DEEP LEARNING

(PART 2)

Contents

- Optimizers
- Shortcoming of RNNs
- Long Short-Term Memory (LSTM)
- Information Regulation using the Gating Mechanism
- PyTorch LSTM
- LSTM Layers
- Loss Function
- Learning Rate

Optimizers

Neural Network Optimizers

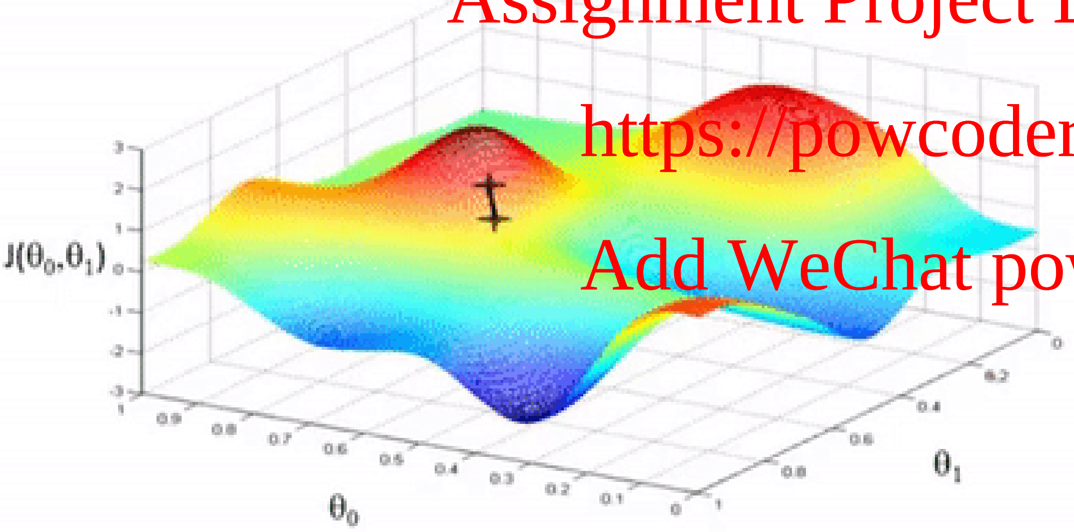
A neural network typically involves a large number of nodes and connections, optimizing the weights for each perceptron's connections cannot therefore be done manually and is often performed by an optimizer. An optimizer is an algorithm that changes the attributes of the neural network such as weights and learning rate to reduce the loss. Optimizers are used to solve optimization problems by minimizing the loss function.

How do neural network optimizers work

- For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on
- It is impossible to know which direction to go in, but there is one thing the hiker can know: if he/she is going down (making progress) or going up (losing progress)
- Eventually, if the hiker keeps taking steps that lead him/her downwards, he/she will reach the base
- Similarly, it's impossible to know what your model's weights should be right from the start
- But with some trial and error based on the loss function, you can end up getting there eventually
- How you should change your weights or learning rates to reduce the losses is defined by the optimizers you use
- Optimization algorithms are responsible for reducing the losses and to provide the most accurate results possible

Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function

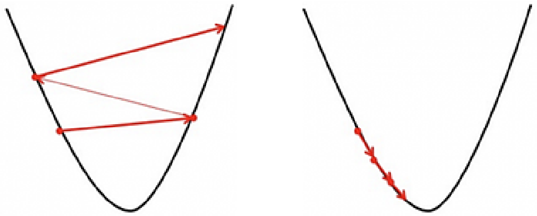
A gradient measures how much the output of a function changes if you change the inputs a little bit



How big the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slows we will move towards the optimal weights.

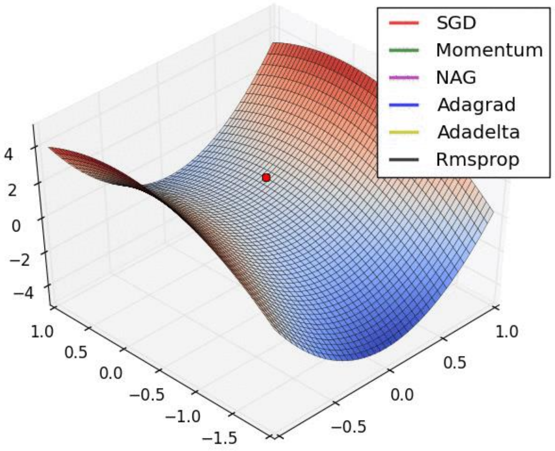
For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high.

If the steps are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent. If the steps are too small, gradient descent will eventually reach the local minimum but that may take a while.



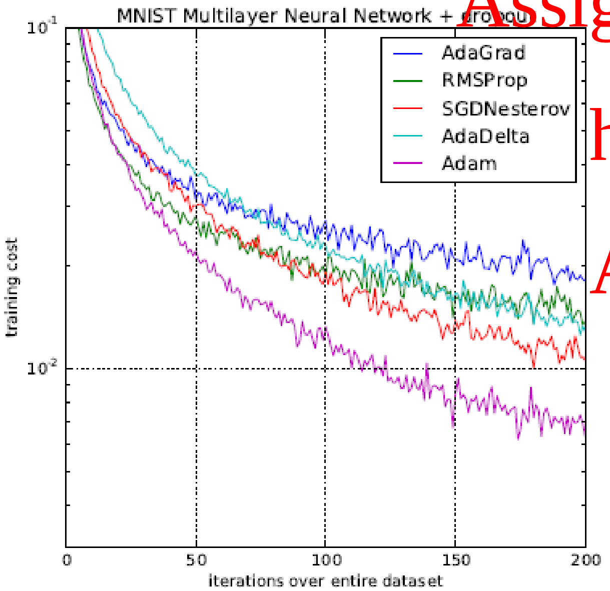
high learning rate low learning rate

Neural Network Optimizers



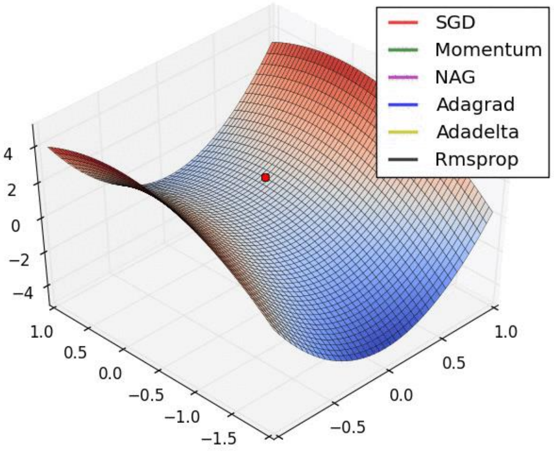
- Gradient Descent
- Stochastic Gradient Descent (SGD)
- Mini-Batch SGD (MB-SGD)
- SGD with Momentum
- Nesterov Accelerated Gradient (NAG)
- Adaptive Gradient (AdaGrad)
- AdaDelta
- RMSprop
- **Adaptive Momentum Estimation (Adam)**
- … many others

Adaptive Momentum Estimation (Adam)



- Calculates adaptive learning rates for each parameter
- Can be considered as a combination of RMSprop and Stochastic Gradient Descent (SGD) with momentum
  - Like RMSprop, Adam keeps an exponentially decaying average of past squared gradients
  - Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface
- Straight forward to implement, computationally efficient, little memory requirements, invariant to diagonal rescaling of the gradients, no stationary objective required, well suited for problems that are large in terms of data/parameters and sparse gradient

Adam is the Preferred Optimizer



- SGD (red) is stuck at a saddle point
  - So SGD can only be used for shallow networks
- All the other algorithms except SGD finally converges one after the other, AdaDelta being the fastest followed by momentum algorithms
- AdaGrad and AdaDelta can be used for sparse data
- Momentum and NAG work well for most cases but is slower
- Adam is not shown but it is the fastest algorithm to converge to minima and is considered the best algorithm amongst all the algorithms included

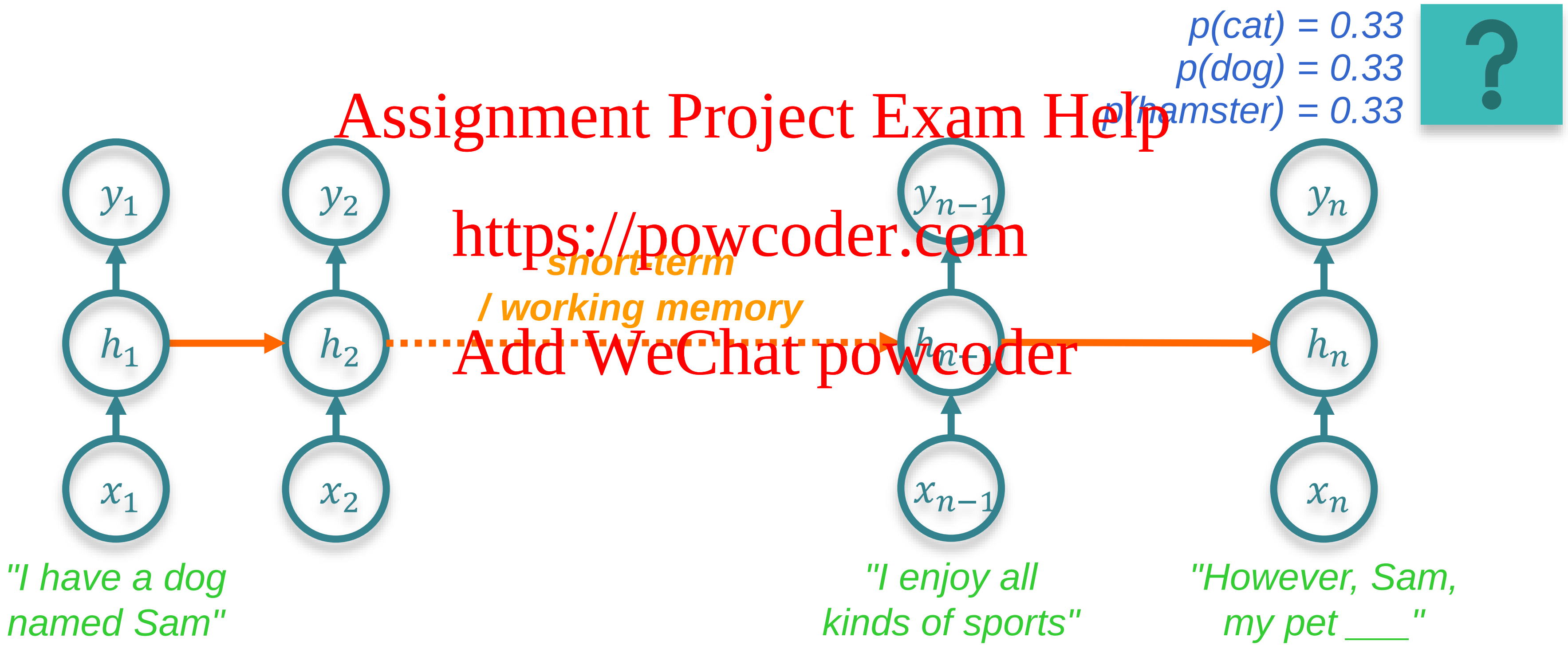
Shortcoming of RNNs

RNNs carry information over different time steps rather than keeping all the inputs independent of each other

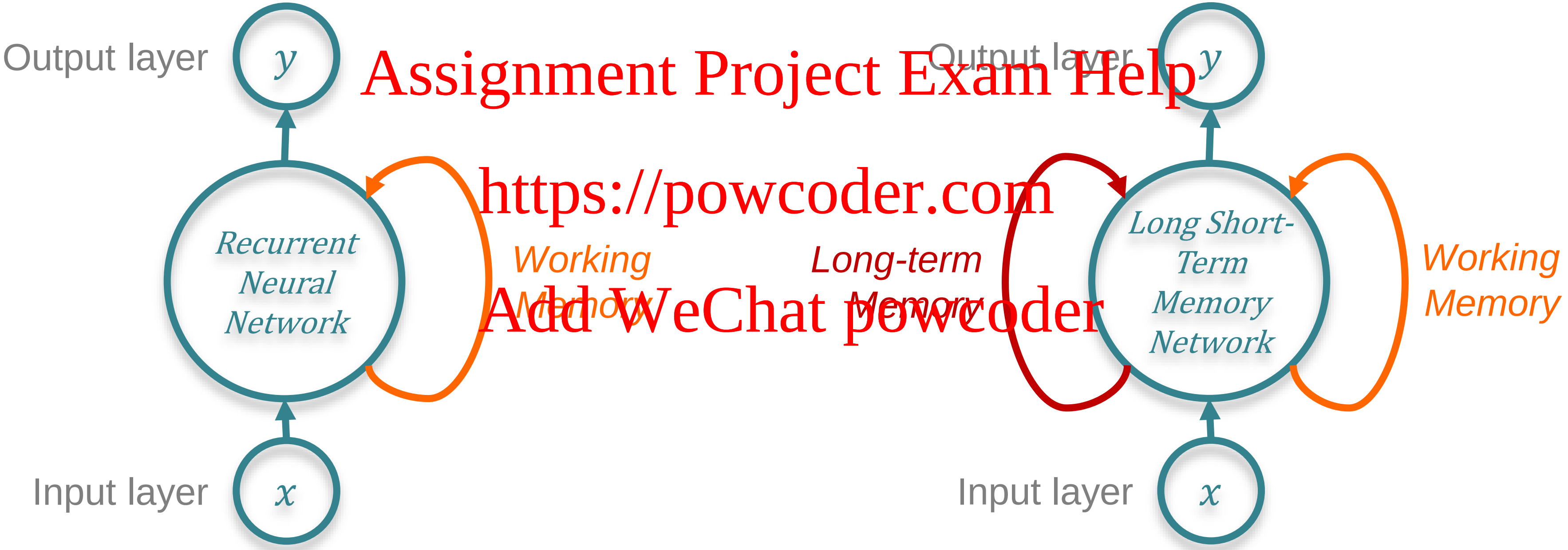
- Having a gradient that is too small prevents the weights from updating and learning, whereas extremely large gradients cause the model to be unstable
- RNNs are therefore unable to work with longer sequences and hold on to long-term dependencies, making them suffer from "**short-term memory**"
- A significant shortcoming that plagues the typical RNN is the problem of vanishing / exploding gradients
- These problems arise when back-propagating through the RNN during training, especially for networks with deeper layers
- The gradients have to go through continuous matrix multiplications during the back-propagation process due to the chain rule, causing the gradient to either shrink exponentially (vanish) or blow up exponentially (explode)

Long Short-Term Memory (LSTM)

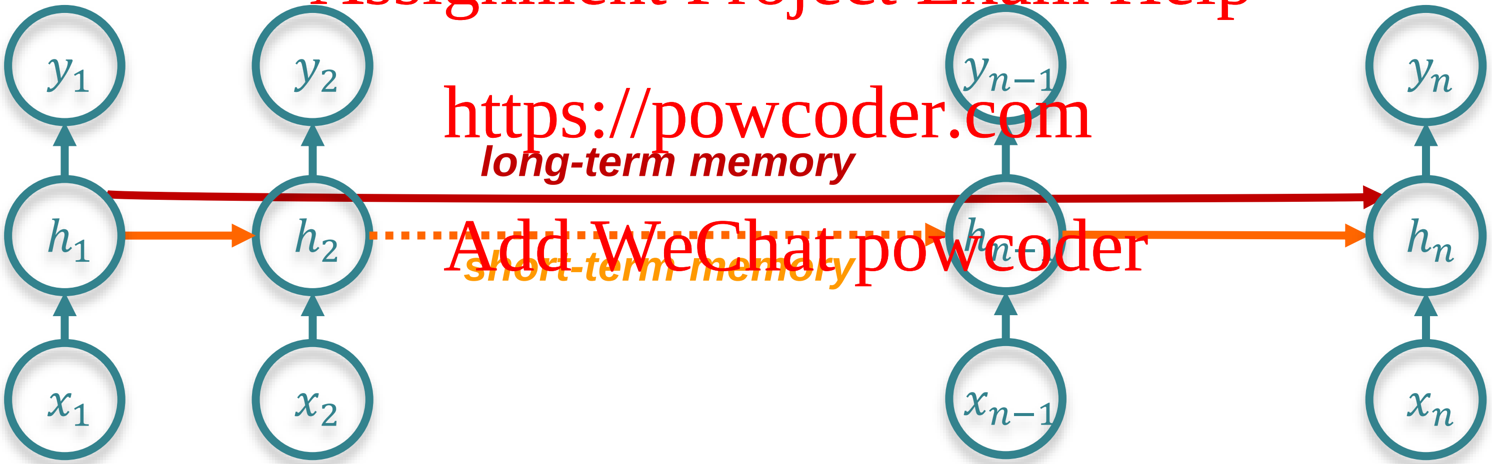
RNNs are unable to remember information from much earlier - the context is lost



LSTM is a kind of RNN, its support on long-term memory and the use of gating mechanism is what sets it apart



LSTM can retain earlier information through its long-term memory

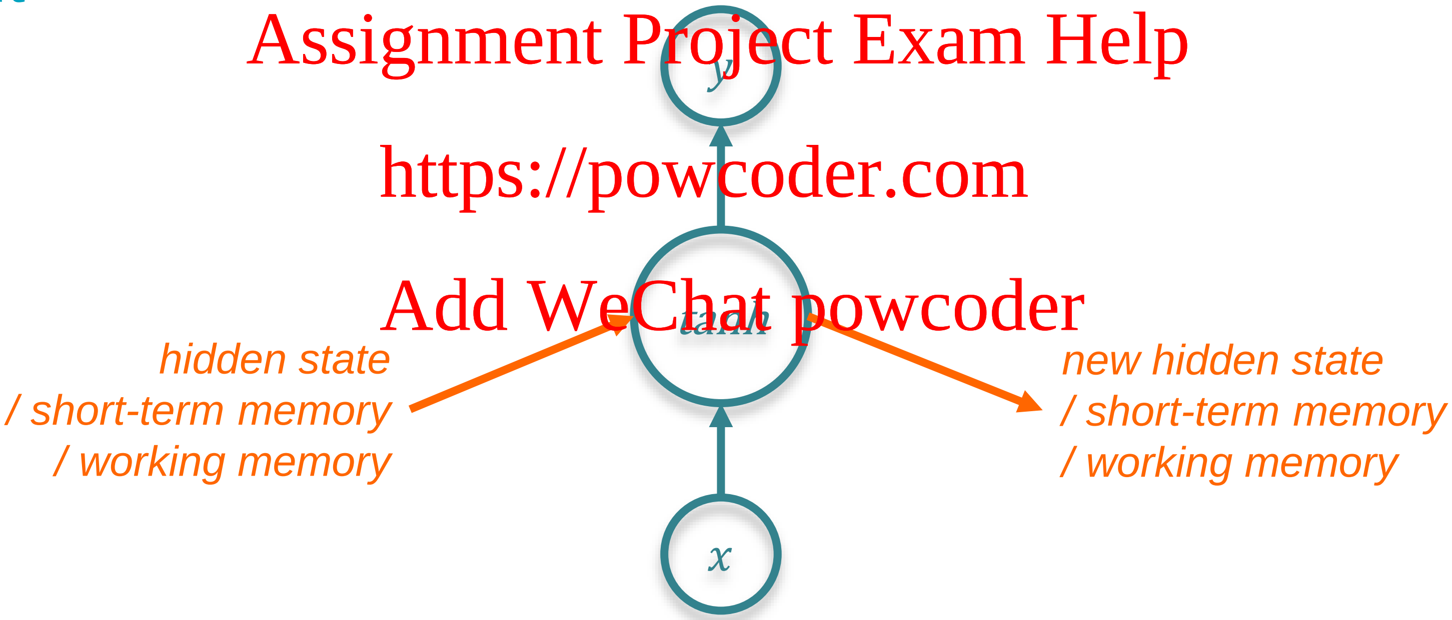


"I have a dog named Sam"

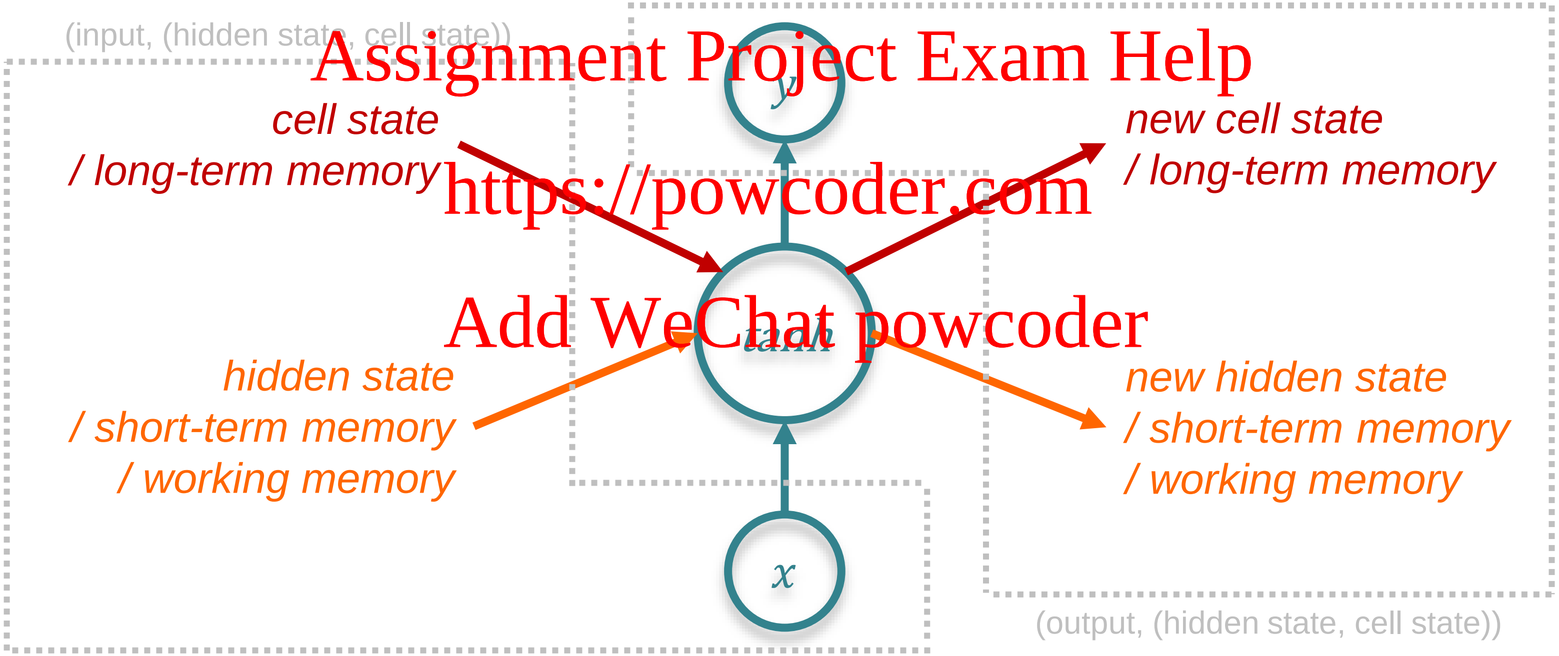
"I enjoy all kinds of sports"

"However, Sam, my pet ___"

In a normal RNN cell, the input at a time-step and the hidden state from the previous time-step is passed through a tanh activation function to produce a new hidden state and output

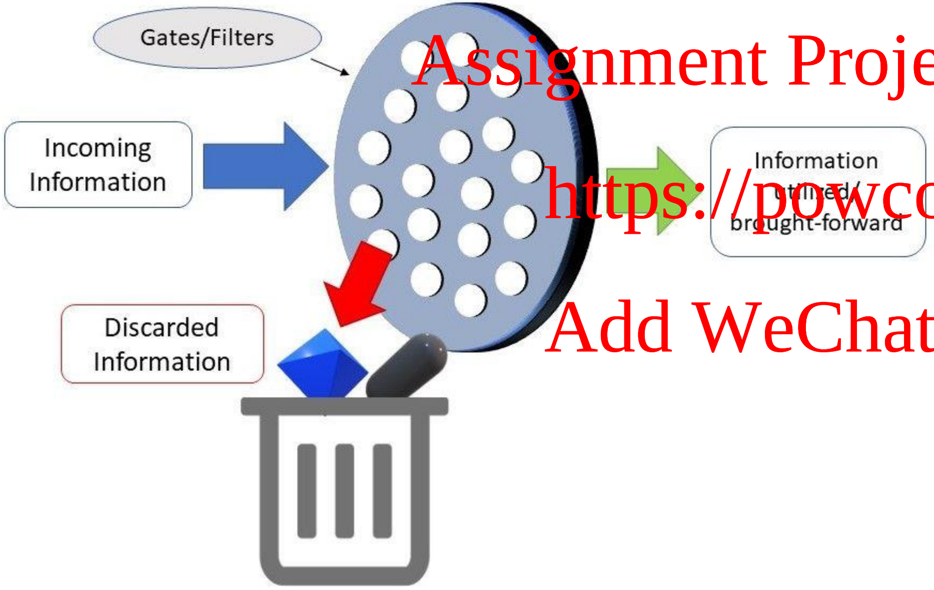


An LSTM cell has a slightly more complex structure and takes in 3 inputs: the current input data, the short-term from the previous time-step, and the long-term memory



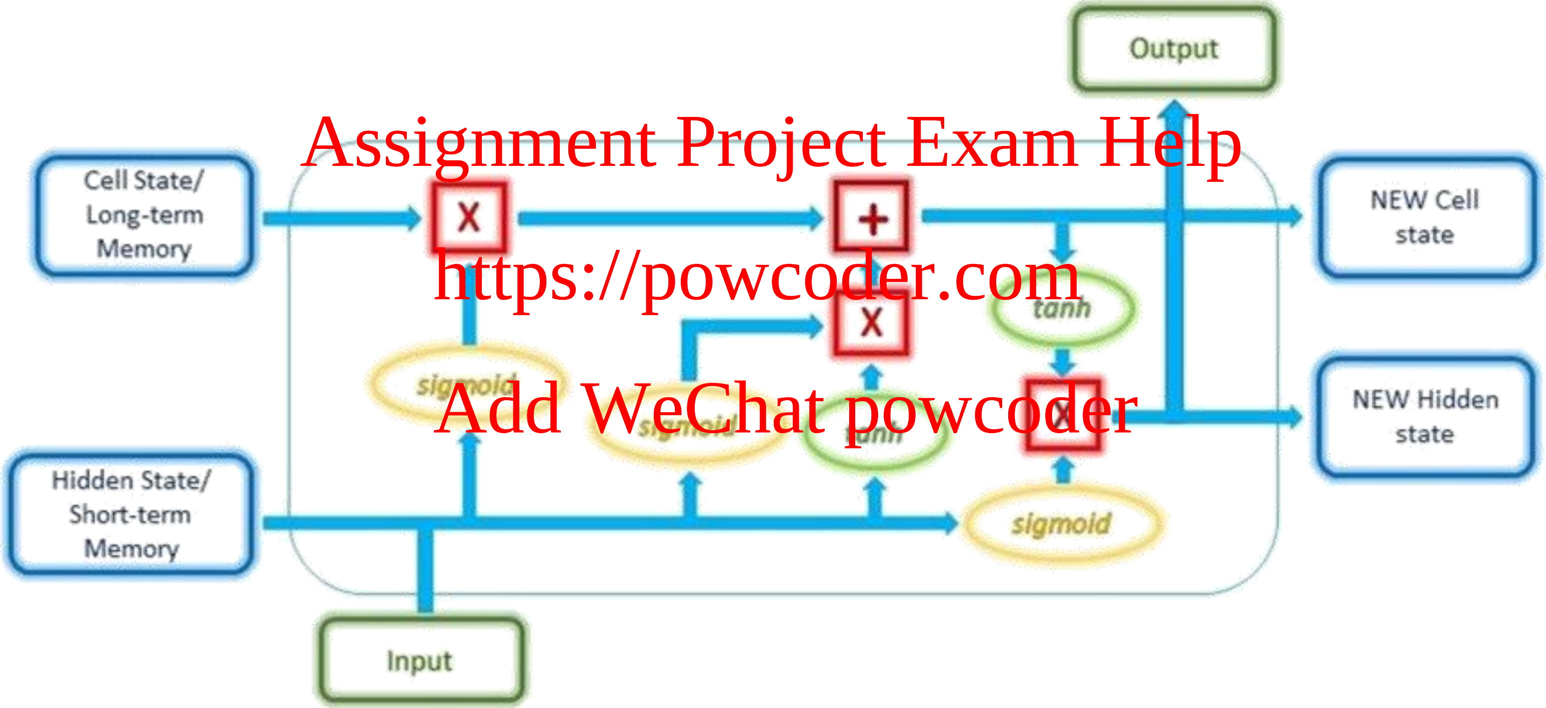
Information Regulation using the Gating Mechanism

A gate can be seen as a filter that lets relevant information through and discards irrelevant information

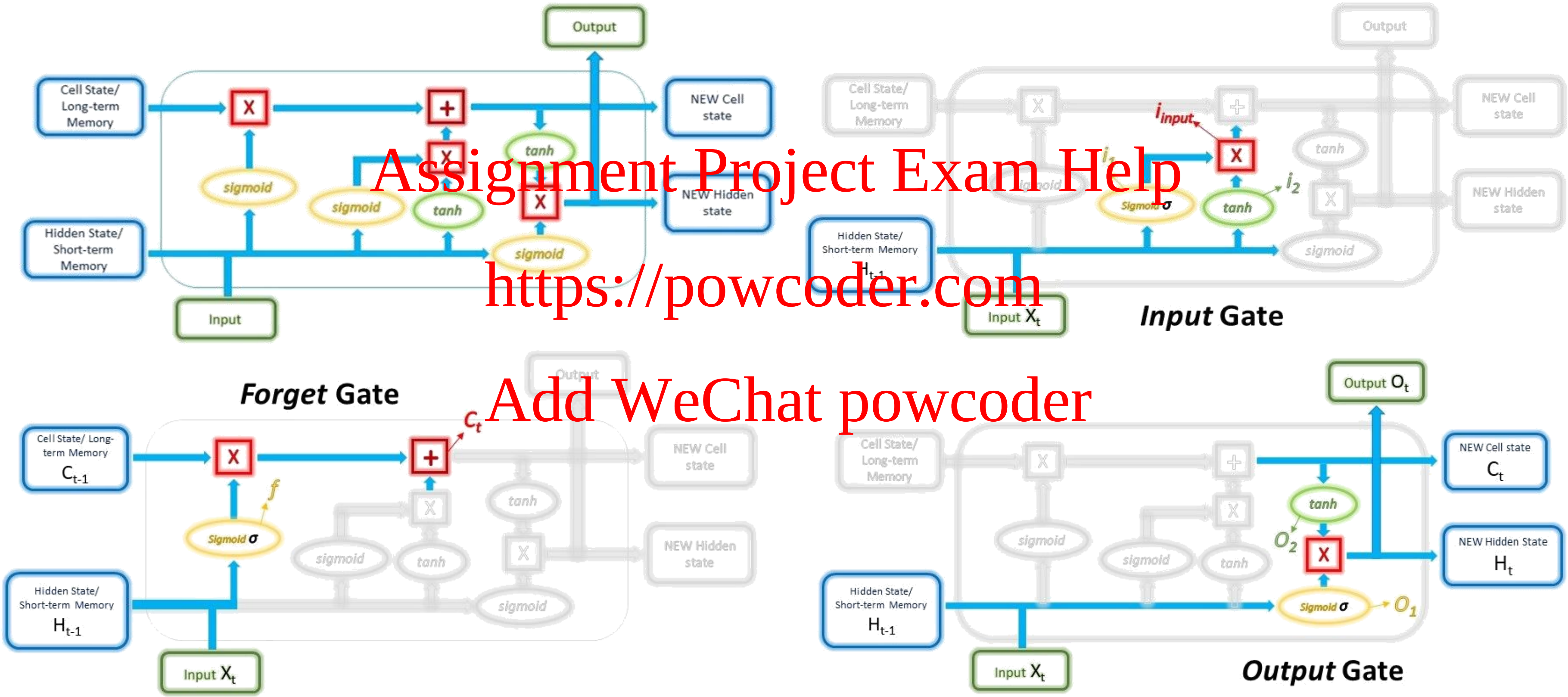


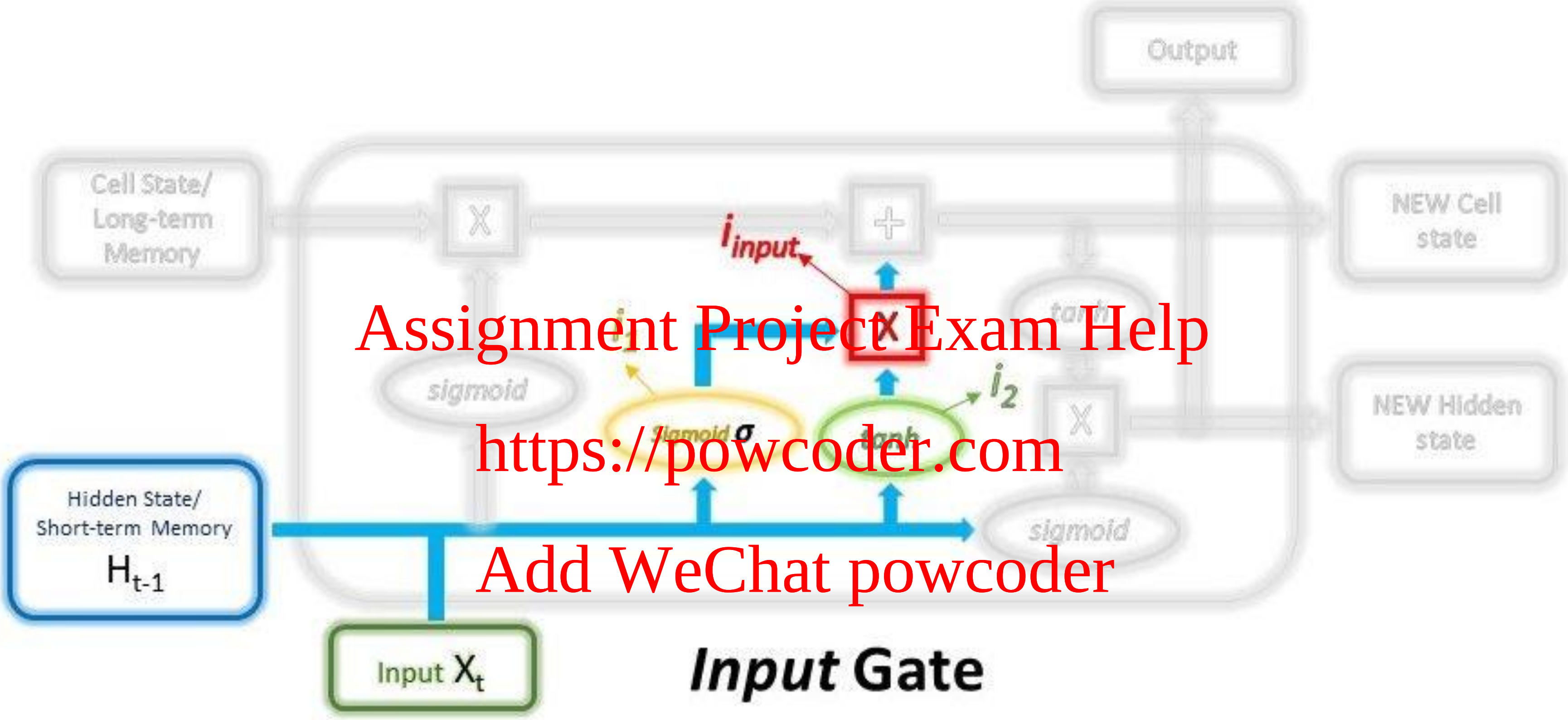
- LSTM cell uses gates to regulate the information to be kept or discarded at each time-step before passing on the long-term and short-term information to the next cell
- Ideally, the role of these gates is supposed to selectively remove any irrelevant information
- At the same time, only holds on to the useful information
- These gates need to be trained to accurately filter what is useful and what is not

LSTM uses Input Gate, Forget Gate, and Output Gate to regulate information flowing across the network

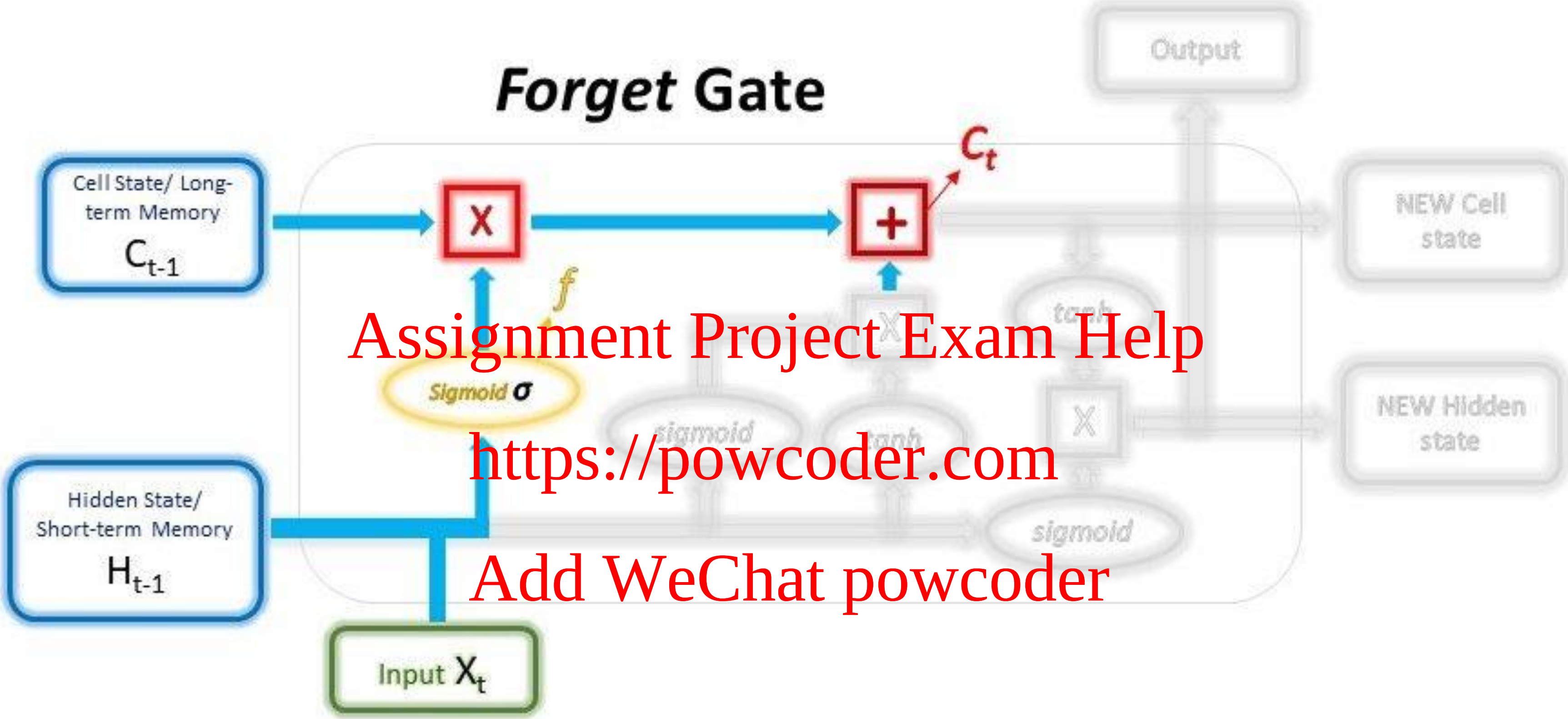


LSTM Input Gate, Forget Gate, and Output Gate



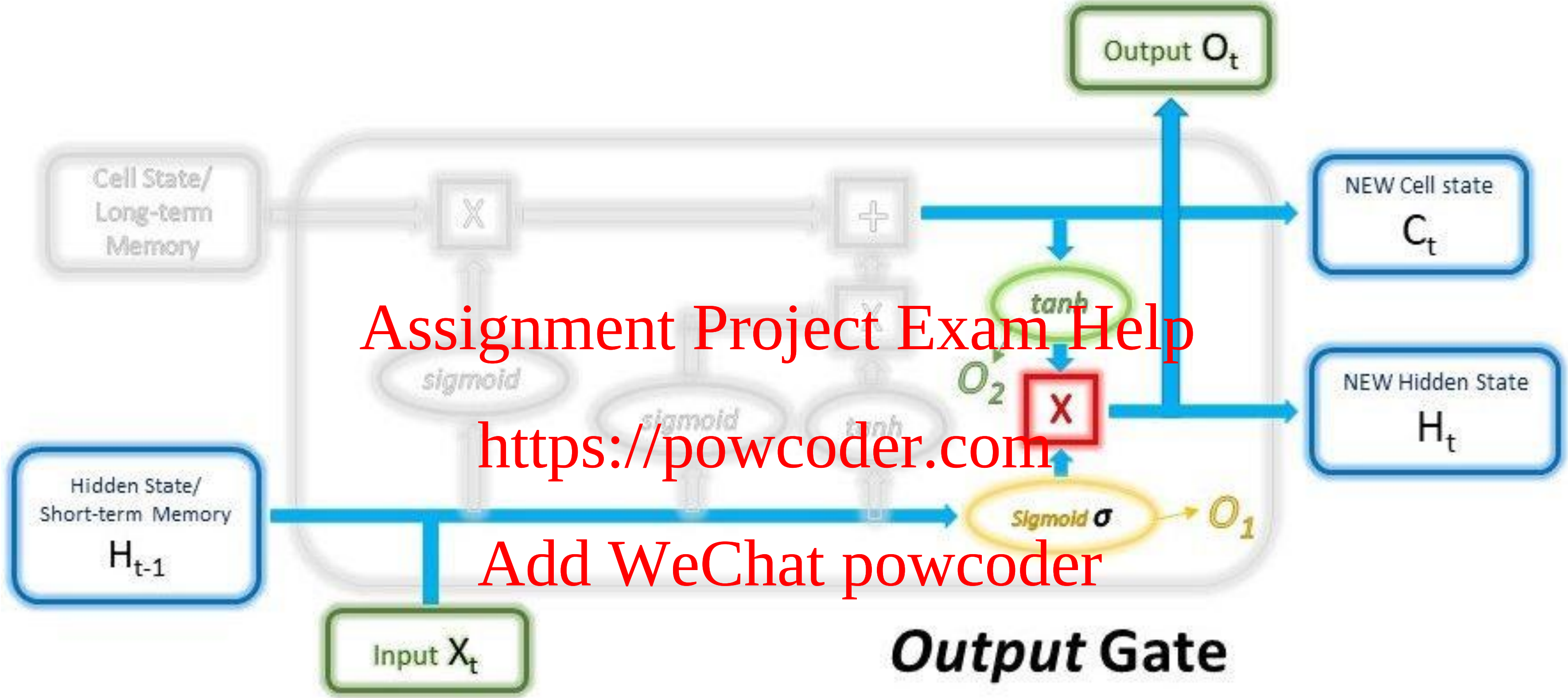


$i_1 = \sigma(W_{i_1} \cdot (H_{t-1}, x_t) + bias_{i_1})$ $i_2 = tanh(W_{i_2} \cdot (H_{t-1}, x_t) + bias_{i_2})$ $i_{input} = i_1 \cdot i_2$



$$f = \sigma(W_{forget} \cdot (H_{t-1}, x_t) + bias_{forget})$$

$$C_t = C_{t-1} \cdot f + i_{input}$$

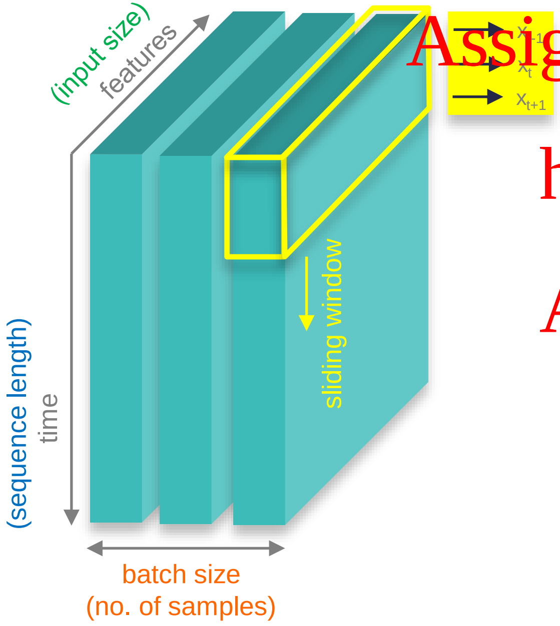


$$O_1 = \sigma(W_{output_1} \cdot (H_{t-1}, x_t) + bias_{output}) \qquad O_2 = tanh(W_{output_2} \cdot C_t + bias_{output_2}) \qquad H_t, O_t = O_1 \cdot O_2$$

It should be noted that the output is actually a tuple containing both the hidden state and the prediction value

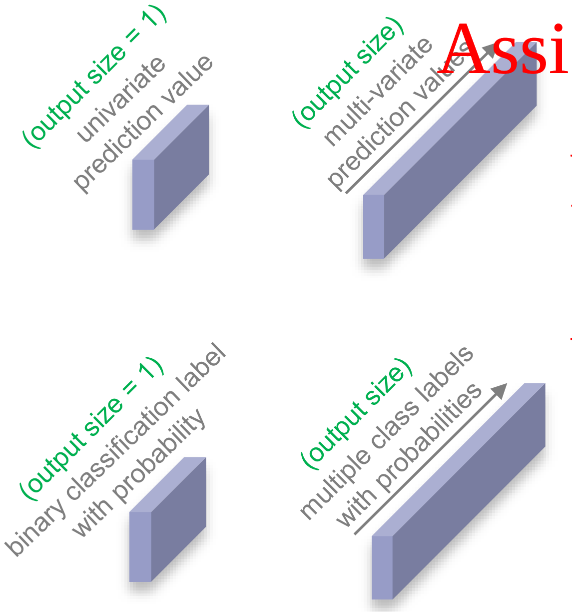
LSTM Layers

Input Layer



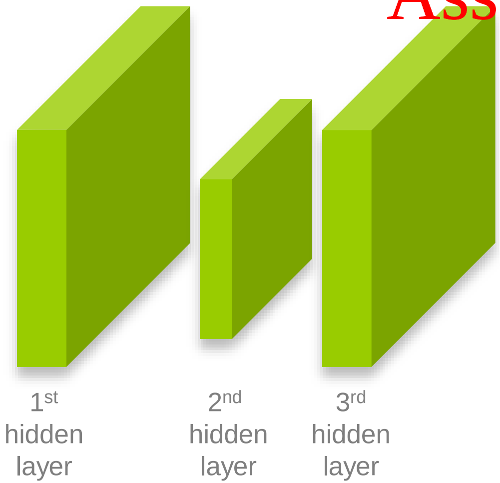
- Every neural network has one input layer
- The number of perceptrons/nodes in this layer is completely and uniquely determined by the number of features uses to make predictions (i.e. **input size** or **input dimension**)
- For time series data
  - One row of features represents the input at one time step
  - The number of time steps represents the sequence length
  - Data along a series of time steps form a batch of data
  - The number of samples is referred to as batch size
- For some problems, prediction is made using several time steps (e.g. moving average), the number of time steps used is referred to as a window

Output Layer



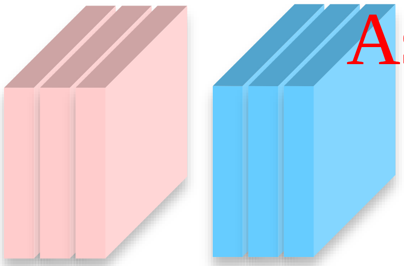
- Every neural network has exactly one output layer
- The number of perceptrons/nodes in this layer is determined by the number of predictions to make (e.g. 4 perceptrons when predicting a bounding box)
- For classification problems, the output layer can comprise of one single node unless the softmax function is applied giving the output layer one node per class label
- For regression problems, the output layer typically comprises of a single node but multiple values are also possible

Hidden Layers



- Linear data does not need any hidden layer!
- One hidden layer is sufficient for the large majority of problems
  - There is a consensus that there are very few situations in which a 2nd or 3rd hidden layer would improve performance
  - There are however counter examples that cannot directly be learned via a single one-hidden-layer MLP or require an infinite number of nodes
- Increasing hidden layers would also increase the complexity of the model and choosing hidden layers such as 8, 9, or in two digits may sometimes lead to overfitting
- In general, the number of layers cannot be analytically calculated or the number of nodes to use per layer

Hidden Perceptrons



hidden state　　cell state

To prevent over-fitting, the number of hidden perceptrons should be

$$< \frac{number\ of\ samples}{scaling\ factor \cdot (input\ size + output\ size)}$$

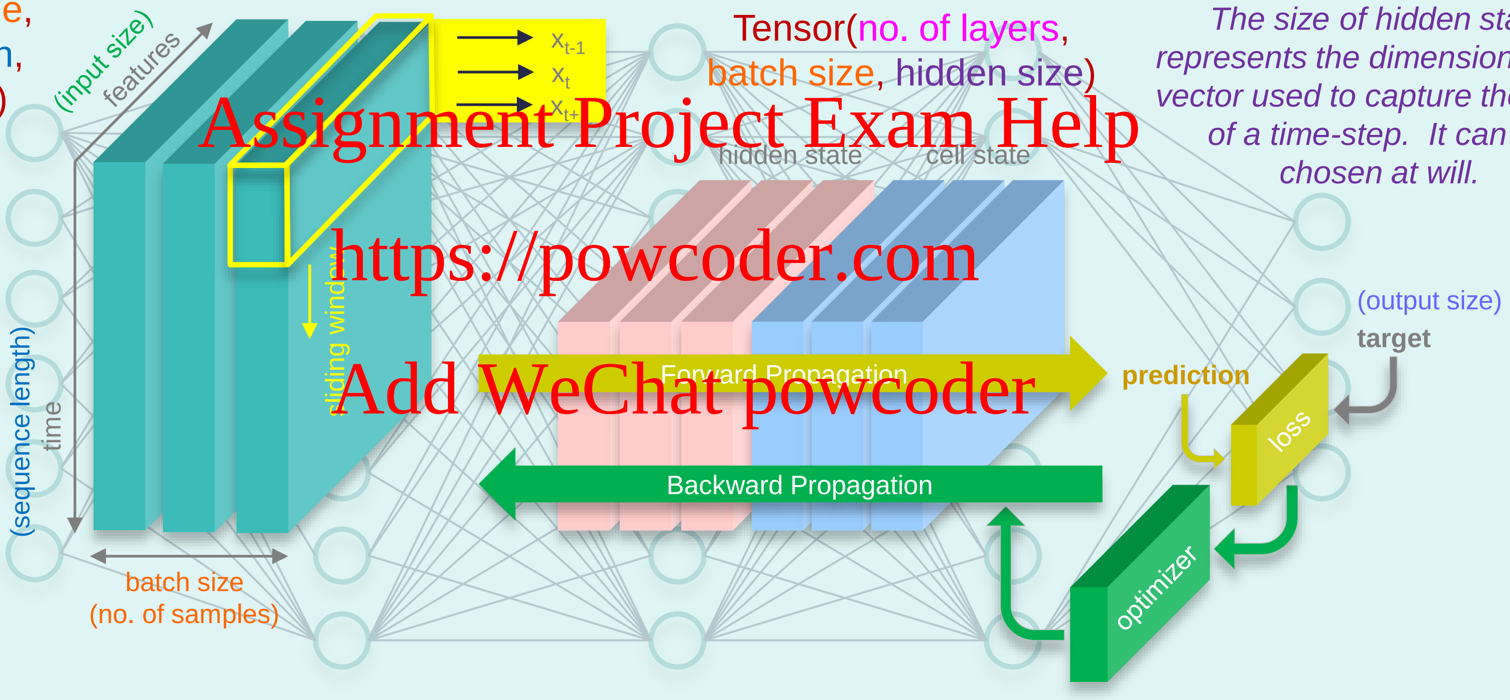
where the $scaling\ factor$ is usually between 2 and 10

- In general, using the same number of perceptrons for all hidden layers will suffice
- Usually, more performance boost can be gained from adding more layers than adding more perceptrons in each layer
- If the number of layers/perceptrons is too small, the network might not be able to learn the underlying patterns in the data and thus become useless
- Some suggested estimations:
  - Between the size of the input layer & the output layer
  - $\frac{2}{3} \cdot input\ size + output\ size$
  - $\sqrt{input\ size\ \cdot output\ size}$
  - $< 2 \cdot input\ size$

LSTM uses forward propagation to provide prediction and the loss function & optimizer to drive backward propagation

Tensor(batch size, sequence length, no. of features)

when batch_first=True



The size of hidden state represents the dimension of the vector used to capture the state of a time-step. It can be chosen at will.

Input, hidden state, and cell state are captured as tensors of specific dimensions

Input Sequential Data*
torch.Tensor(batch_size, seq_length, input_size)

Hidden State /Short-Term Memory
torch.zeros(num_layers, batch_size, hidden_size)

Cell State /Long-Term Memory
torch.zeros(num_layers, batch_size, hidden_size)

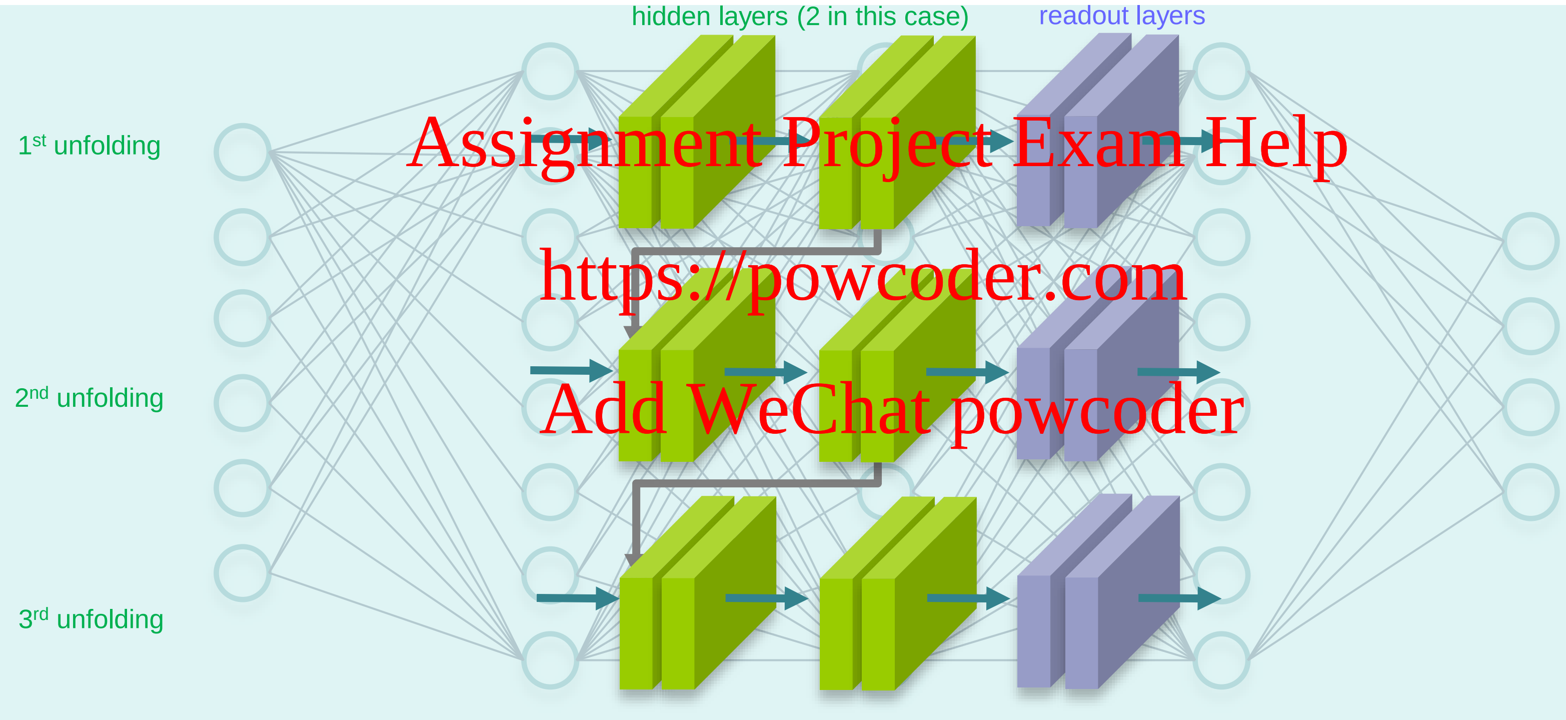
Output Data
torch.Tensor(batch_size, seq_length, output_size)

Hidden State /Short-Term Memory
torch.Tensor(num_layers, batch_size, hidden_size)

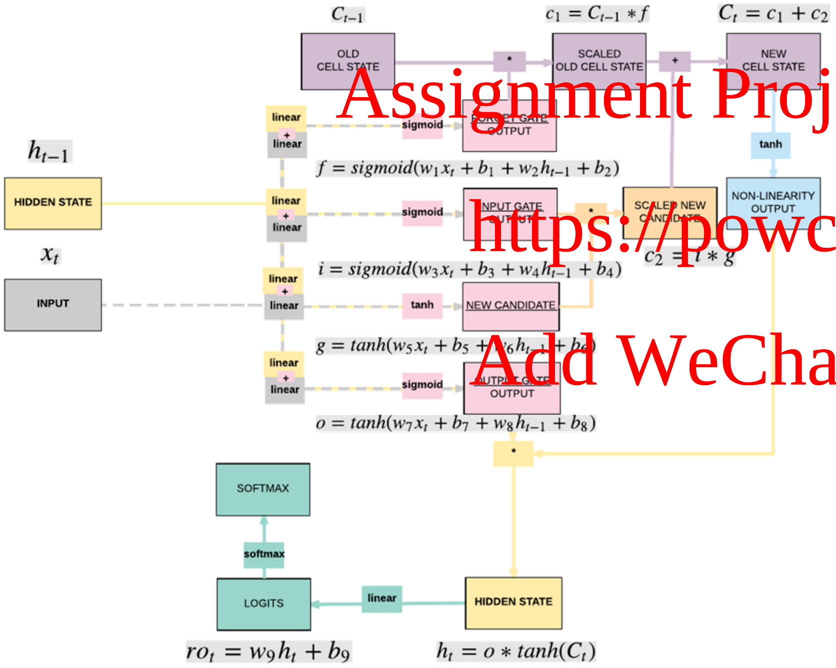
Cell State /Long-Term Memory
torch.Tensor(num_layers, batch_size, hidden_size)

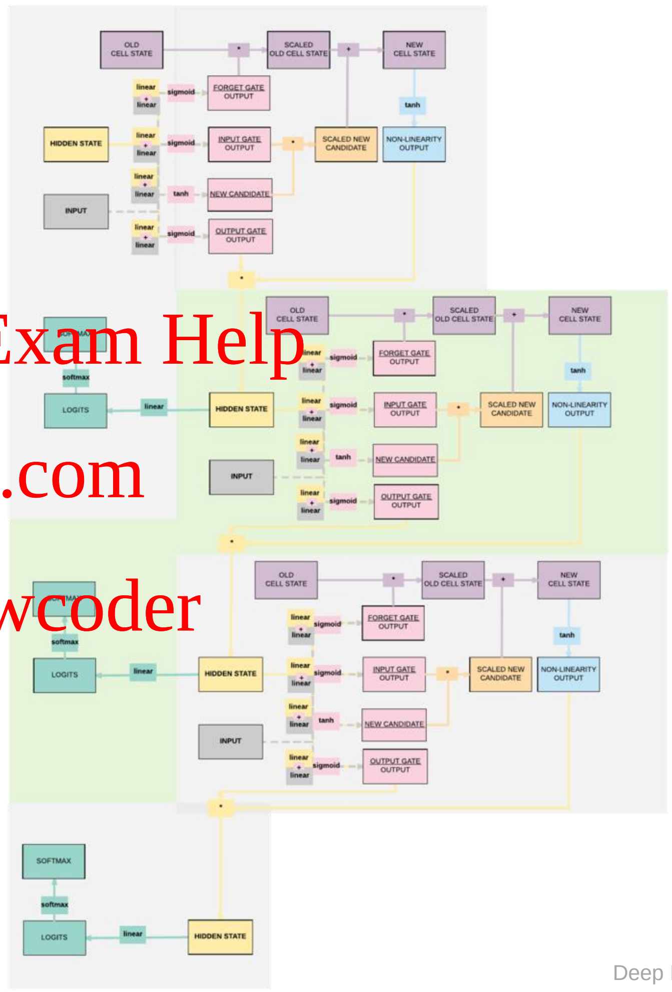
* when batch_first=True; otherwise [seq_length, batch_size, input_size]

RNN can be perceived as a stack of layers with each layer representing one unfolding of the RNN



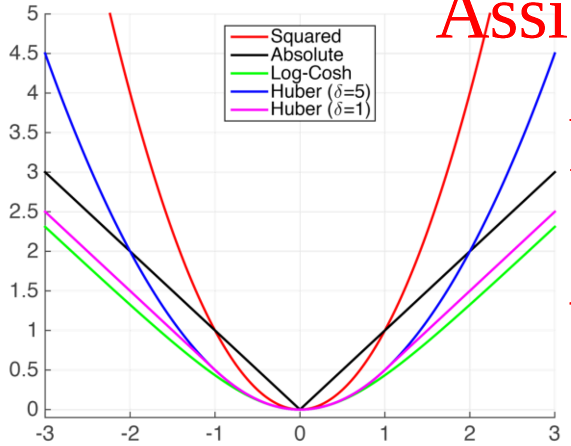
Stacking is established through the interactions in the gating mechanism





Loss Function

Loss Function



- For regression problems, **Mean Squared Error** (**MSE**) is the most common loss function to use

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(predicted_i - actual_i)^2$$

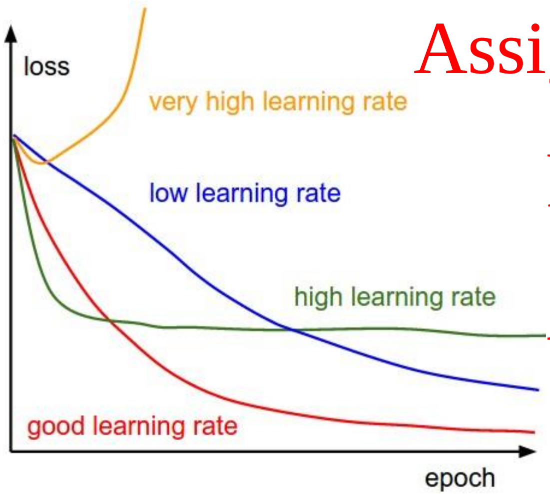
- If there are a significant number of outliers, **Mean Absolute Error** (**MAE**) or the Huber loss function can be used

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|predicted_i - actual_i|$$

- For classification problems, **cross-entropy** will serve well in most cases

Learning Rate

Learning Rate



- To find the best learning rate, start with a very low value (10^{-6}) and slowly multiply it by a constant until it reaches a very high value (e.g. 10)
- Measure the model performance (against the learning rate) to determine which rate served well for the problem
- The model can then be retrained using this optimal learning rate
- The best learning rate is usually half of the learning rate that causes the model to diverge

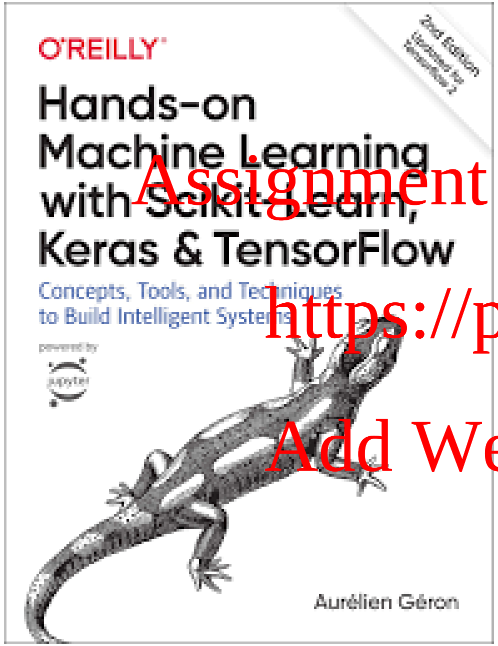
Conclusion

RNN/LSTM is designed for sequential data predictions

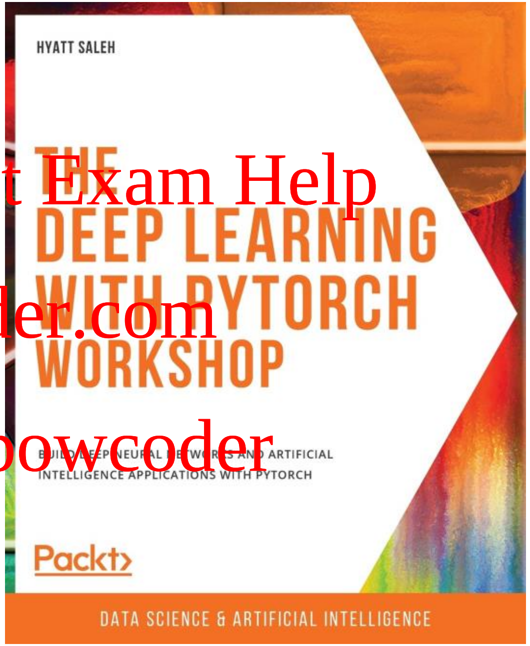
	Property	Description
1	Feature Data Types	Numeric data. Variable encoding is therefore necessary for categorical data. Normalised data is advised. For time series data, differencing is performed to make time series stationary and a number of differences (referred to as the order of integration) may be performed depending on the lag time. Train and test split should be done based on sequential sample.
2	Target Data Types	Numeric data. Univariate/Multivariate predicted values. Univariate/Multivariate predicted class labels..
3	Key Principles	Random weights are assigned initially. Multiple layers and non-linear activation functions are used to capture complex patterns. Loss function is used to compute the accuracy and drives the backward propagation that updates the gradients and therefore the weights. The number of hidden states reflects the complexity that can be captured by the RNN. Learning rate often begins with a low value, e.g. 10^{-6}.
4	Hyperparameters	Number of hidden layers, number of output layers, number of perceptrons per layer, number of hidden states, loss function, optimizer, learning rate, number of epochs. No hard and fast rules to determine the number of hidden/output layers, number of perceptrons/layer, number of hidden states.
5	Data Assumptions	Non-parametric – no assumption about data distribution. All data are used.
6	Performance	Training time is generally high. The use of GPU, TPU, and various distributed platforms should provide better performance.
7	Accuracy	.Accuracy has more correlation with the quality and size of the dataset than the number of layers or number of perceptrons per layer.
8	Explainability	Poor.

References

References



“Hands-On Machine Learning with Scikit-Learn and TensorFlow”, Aurelien Geron, O'Reilly Media, Inc., 2017



“The Deep Learning with PyTorch Workshop”, Hyatt Saleh, Packt Publishing, 2020

References

- "Long Short-Term Memory: From Zero to Hero with PyTorch", Gabriel Loye (https://blog.floydhub.com/long-short-term-memory-from-zero-to-hero-with-pytorch/)
- “Understanding LSTM and its Diagrams”, Shi Yan (https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714)
- “Understanding LSTM Networks”, Christopher Olah (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)
- "Illustrated Guide to LSTM's and GRU's: a Step by Step Explanation" (https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21)
- "Neural Networks" (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py)
- "Designing Your Neural Networks" (https://www.kdnuggets.com/2019/11/designing-neural-networks.html)
- "pytorch / tutorials" (https://github.com/pytorch/tutorials) (https://pytorch.org/tutorials/)

References

- "Counting No. of Parameters in Deep Learning Models by Hand", Raimi Karim (https://towardsdatascience.com/counting-no-of-parameters-in-deep-learning-models-by-hand-8f1716241889)
- "Hyperparameter Tuning in Python: a Complete Guide 2020" (https://neptune.ai/blog/hyperparameter-tuning-in-python-a-complete-guide-2020)
- "How To Make Deep Learning Models That Don't Suck" (https://nanonets.com/blog/hyperparameter-optimization/)
- "Practical Guide to Hyperparameters Optimization for Deep Learning Models" (https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/)
- Ray documentation (https://docs.ray.io/en/master/index.html)
- Ray documentation (https://simon-ray.readthedocs.io/en/latest/tune.html)

THANK YOU