term “decoupled”, i.e. added to the update step instead of to the gradients. **“AdamW should probably be your *default* optimiser for new problems”** – Justin Johnson. SGD+Momentum can outperform Adam but may require more tuning.

#### Comparison of Optimisation Algorithms

### Second Order Optimisation

The above optimisation algorithms are all first order algorithms, they only use the first derivation to approximate the loss surface (e.g. a line in 2D space, hyperplane in high dimensional space). Second order optimisers use the gradient and the Hessian matrix (the matrix of second order partial derivatives) to approximate the loss function surface. This takes into account the loss’ curvature, taking bigger steps in regions of low curvature.

Second order Newtonian algorithms require inverting the Hessian matrix which is very expensive.

In practice, approximate methods such as BFGS are used instead (calculates the approximate inverse Hessian with rank 1 updates over time, each). **L-BFGS**, limited memory BFGS, does not form/store the full inverse Hessian.

L-BFGS works very well in full batch, deterministic mode. i.e. if you have a single, deterministic then L-BFGS will probably work very nicely. However, it does not transfer very well to mini-batch setting. Adapting second-order methods to large-scale, stochastic setting is an active area of research. If you can afford to do full batch updates then try out L-BFGS (and don’t forget to disable all sources of noise)