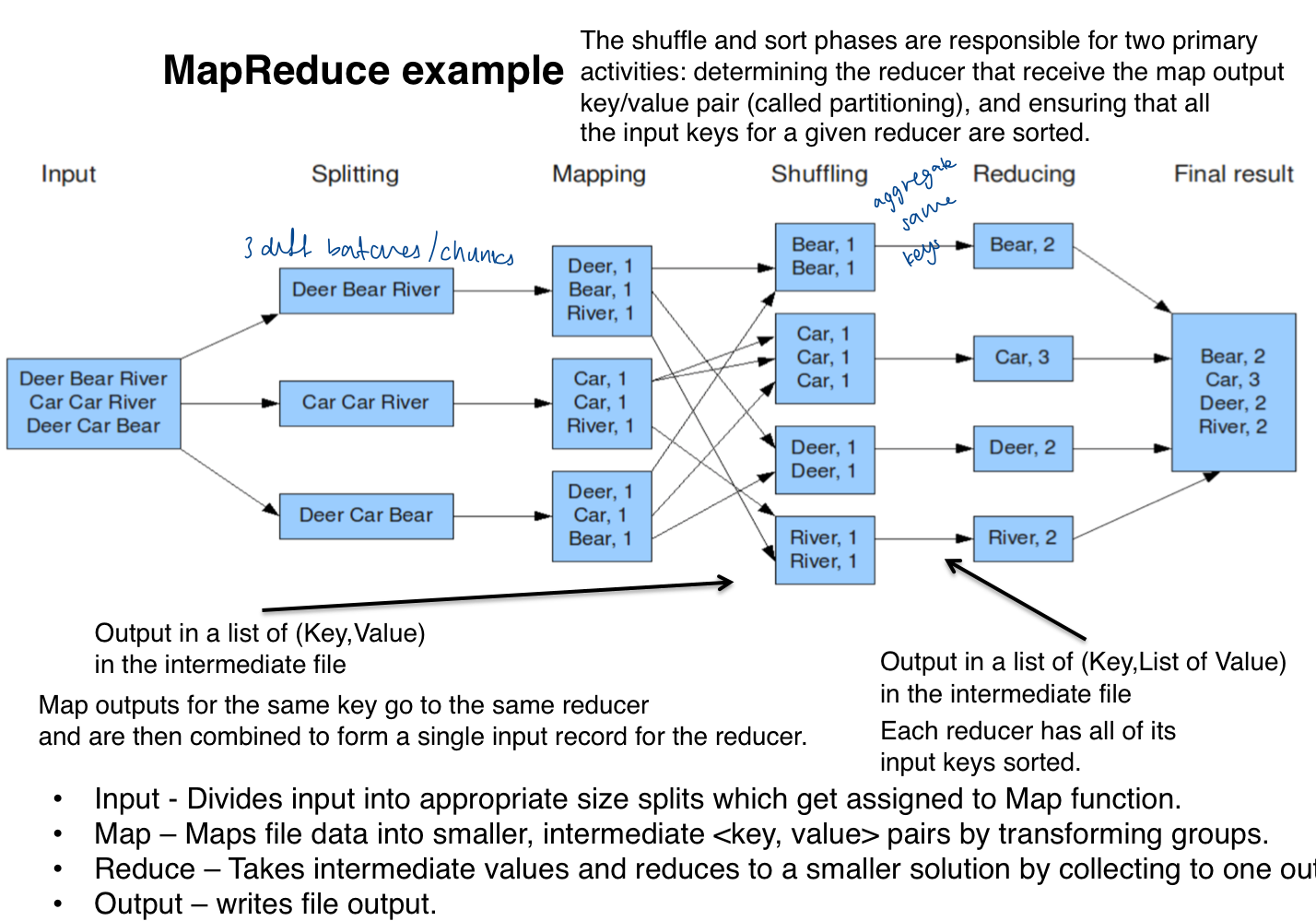
**Hadoop (Dist. Storage & Processing)**

*HDFS (Namenode, DataNode):* Data split into blocks of 64/128MB, each stored and replicated on multiple datanodes. Block locations do not persist in namenode (only in memory for quick lookup), but knows datanodes on which blocks for a given file are located.

*MapReduce (JobTracker Node, TaskTracker Node*): Get descriptive stats/counts. Master splits data into multiple map-reduce job parts. Task tracker manages map and reduce task for each part. Map takes kv pair from HDFS, converts to intermediate pair thru shuffle and sorting data across servers acc to key, reducer gets counts of each key from mappers and arrives at final result.



Hadoop VS MPP

*HDFS* - No Schema: Works better for unstructured data, high latency retrieval; Fault tolerance/High Availability: Block Replication; Reduced seeking time -> Increased transfer rate corresponding to disk bandwidth; No need to read from multiple disks concurrently to reduce time (risks hardware failure + data loss); Simpified storage sub system (unit of abstraction is block not file)

*MapReduce:* Fast indexing (parallel processing), Fault tolerance: If task fails, tasktracker notifies jobtracker to reschedule task. If datanode fails, namenode replicates data to another node and job tracker reschedules tasks on failed node. If jobtracker/namenode fails, whole system fails.

**Spark**

*RDD Processing:* Distributed processing on a cluster of nodes. Obtain data from HDFS, distributes each chunk to multiple nodes’ RAM (RDD) for parallel processing. Cluster manager allocates resources across worker nodes, acquire executors to run computations and store data, then sends tasks for executors to run on cache, results assembled back.

Spark VS Hadoop

*Batch Processing VS Batch/Interactive/Real Time Data Streams*; *Higher Fault Tolerance Efficiency*: RDD Resilient – Only remembers lineage of deterministic ops/file path of data so can rebuild lost data and assign to another node, but Hadoop replicates data to multiple machines so it can be restored from healthy replicas. RDD Immutability – Driver can recreate lost data from failed nodes and assign processing task to another node in case of worker failure (in-memory loss); *Speed 100x faster*: In memory processing/RAM – Reduces need for disk I/O, no need to read/write from HDFS/disk like Hadoop (60-90% of process). Reduces communication proccesses like replication/serialisation with hardidsk. Able to handle interactive/adhoc queries. Efficient Scheduling.; *Horizontal Scalability*: Large datasets dist. across multiple nodes.

**ML VS Deep Learning**

Frame, Symbol Grounding, Feature engin Problem; Does not require structured data;No human intervention for mistakes; CPU VS GPU; Time; Uses

**Neural Networks**

Goal: Minimise loss function by adjusting weights/biases during training.

Input: un = Wn xn +bn; Output: zn=f(un) where f is activation fn

Num Parameters: (Input dim +1)\*Num units in l1 + (Num units in l1 +1)\*Num units in l2

Method: Feedforward; Backpropagation (Compute errors and use gradient descent mthd to adjust errors and change weights – use partial derivative of E wrt w to find min/convergence). wt+1=wt - Learning Rate \* Gradient, where learning rate is how much we adjust weight wrt loss gradient fn (low: slow training, fast: converge too quickly at suboptimal solution)

Activation Fns: sigmoid(0-1, binary); softmax (multiclass)

tanh (-1-1); relu(max(0,x)); LeakyReLu(max(0.2x,x)) - A variant of the traditional ReLU activation function.It allows a small non-zero gradient for negative inputs, typically a small fraction like 0.1, to address the issue of dying ReLU in a state of zero activation, leading to degraded performance. \*\* f(x) = x if x > 0, and f(x) = ax if x <= 0; Linear (e.g.1.5x): Perf badly in backpropagation as output of gradient descent is constant -> Lack accuracy. Also equivalent to performing a linear regression.

Loss Fns: Binary: binary\_crossentropy; Multiclass: categorical\_crossentropy; Regression: mean\_squared\_error (root to ensure mangitude of error is similar to actual values. Allow it to tune weights during training to reduce error of predictions. Penalizes outliers heavily), MAE (handles outliers better), keras.losses.Huber() (best of both, not that sensi to outliers, but quad function changes to linear if loss is too much)

\*\* MSE function for classification problem is not convex = Cannot do gradient descent to find optimal weights. Does not strongly penalise misclassifications even for perfect mismatch (only ranges 0 to 1).

Metrics: Classification: [‘acc’]; Regression:[‘mae’/’mse’]

Optimizers: RMSprop/adam/SGD

Output Fns: Output units: Regression: 1, NA; Binary - outputs P(Belongs to Class 1) : 1, sigmoid; Multiclass - P(Belongs to Each Class): Num classes, softmax.

Main Issue: Data Loss during feedforward;Timing/Accuracy during backprop

Underfitting, Vanishing Gradient (Sigmoid saturates at 0 and 1, tanh saturates at -1 and 1): Produces small gradients when inputs to fn is large positive/negative values. More hidden layers = in backprop, gradient calc. recursively from output to input layer, derivatives in each layer get multiplied tgt to give even smaller derivatives by chain rule = gradient of loss fn wrt weights of the earlier layers decreases to 0 exponentially = hard to train cos cannot get optimal convergence => Use RELU (max(0,x)) which gives max derivative of 1 when values are positive, no matter how many times you multiply you still get 1 (not 0). Choose based on domain of input/range of output. Slow Speed (due to gradient descent, randomness of finding the best route): Huge input data = Slow gradient desecent => Use stochastic/mini batch gradient descent; Overfitting (due to complexity, or small training data wrt num of model param): NN trained too well on training data, becomes too specific, cannot generalize to new unseen data, perf well on train, bad on test Underfitting (due to low complexity, not trained for long enough): Cannot capture underlying patterns in data, perf. bad on train & test => Choose model complexity properly

Soln - Data: Structured data (missing value - .fillna(df[‘col’].mean()), normalization - Standard Scaler: Transform numeric variables so each variable has 0 mean and unit variance. Transform diff magnitudes to same scale. Leads to faster convergence during backprop and better perf); Text data(preprocessing, vectorization, word embedding (doc2vec, word2vec)); Image data (normalization of size, augmentation); Video data (opencv fn)

Soln – NN: Diversifiying network structure (# hidden layers, nodes (512,128,64,32,16), # epochs); Dropout: Randomly select units with prob p to drop. Aggregate multiple samples to get avg value through normalization > Reduce high deg of freedom + Increase randomness in model; Change activation fn; Tune parameters (optimizers, learning rate), change/combine models (CNN +LSTM)

Soln – Others: Increase accuracy from image augmentation by incl. different shapes of input objects to generate noise data; Increase accuracy from dropout by preventing co-adapting neurons information (increase randomness); Increase noise and randomness as distribution might not be fully captured/not representative actual distribution.

**CNN** (Image classification, Object Detection, Instance Segmentation)

In>Convolution (relu – faster)<>MaxPooling>FCNN(relu)>Out(sigmoid/softmax)

Conv/Pool to identify spatial features, FCNN to make pred.

Pixels (intensity, 0-255), Channels (1 for greyscale, 3 for RGB), Shape (Characteristics)

Problem: Overfitting cant effectively capture feature of image if size, angle,shape diff

Convolution: Goal: Applying several filters (kernel) to image to extract diverse features throughout the entire image/maximise feature of img and calculate the filtered values (feature map).How: Slide over image spatially, over full depth, to compute dot products. Provdes map to areas where characteristic feature is found.

Adv: Reduces model complexity, dynamic object recognition.

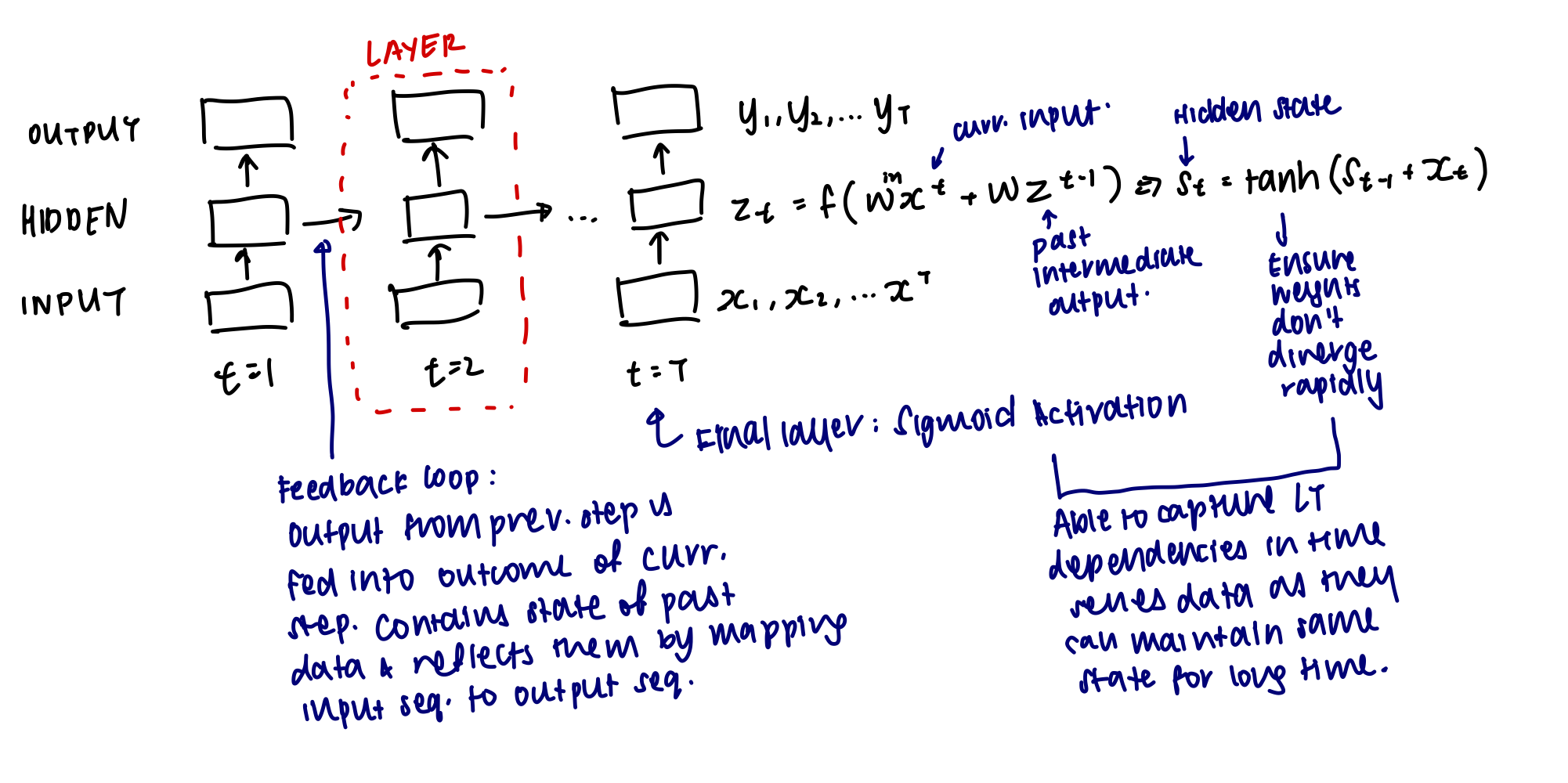
Soln - Pooling: Goal: Enables network to be translation invariant (generalise and recognise features indepenent of location in image, by indicating only the presence of the feature). How: Max (More attentive to prominent edges/features), Average: Smoother. Adv: More adaptive to change in viewpoints, angle and size; Reduce resolution = Reduce num param = Less overfitting. Disadv: Hard to make intermediate outputs have same size after filtering; hard to compute CNN outputs on hidden layer; Losing info on corner of images.

Padding: Goal: Make input and output have same size after filtering = output size easier for further computation.

Soln – Data Augmentation: Images have different dimensions, orientations, sizes, location of subject in image. Goal: Increase size, diversity, range and type of original data by introducing variations to original data by cropping, rotating, distortion, translation, flip, zoom = Decrease overfitting = Generalise better = Increase robustness. Avoid extreme transformations (distroted images don’t represent real world).

**RNN**

Goal: Targets sequential problems where order of features presented to model is impt for making predictions (e.g. NLP – Seq. captures info required to make predicitons, conveys context & subtle nuances/meanings, Time Series). How: Breaks input into time steps > Each chunk/word is passed to corresponding layer > Each layer passes its intermediate output (ie. Hidden state) to next layer > Allows RNN to maintain memory of intermediate states from sequential data.

****

**Spark RDD Operations**

*Transformation*: Operations on RDD that return new RDD;

map(x⇒x+2), filter(x⇒x==2), flatMap(x⇒x.split()) (1 element can give multiple outputs, eg split string into words), groupByKey(), reduceByKey(a,b⇒ a+b), reduceByKey(max) or reduceByKey(lambda a,b: a if a>b else b) or reduceByKey(lambda a,b: max(a,b)), aggregateByKey(), sortByKey(), flatMapValues(x⇒x.split()), values(), sortBy(lambda x: x[1], False) (sort by value, descending order), mapValues(x⇒x/10), .max() gives the max key value. rdd.distinct() ⇒ remove dup, rdd.union(rdd2), rdd.intersection(rdd2), rdd.subtract(rdd2), rdd.cartesian(rdd2) ⇒ all possible combinations of both RDD. rdd.join(rdd2) ⇒ joins with matching keys, values are in tuples. .startswith(’SB’). .lookup(key) => returns list of values for that key.

*Action*: Operations that return a result to driver or write to storage;reduce(lambda a, b ⇒ a+b), collect(), count(), saveAsTextFile(’results.txt’), countByKey()

*Lazy Evaluation:* Operations are executed on need-to-do basis – Efficient, saves RAM for processing (only final reduced output returned, not larger mapped dataset), allows recovery from lost data partitions.

*Dataframes/SQL:* Used for structured data/tables. Stores more info about data structure, query optimization, relational processing within native RDDs & external data soources, high perf using established DBMS techniques, supports new data sources(semistructured, external db),can use advanced analytics(ML,Graph processing). Results of sql queries are rdd, supports rdd ops.

**Spark Applications**

Data Stream/Real Time Processing*:* Data collected from beginning of time interval to end of time interval, grouped and processed in micro batch.

*VS Batch Processing*

Batch – Static data, series of non-interactive jobs executed by computer all at once. Manually upload data on system and issue execution commands; Stream – Data generated continuously by many data sources that simultaneously send data records in small sizes. Automatically updates analyzed resutls as new set of data is dynamically added. Data scope (Entire dataset vs data over rolling time window/most recent), Data size (large batch vs micro batch), performance (long vs short), analyses (complex vs simple response functions, aggregates, rolling metrics)

*Lambda Architecture*: Processing - Data fed to batch (manage immutable, raw master dataset (used to recover lost data from speed/serving layer) & precompute batch views) and speed layer (store real time views & process incoming data stream), serving layer (index batch/realtime views and aggregate results to query in low-latency, adhoc way); Re-processing – Every batch cycle; Object – Consistency + Low latency; Resource consumption – Query; Reliability – Batch reliable, streaming approx; User – Twitter, Liverperson

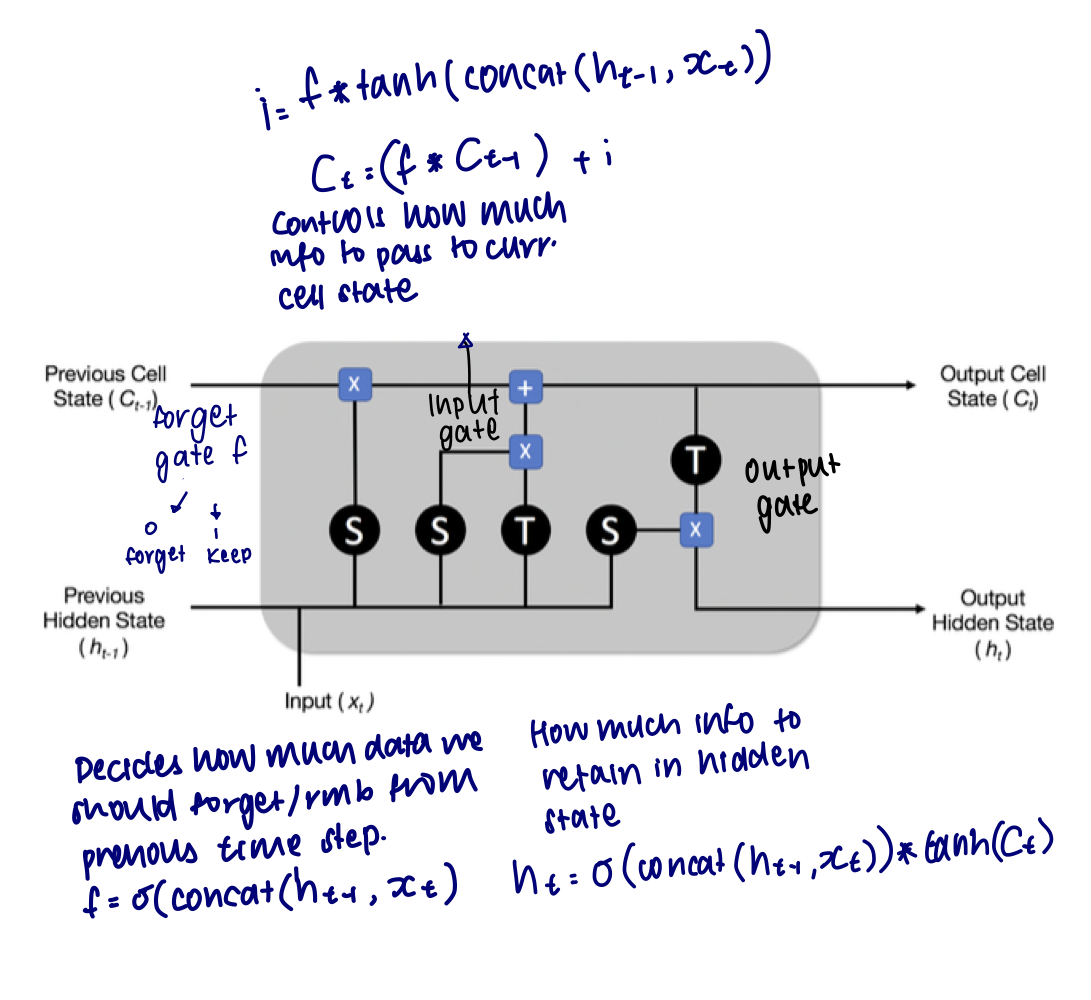
*Kappa Architecture:* Data fed to real-time/stream processing layer; only when stream processes modified; low complexity; Incremental algo; Streaming consistency; Linkedin, Yahoo

*Spark Streaming (Lambda):* Dstream: Input data streams sliced into small batches of time intervals before execution in parallel, each stored as RDD. Can be modified via transformations (creates new stream over sliding window of data) & output operations to external system. Can be recomputed since it was derived from deterministic tasks (map, reduce, groupby). .window(“5s”): Groups all records from sliding window into 1 RDD. .reduceByWindow(“5s”, (a,b) -> a+b, (a,b) -> a-b) (Association & Invertible): account for overlapping timestamps. .countByWindow(window, slide)

(For RNN) Vanishing Gradient Problem: Backpropagation: By the time loss is propagated back to first few layers, loss has diminished so much that the weights do not change much. RNN cannot learn from early time steps, has poor long term memory. Only deals with short-term dependencies, but not long term dependencies (significant gap from output time step).

**LSTM**

Conveys *important* information that has to be retained in its cell state, which flows from one repeating structure to the next. Hidden state is the overall memory of entire LSTM, having everything incl. *unimportant* inputs.



**CNN + LSTM**

CNN learns pattern of changes overtime at a macro level (feature extraction: learn features from both spatial and time dimensions), LSTM learns long term dependencies between time steps (interpret features across timesteps and predict). Good for sequence prediction problems w spatial inputs. How: Conv > Maxpooling>Flatten>LSTM>Dense Output Layer. Kernel size (windowing), zero padding(mark start and end of time), Stride (how much window can move).

**Language Modelling**

Word Embeddings: Learned form of vector representation of words

Adv: Fewer dimensions than OHE; Similar words are closer tgt, dissimilar words are far away => Captures semantic r/s in embedding space using cosine similarity (computes compatability of words in context – prob of using each word with another word).

Preprocessing: If each doc has different len, use keras.pad\_sequence() to fix size of vectors.

Language Modelling: Compute the probability of sequence of words/likelihood of a given word to follow a sequence of words.

Probability Distribution:

Ngram Model:

*Data sparsity:* N needs to be as large as possible to get good estimate of original probability distribution, more data needed to get good estimate of ngram probs.

Word2Vec: 2 Layer NN that processes text by generating feature vectors of words. Given a sequence of n-1 words, output the probability distribution of the next word w. We are interested in outputs of hidden vectors (embedding vectors), not so much the probability output.

*CBOW:* Predict current word from window of surrounding context words, train to predict which word fits in a sequence, with a single word intentionally removed. Want to max probability of correct word being predicted. Negative Sampling: Solve overfitting/slow speed by allowing each training sample to update a small % of weights.

*Skipgram:*  Predict surrounding window of words based on input word, loop on each word of sentence.

Doc2Vec: Understand which words are used tgt, computed based on bag of words of all documents. Finds prob of each doc in pool of words.

**Generative Adversarial Network**

Goal: Generator: Generates new data similar to input data and increases noise (tries to fool discriminator); Discriminator: Evaluates whether generated samples are real or fake.Feed back loop from discriminator to generator.

Adv: Introduces noise and randomness, that improves variation in data and prevents overfitting. Auto learns and discovers underlying patterns in complex data distributions.

