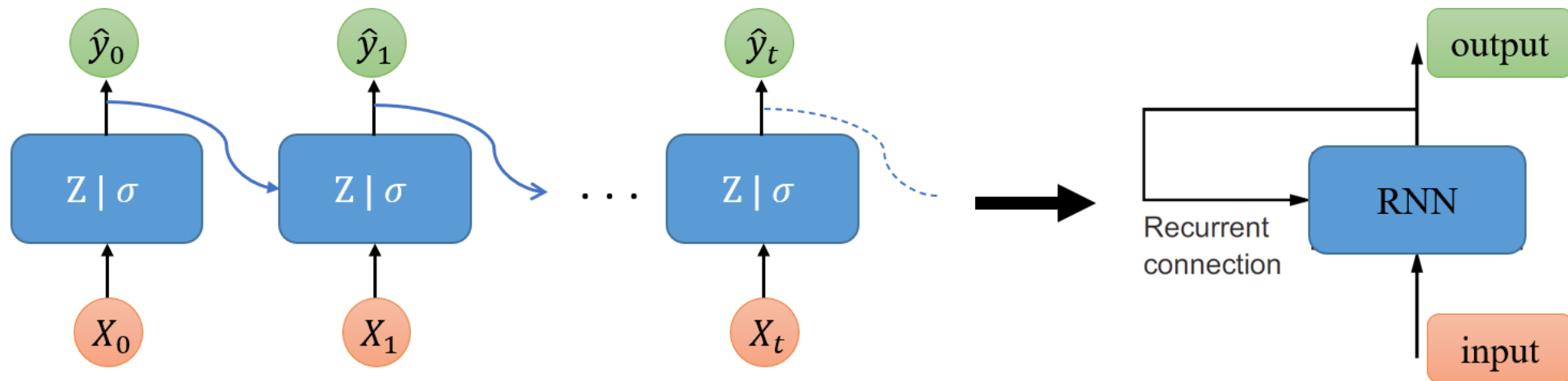


Module 6 – Part I

Deep Sequence Modeling

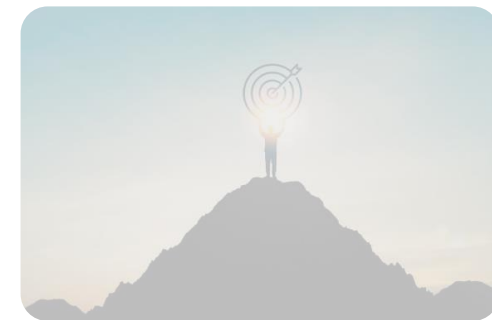
Recurrent Neural Networks (RNN)





Road map!

- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- **Module 6- Deep Sequence Modeling (RNN, LSTM, NLP)**
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

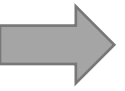


➔ Different kinds of sequence data

Sequence data refers to any data that has a specific order or sequence to it!

- **Time series data:** sequence of data measured at regular intervals over time. (stock prices, weather patterns, medical records, ...)
- **Text data:** sequence of data that is composed of words, sentences, or paragraphs. (tweets, news articles, product reviews, ...)
- **Audio data:** sequence of data that is recorded or generated as sound waves. (speech recordings, music tracks, ...)
- **Video data:** sequence of data that is represented as a sequence of images or frames. (movie clips, surveillance footage, ...)



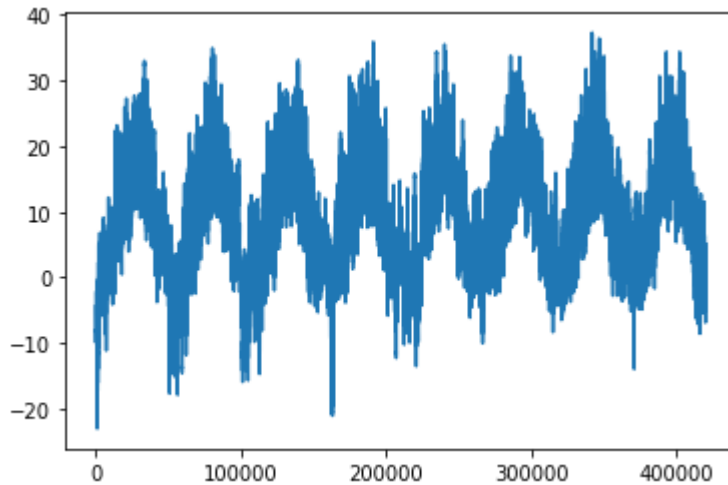


Different kinds of timeseries tasks

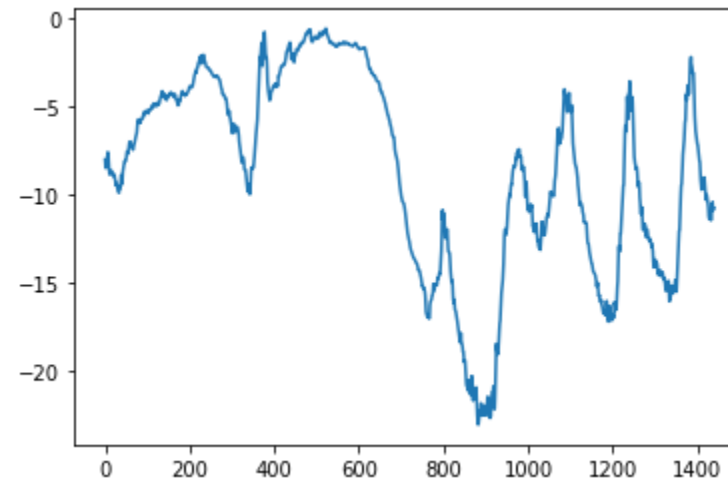
- **Forecasting**: Predicting what happens next
Electricity consumption, temperature forecast, stock prediction, ...
- **Classification**: Assign one or more labels to a time series
Classify website visitors as human or bot based on their activity
- **Event detection**: Identify the occurrence of a specific expected event within a continuous data stream
Hotword detection: “Ok Google” or “Hey Siri”
- **Anomaly Detection**: Detect anything unusual happening within a continuous data stream
Unusual activity on a network, ...

➔ A simple timeseries example

- A temperature forecasting example: [deep-learning-with-python-notebooks](#)
- Predicting the temperature 24 hours in the future
 - Target: temperature
 - Features: 14 different variables including pressure, humidity, wind direction and etc
 - Data recorded every 10 minutes from 2009-2016



Temperature between 2009-2016



Temperature in the first 10 days: $10 \times 24 \times 6 = 1440$



Preparing the data

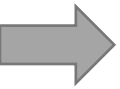
- Given the **previous 5 days (120 hours)** and samples **once per hour**, can we predict temperature **in 24 hours (after the end of the sequence)**?
- Data batches:
 - Sequence length = 120
 - $[1, 2, 3, \dots, 120][144]$
 - $[2, 3, 4, \dots, 121][145]$
 - $[3, 4, 5, \dots, 122][146]$
 - Bath size: 256 of these samples are shuffled and batched
 - Sample shape: (256, 120, 14)
 - Target shape: (256,)



Naïve forecaster: common-sense baseline

- Temperature 24 hours from now = Temperature right now
- This is our random walk with no drift forecaster.
- Performance:
 - Validation MAE = 2.44 degrees Celsius
 - Test MAE = **2.62** degrees Celsius
 - The baseline model is off by about 2.5 degrees on average. Not bad!!



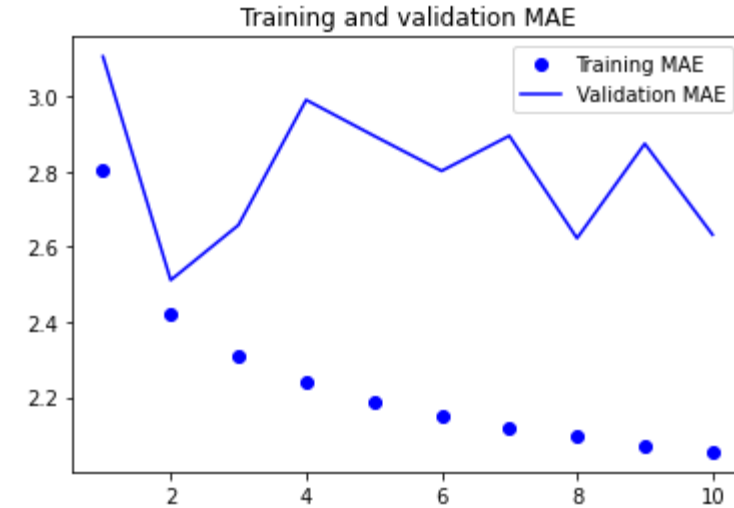


Let's try DNN (Deep Neural Networks)

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 120, 14)]	0
flatten (Flatten)	(None, 1680)	0
dense (Dense)	(None, 16)	26896
dense_1 (Dense)	(None, 1)	17

=====
Total params: 26,913
Trainable params: 26,913
Non-trainable params: 0
=====



- Test MAE = **2.62** degrees Celsius
- No improvement!!
- Flattening a timeseries data is not a good idea!

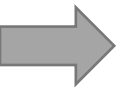


Let's try CNN (Convolutional Neural Networks)

- **Motivation:** Maybe a temporal convnet could **reuse the same representations** across different days, much like a spatial convnet can reuse the same representations across different locations in an image!

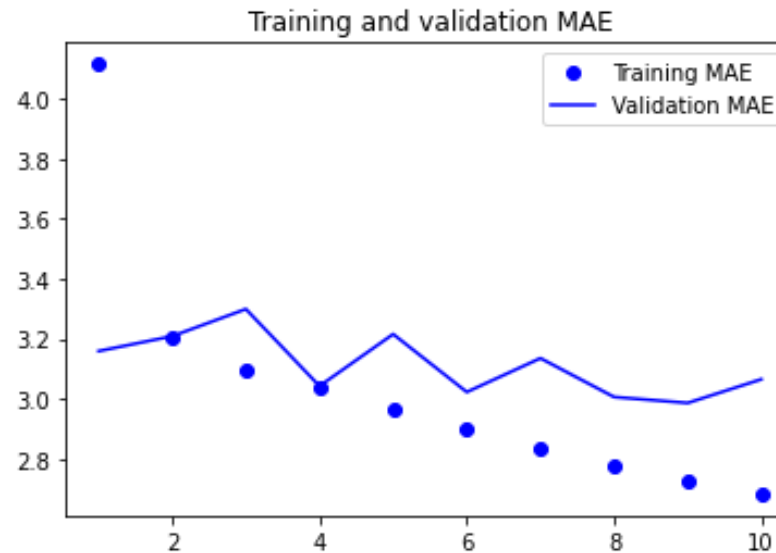
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 120, 14)]	0
conv1d (Conv1D)	(None, 97, 8)	2696
max_pooling1d (MaxPooling1D)	(None, 48, 8)	0
conv1d_1 (Conv1D)	(None, 37, 8)	776
max_pooling1d_1 (MaxPooling1D)	(None, 18, 8)	0
conv1d_2 (Conv1D)	(None, 13, 8)	392
global_average_pooling1d (GlobalAveragePooling1D)	(None, 8)	0
dense_2 (Dense)	(None, 1)	9
Total params: 3,873		
Trainable params: 3,873		
Non-trainable params: 0		



CNN performance

- Test MAE = **3.10** degrees Celsius
- Even worse than the densely connected model!!!
 - CNN treats every segment of the data the same way!
 - Pooling layers are destroying order information.



→ Sequence Modeling

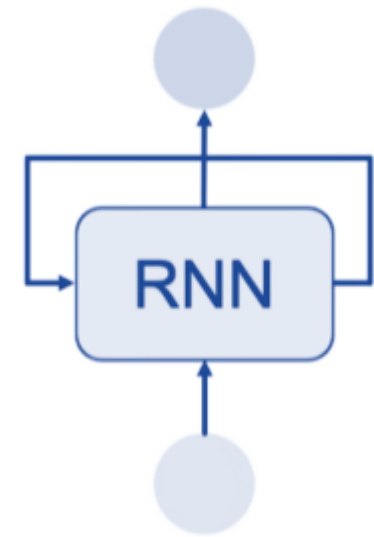
To model sequence data efficiently, we need a new architecture that:

- Preserve the **order**
- Account for **long-term dependencies**
- Handle **input-length**
- **Share parameters** across the sequence

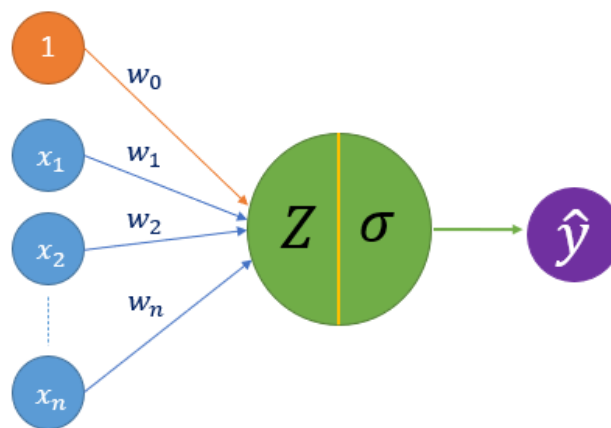
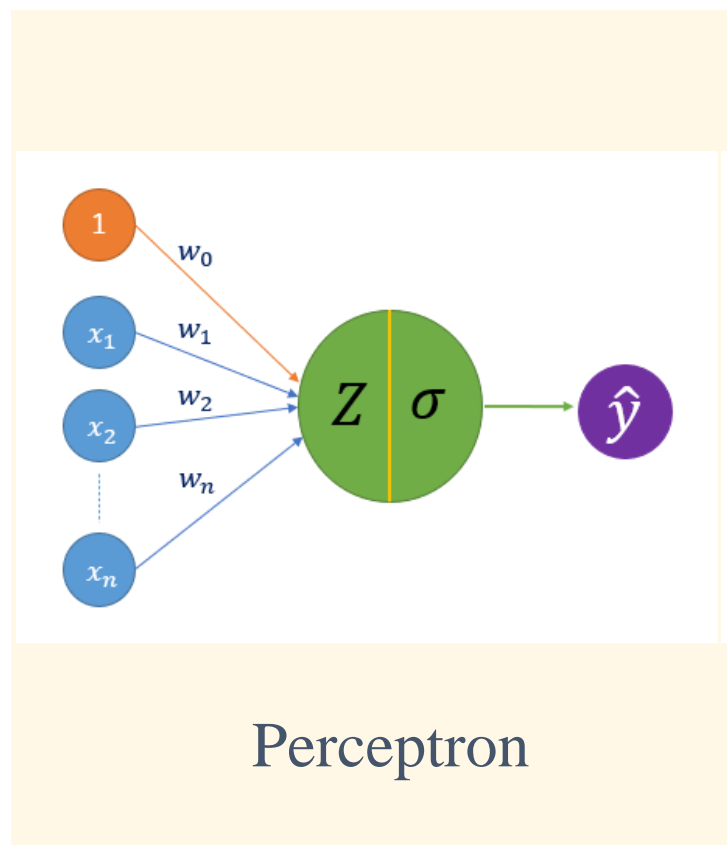


What is RNN (Recurrent Neural Network)?

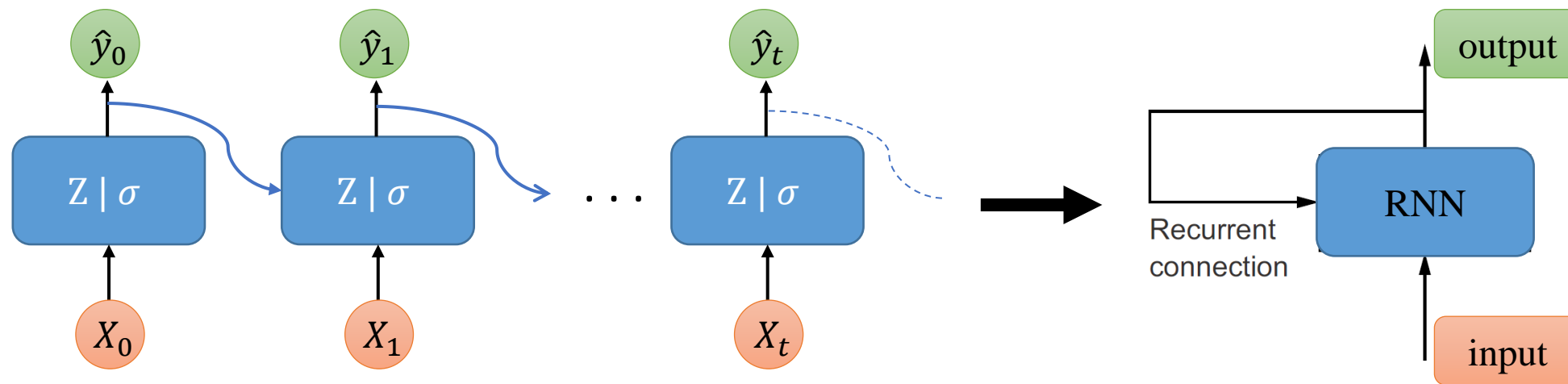
- The architecture of RNNs is inspired by the way **biological intelligence** processes information **incrementally** while **maintaining an internal model** of what it is processing.
- This ability to **remember previous inputs** and incorporate them into the current output allows RNNs to model sequential data.
- RNN maintains a **state** that contains information relative to what it has seen so far
- RNNs can be thought of as neural networks with an **internal loop**, which allows them to process sequences of varying lengths and learn from temporal dependencies.

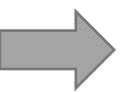


➔ Perceptron vs Recurrent Cell

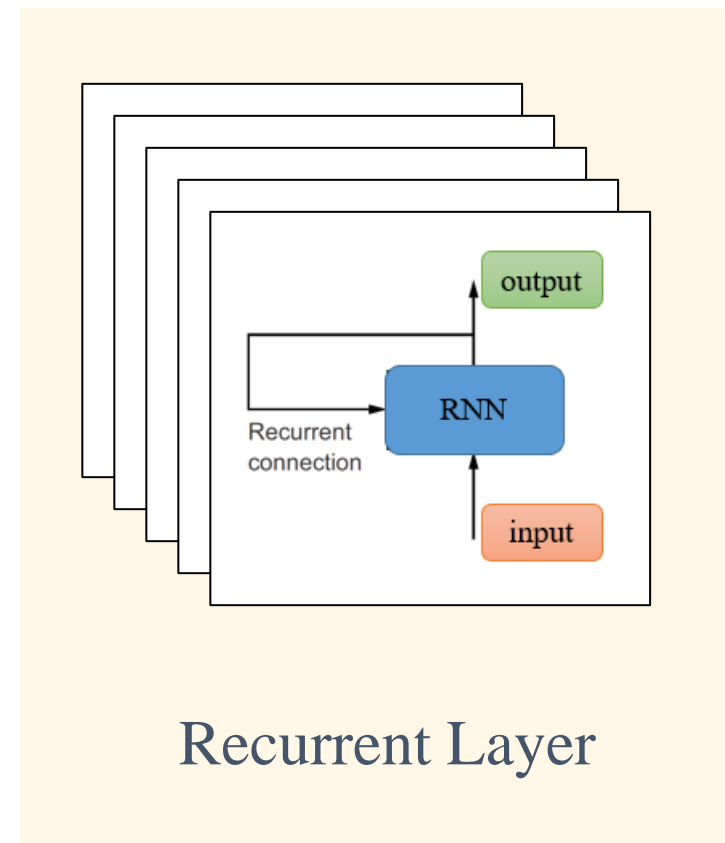
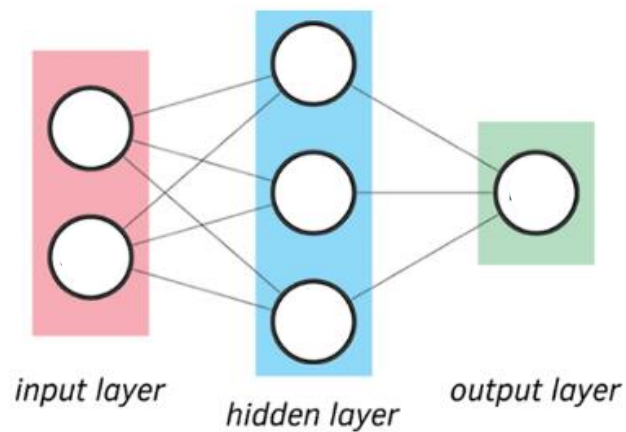
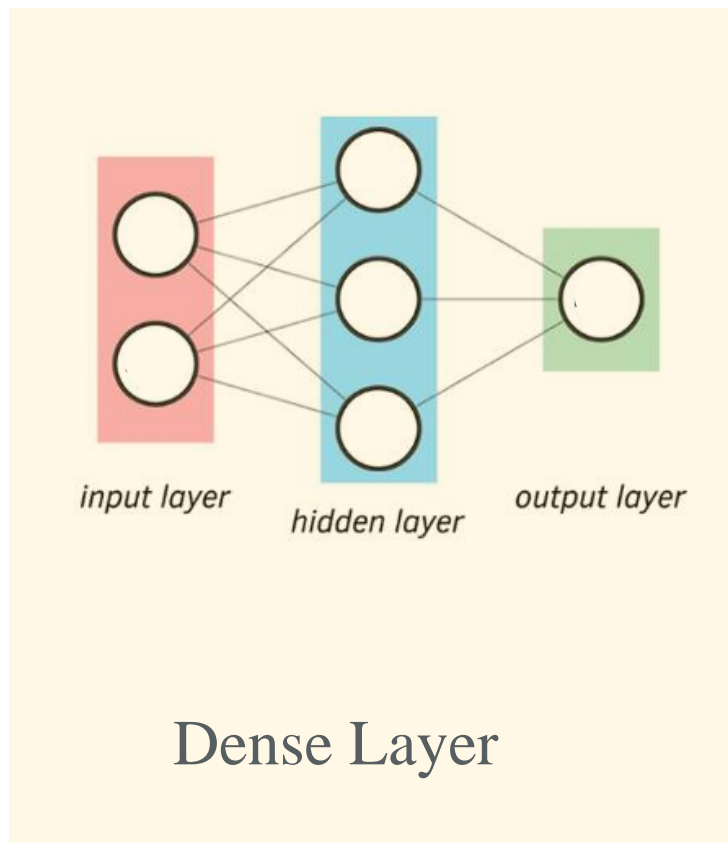


→ Unrolling the Recurrent Cell



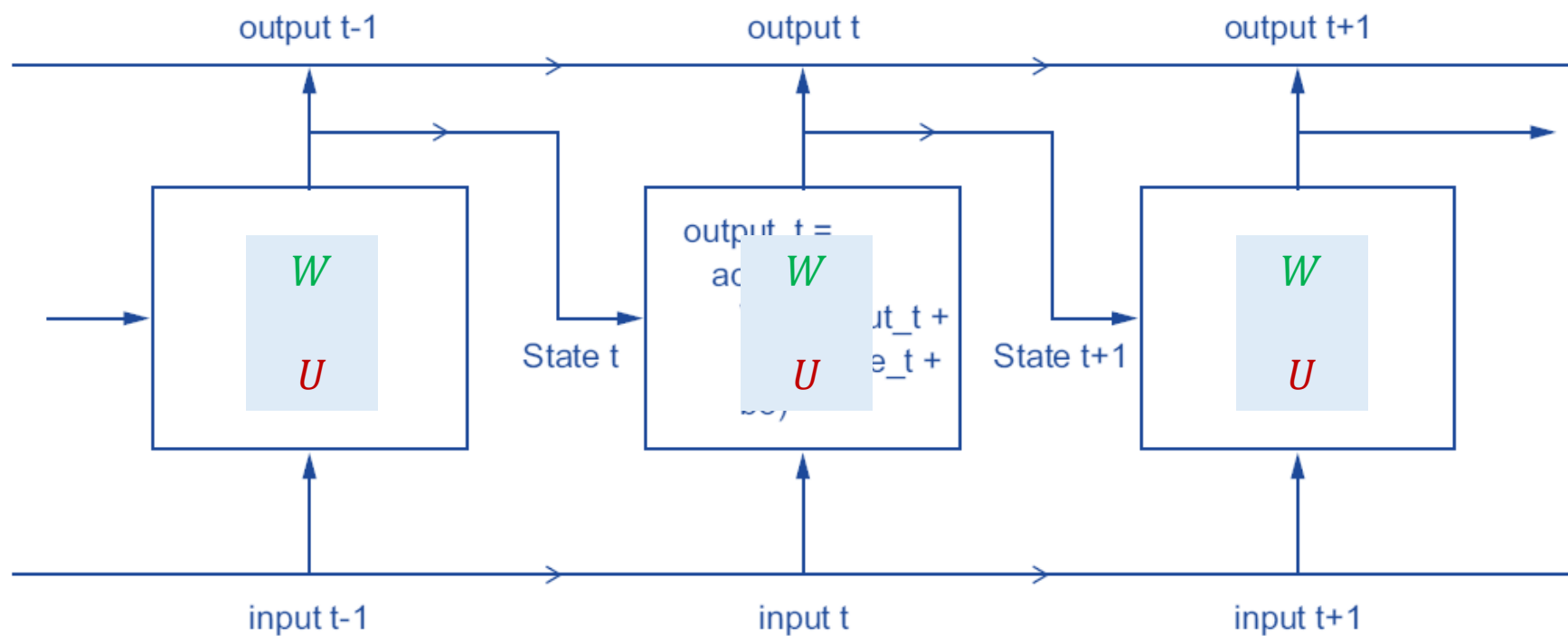


Dense Layer vs Recurrent Layer



→ Inside the Recurrent Cell

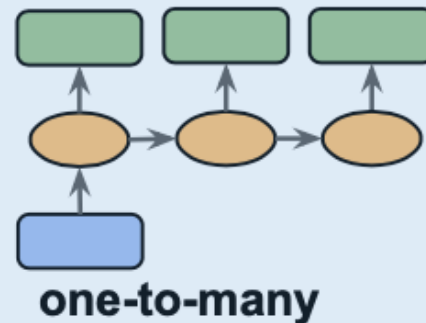
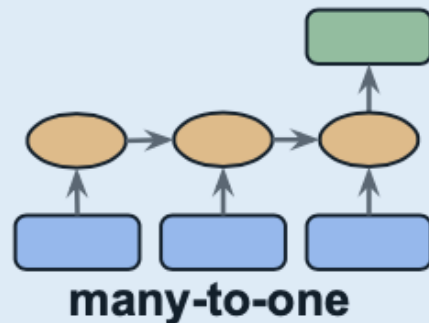
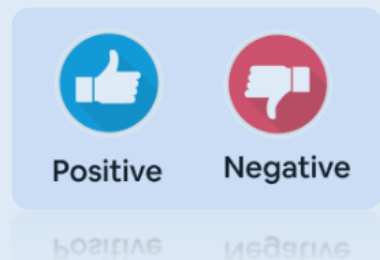
$$output_t = f(input_t, State_t)$$



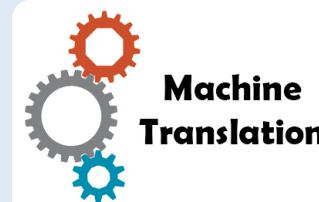
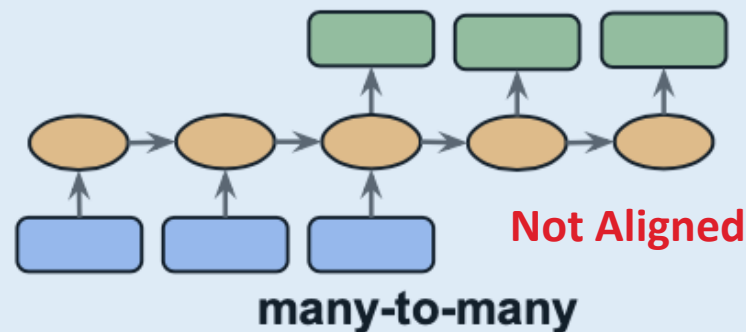
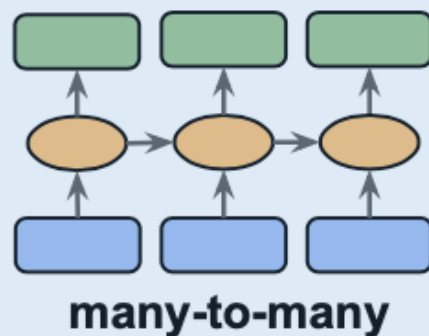
$$s_{t+1} = activation(WX_t + Us_t + b)$$



RNN architectures



a train traveling down a track next to a forest.





How does RNN learn representations?

- Backpropagation Through Time (BPTT)

- $\frac{\partial J}{\partial P}$ P are the parameters

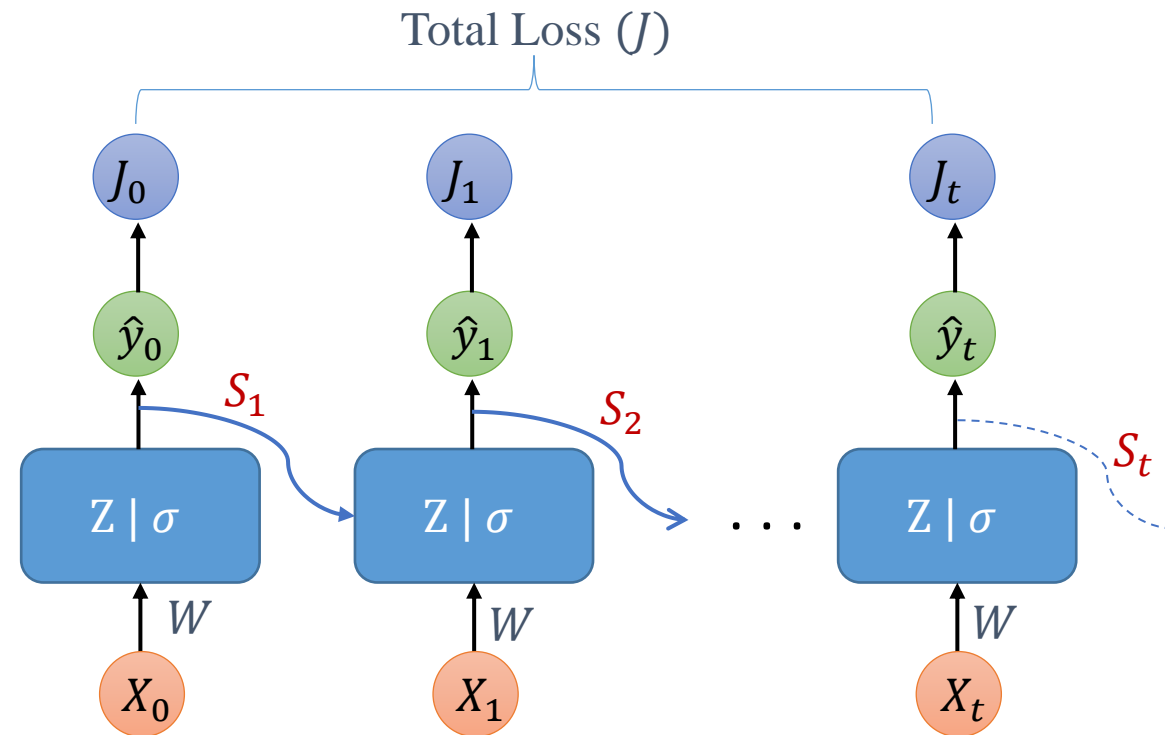
- $\frac{\partial J}{\partial W} = \frac{\partial J_0}{\partial W} + \frac{\partial J_1}{\partial W} + \dots$

- $\frac{\partial J_0}{\partial W} = \frac{\partial J_0}{\partial y_0} \frac{\partial y_0}{\partial s_0} \frac{\partial s_0}{\partial W}$

- $\frac{\partial J_1}{\partial W} = \frac{\partial J_1}{\partial y_1} \frac{\partial y_1}{\partial s_1} \frac{\partial s_1}{\partial W}$, $\frac{\partial s_1}{\partial W} = \frac{\partial s_1}{\partial s_0} \frac{\partial s_0}{\partial W}$

- ...

- $\frac{\partial J_t}{\partial W} = \sum_{k=0}^t \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}$



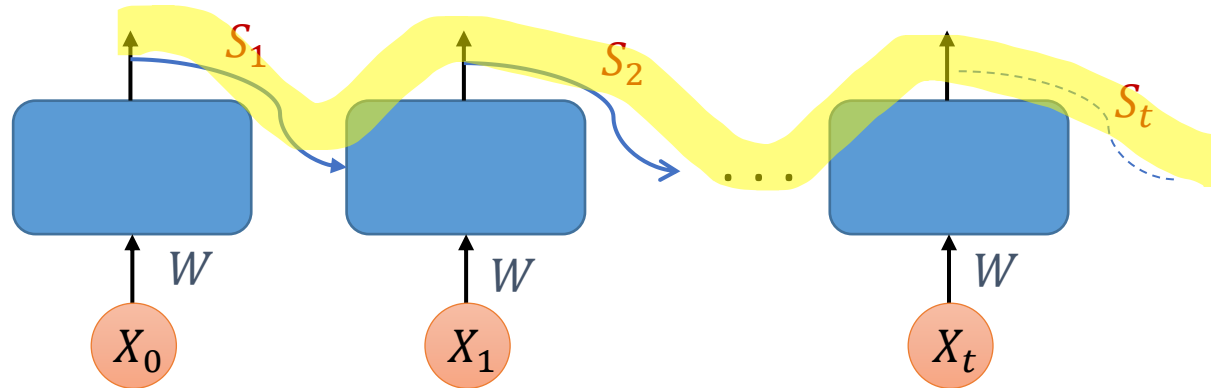
→ Vanishing Gradient Problem

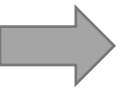
- As the time horizon gets bigger, this product gets longer and longer.
- We are multiplying a lot of small numbers → smaller gradients → biased parameters unable to capture long term dependencies.

$$\bullet \frac{\partial J_t}{\partial W} = \sum_{k=0}^t \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial S_t} \frac{\partial S_t}{\partial S_k} \frac{\partial S_k}{\partial W}$$

$$\bullet \frac{\partial S_{10}}{\partial S_0} = \frac{\partial S_{10}}{\partial S_9} \frac{\partial S_9}{\partial S_8} \frac{\partial S_8}{\partial S_7} \frac{\partial S_7}{\partial S_6} \cdots \frac{\partial S_1}{\partial S_0}$$

$$S_t = \text{activation}(W X_{t-1} + U S_{t-1})$$





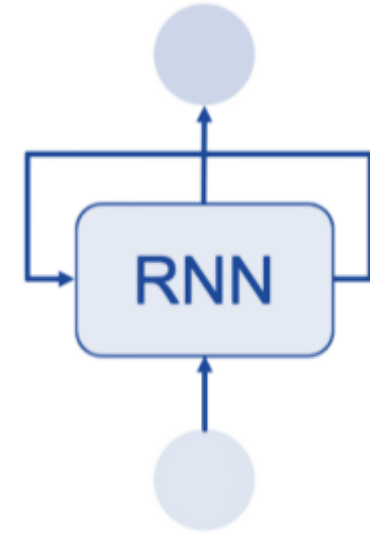
Beyond RNN

RNN can handle the following sequence modeling criteria:

- Preserve the **order**
- Handle **input-length**
- **Share parameters** across the sequence

RNN limitations:

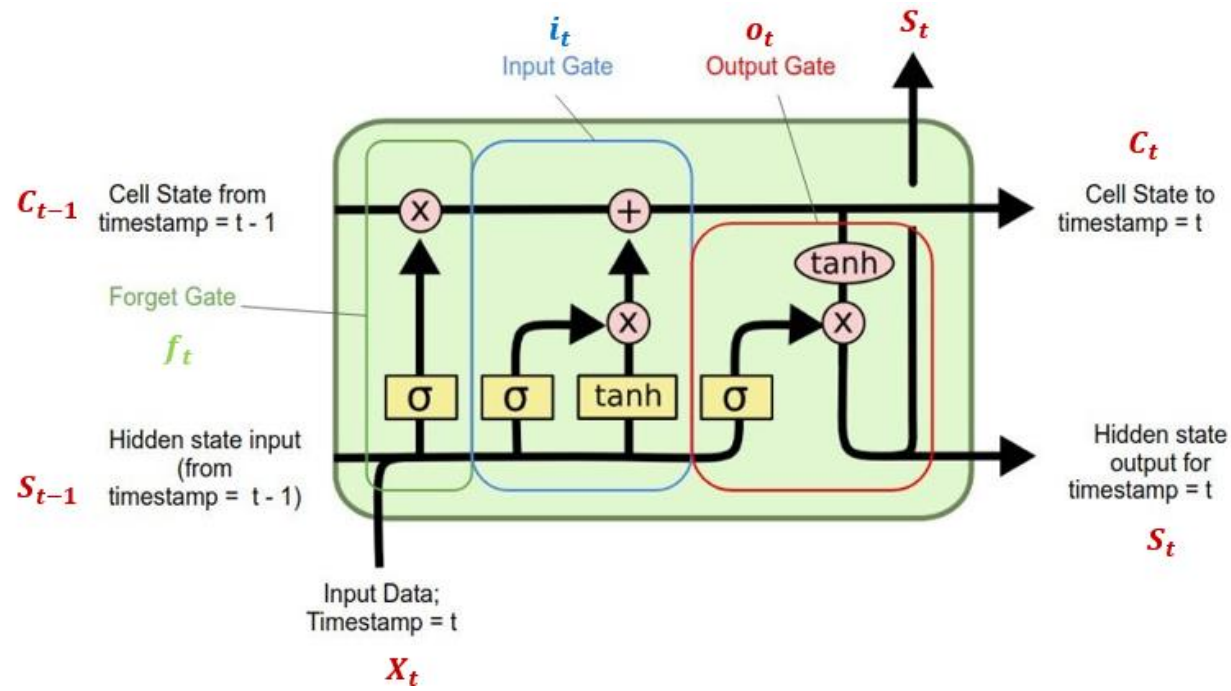
- Does not account for **long-term dependencies** (only remember short term history)
- Vanishing Gradient Problem

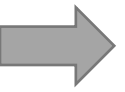


Module 6 – Part II

Deep Sequence Modeling

(Gated cells, LSTM)

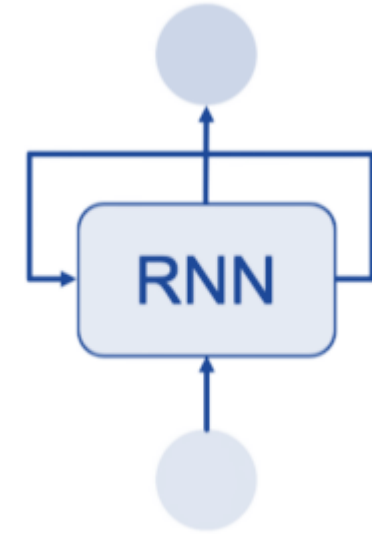




Beyond RNN

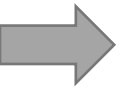
RNN can handle the following sequence modeling criteria:

- Preserve the **order**
- Handle **input-length**
- **Share parameters** across the sequence



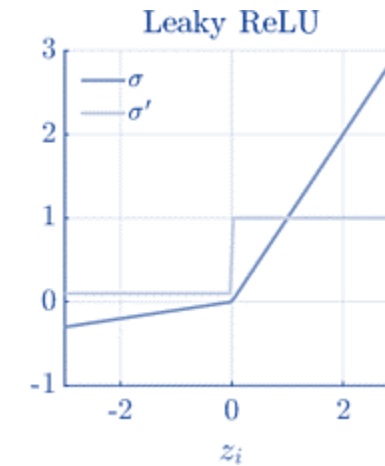
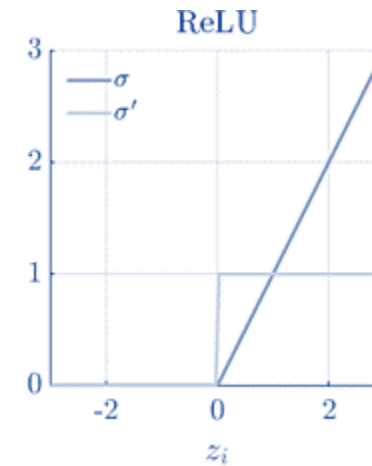
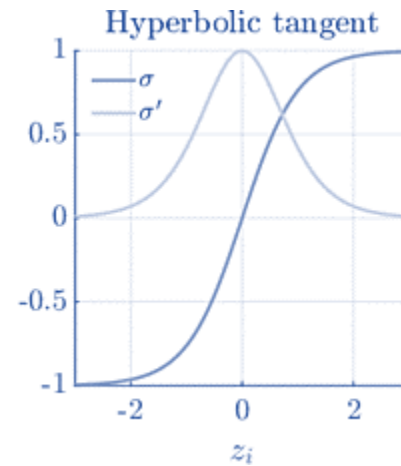
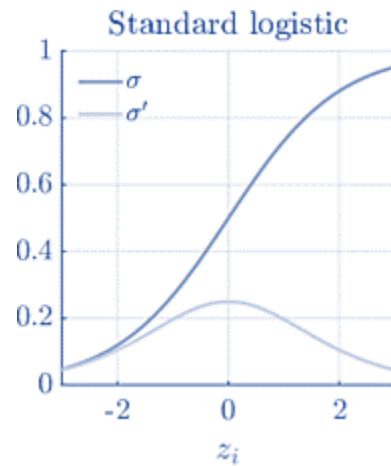
RNN limitations:

- Does not account for **long-term dependencies** (only remember short term history)
- Vanishing Gradient Problem



How to solve vanishing gradient problem

1. Use **Activation Function** that prevents fast shrinkage of gradient

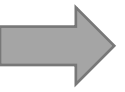


$$S_t = \text{activation}(\textcolor{green}{W}X_{t-1} + \textcolor{red}{U}s_{t-1})$$



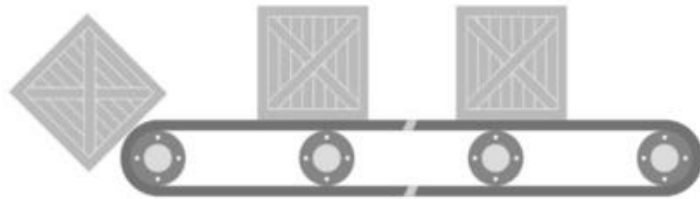
How to solve vanishing gradient problem

1. Use **Activation Function** that prevents fast shrinkage of gradient
2. Use **weight initialization** techniques that ensure that the initial weights are not too small
3. Use **gradient clipping** which limits the magnitude of the gradients from becoming too small (vanishing gradient) or too large (exploding gradient)
4. Use **batch normalization**, which normalizes the input to each layer and helps to reduce the range of activation values and thus the likelihood of vanishing gradients.
5. Use a different **optimization algorithm** that is more resilient to vanishing gradients, such as Adam or RMSprop.
6. **Gated cells:** Use some sort of **skip connections**, which allow gradients to bypass some of the layers in the network and thus prevent them from becoming too small.



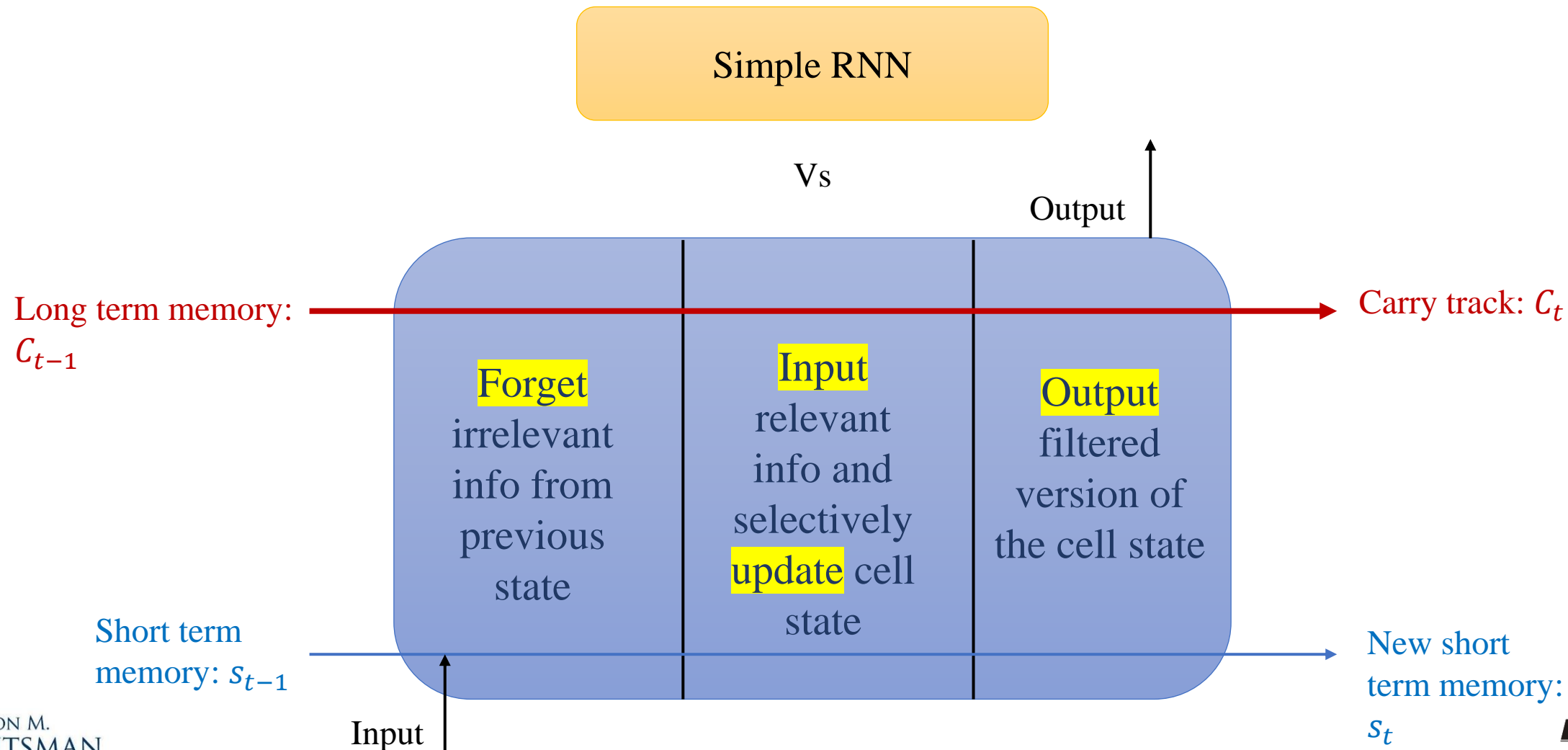
Gated cells

- Instead of using a simple RNN cell, let's use a **more complex cell** with gates which **control the flow of information**.
- Think of a **conveyor belt** running parallel to the sequence being processed:
 - Information can jump on → transported to a later timestep → jump off when needed.
 - This is what a gated cell does! Analogous to **residual connections** we saw before.



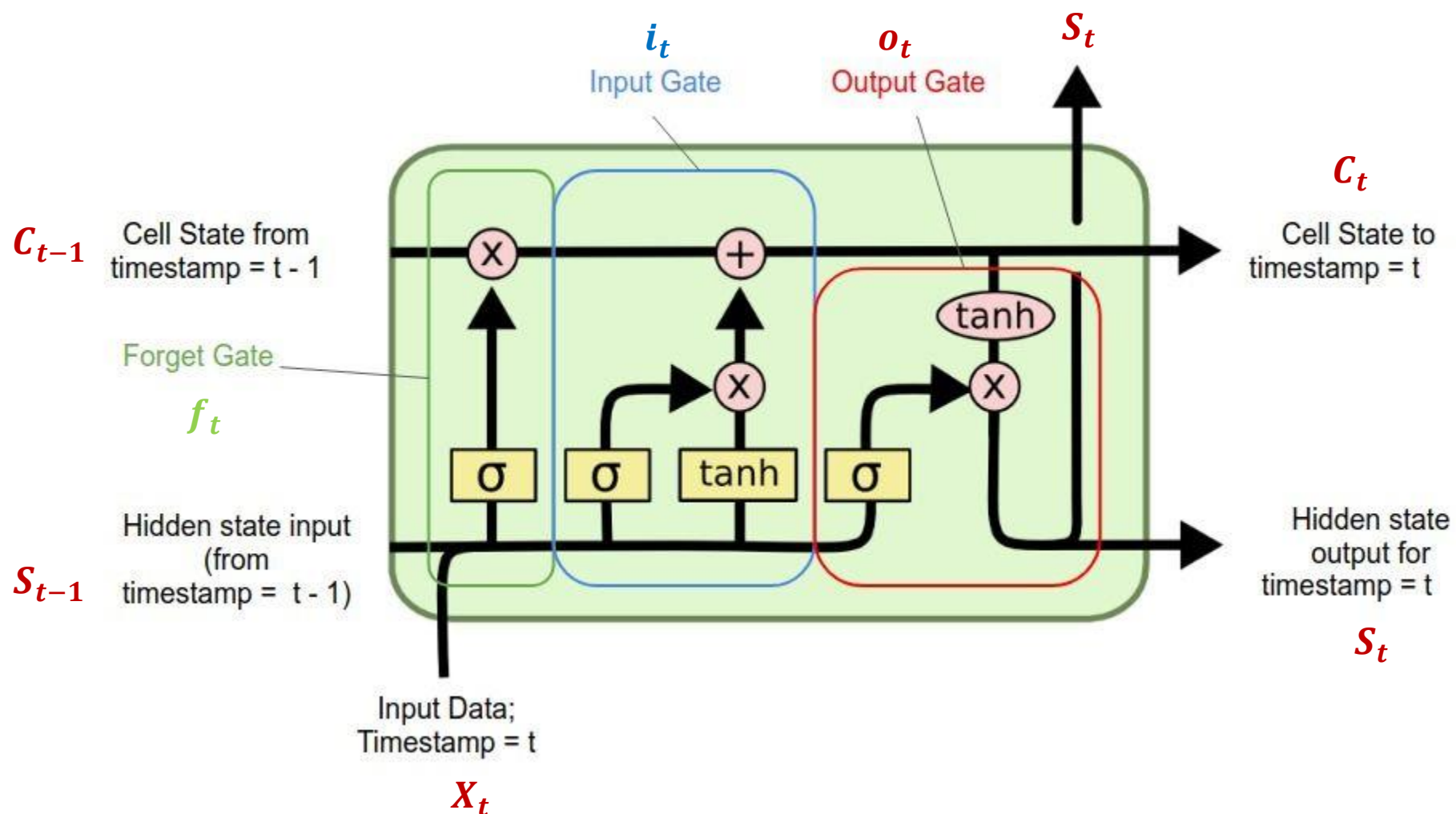
- Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two examples of gated cells that can **keep track of information throughout many timesteps**.

→ Inside the LSTM cell





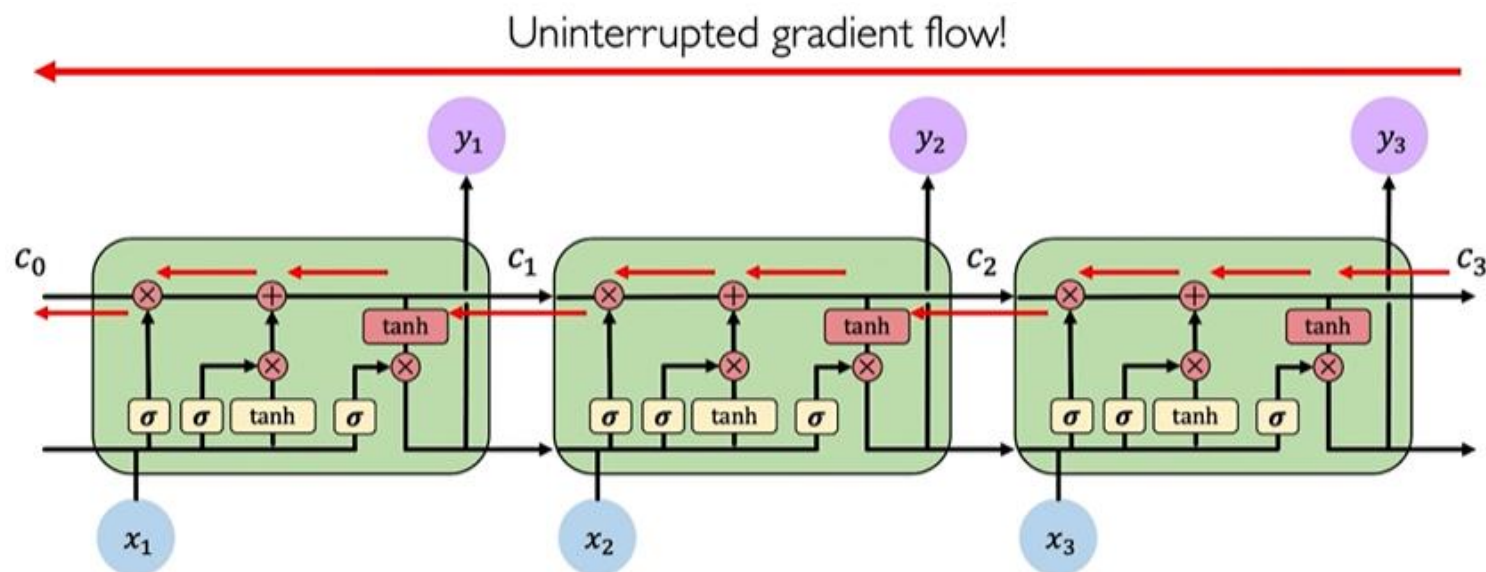
LSTM details

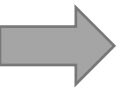




LSTM takeaway

- LSTM uses gates to **regulate the information flow** (allows past information to be reinjected later)
- This new cell state (carry) can better capture **longer term dependencies**
- LSTM **fights** the vanishing gradient problem



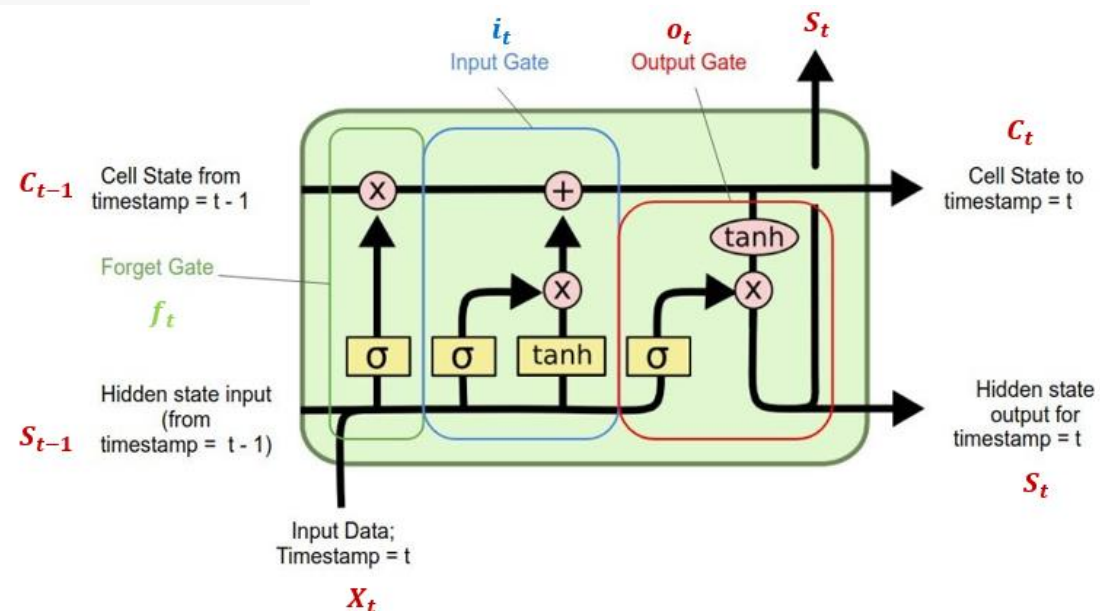


Let's try LSTM on the temperature example

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 120, 14)]	0
lstm (LSTM)	(None, 16)	1984
dense_3 (Dense)	(None, 1)	17

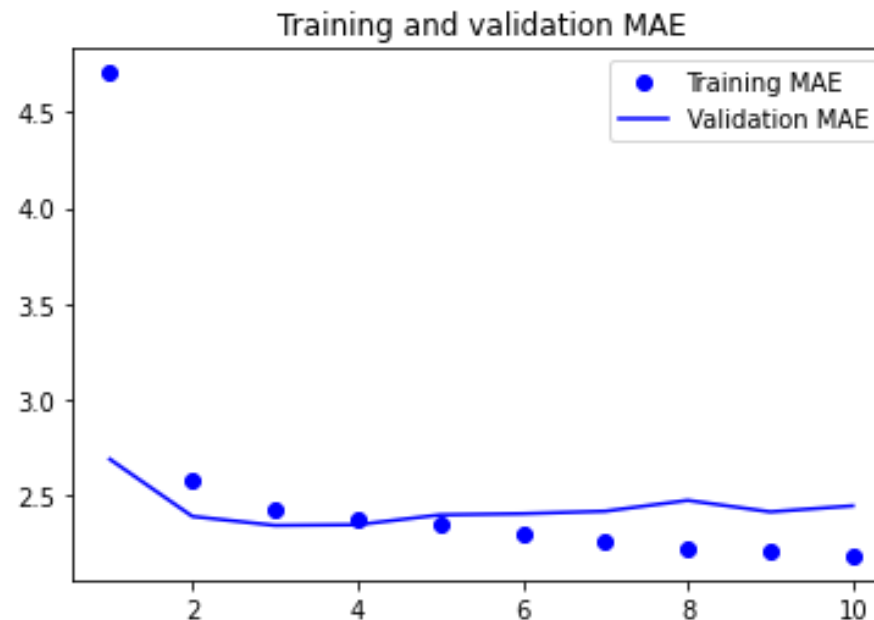
=====
Total params: 2,001
Trainable params: 2,001
Non-trainable params: 0
=====



```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

➔ LSTM performance

- Baseline Test MAE = 2.62
- Simple LSTM Test MAE = 2.53
- Finally beat the naïve forecaster.
- Overfitting?



Can we do better?



Improving the simple LSTM model

- We can improve the performance of the simple LSTM model by:
 1. **Recurrent Dropout** : use drop out to fight overfitting in the recurrent layers (in addition to drop out for the dense layers)
 2. **Stacking recurrent layers**: increase model complexity to boost representation power
 3. **Using bidirectional RNN**: processing the same information differently! Mostly used in NLP.

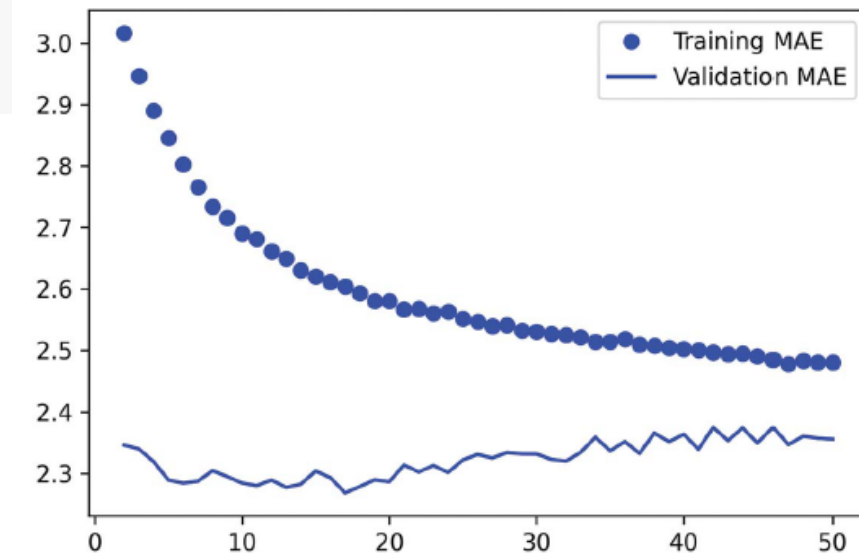


→ Recurrent Drop out

- The **same dropout pattern** should be applied at every timestep

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

- Baseline Test MAE = 2.62
- Simple LSTM, Test MAE = 2.53
- LSTM with dropout, Test MAE = 2.45

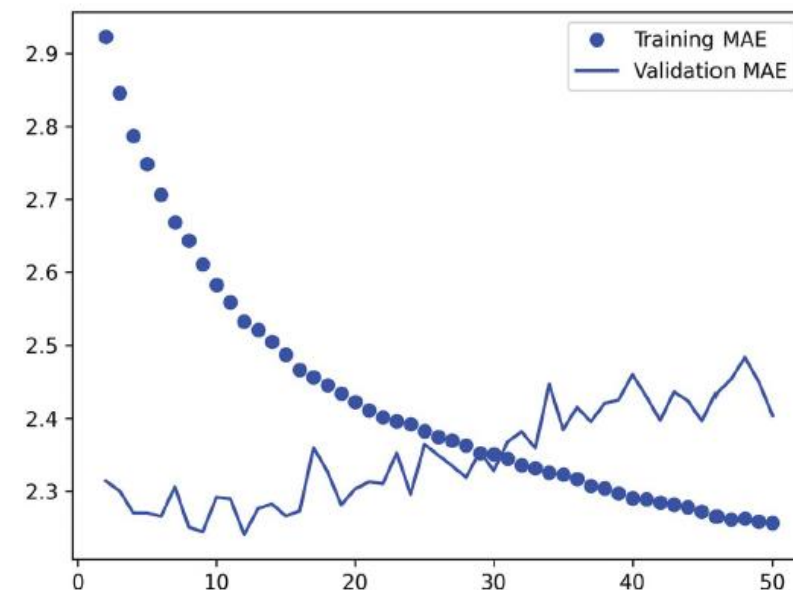


→ Stacking Recurrent Layers

- Let's train a dropout-regulated, stacked GRU model.
- GRU is a slightly simpler version (hence, faster) of LSTM architecture

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```

- Baseline Test MAE = 2.62
- Simple LSTM, Test MAE = 2.53
- Stacking GRU, Test MAE = 2.39

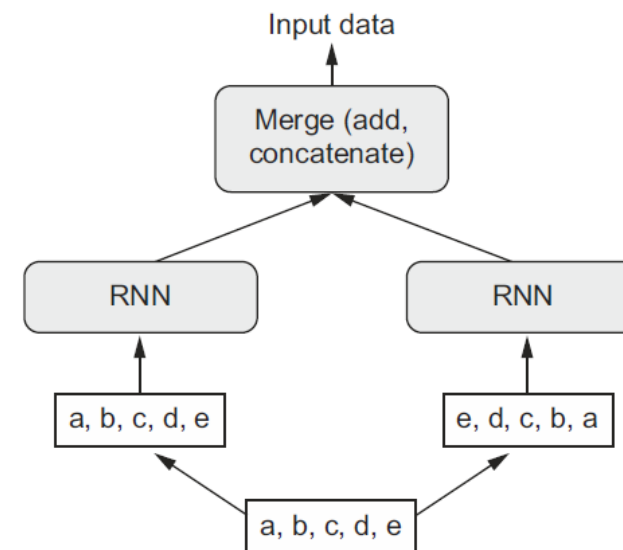


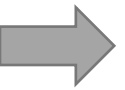


Bidirectional RNN

- Bidirectional RNN process the input sequence both chronologically and antichronologically.
- Idea: capturing patterns (representations) that might be overlooked by a unidirectional RNN.
- For the temperature example, the bidirectional LSTM strongly underperforms even the common-sense baseline.

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
```





Final message

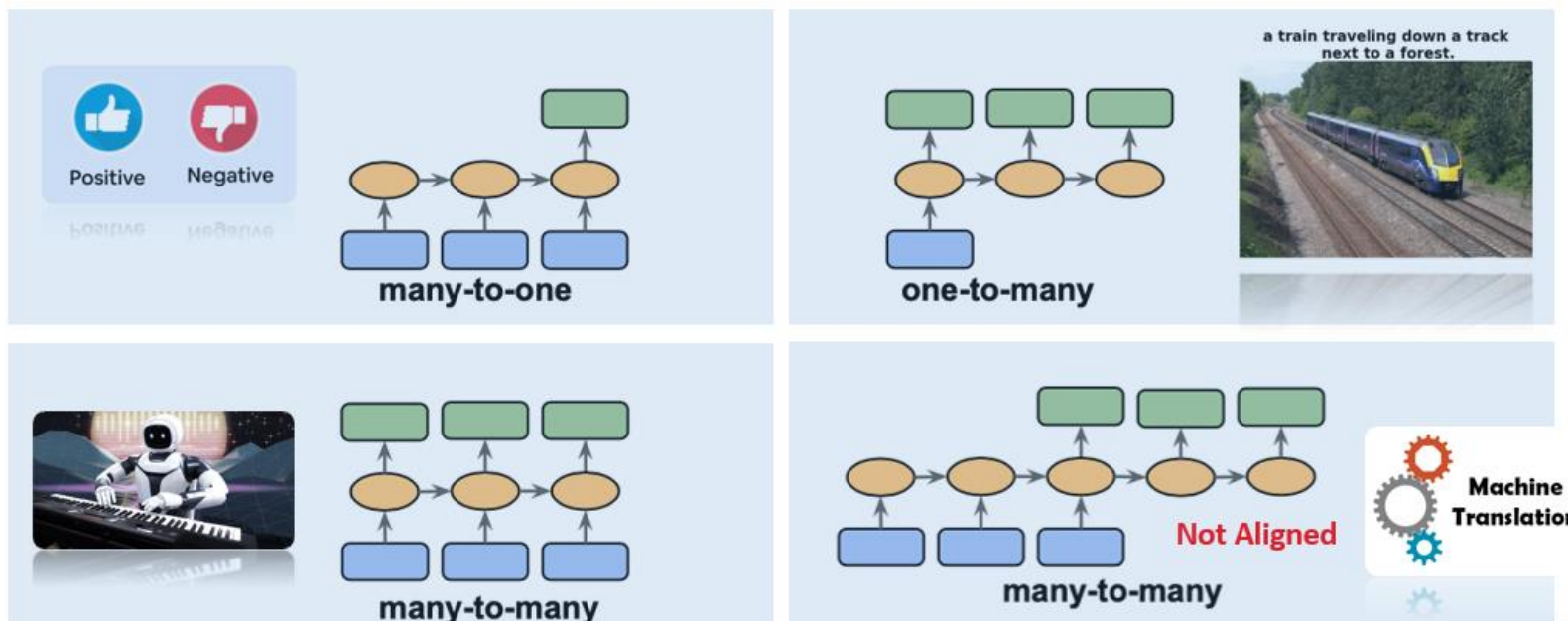
- Deep learning is more an art than science! Too many moving part!
 - Number of units in each recurrent layer
 - Number of stacked layers
 - Amount of dropout and recurrent dropout
 - Number of dense layers
 - Sequence horizon!
 - Optimizers, learning rates and etc
 -
- Apply RNN to datasets that past is a good predictor of the future! **Not the stock market!**

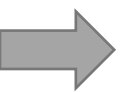


Module 6 – Part III

Deep Sequence Modeling for Text

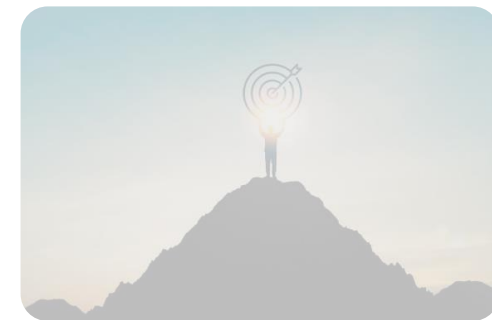
Natural Language Processing (NLP)

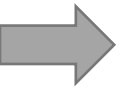




Road map!

- Module 1- Introduction to Deep Learning
- Module 2- Setting up Deep Learning Environment
- Module 3- Machine Learning review (ML fundamentals + models)
- Module 4- Deep Neural Networks (NN and DNN)
- Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- **Module 6- Deep Sequence Modeling (RNN, LSTM, NLP)**
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)





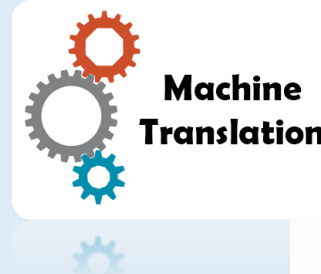
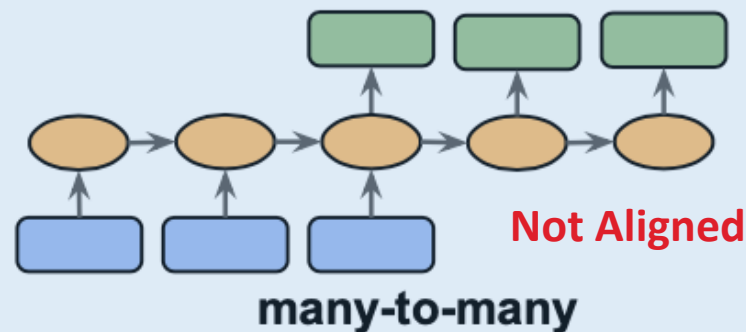
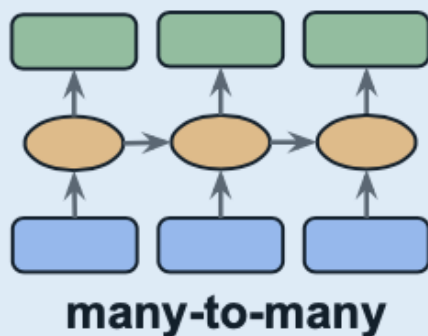
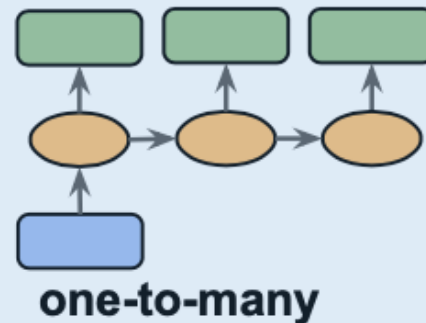
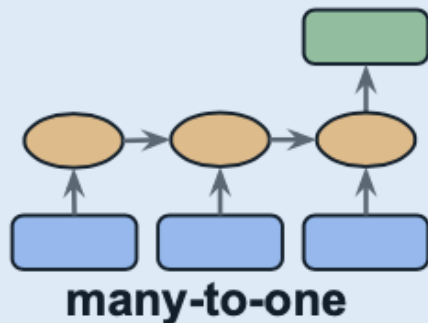
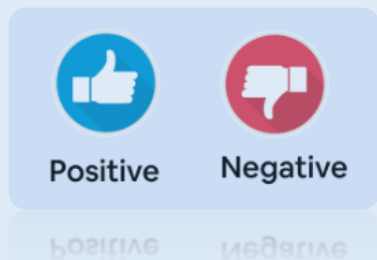
Different kinds of sequence data

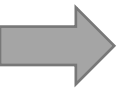
Sequence data refers to any data that has a specific order or sequence to it!

- **Time series data:** Time series data is a sequence of data points that are measured at regular intervals over time. (stock prices, weather patterns, medical records, ...)
- **Text data:** Text data refers to any type of data that is composed of words, sentences, or paragraphs. (tweets, news articles, product reviews, ...)
- **Audio data:** Audio data refers to any type of data that is recorded or generated as sound waves. (speech recordings, music tracks, ...)
- **Video data:** Video data refers to any type of data that is represented as a sequence of images or frames. (movie clips, surveillance footage, ...)



RNN architectures





Human language vs machine language

- In compute science: **Human** language → **Natural** language (shaped by an evolution process)
- **Machine** language designed for machines (XML, Assembly, ...)



Factors	Human Language	Machine Language
Complexity	Highly complex	Structured and logical
Creativity	Can be highly creative	Limited creativity
Ambiguity	High levels of ambiguity	Little to no ambiguity
Learning	Acquired naturally	Taught through programming
Adaptability	Highly adaptable	Fixed and rigid

➔ Natural Language Processing

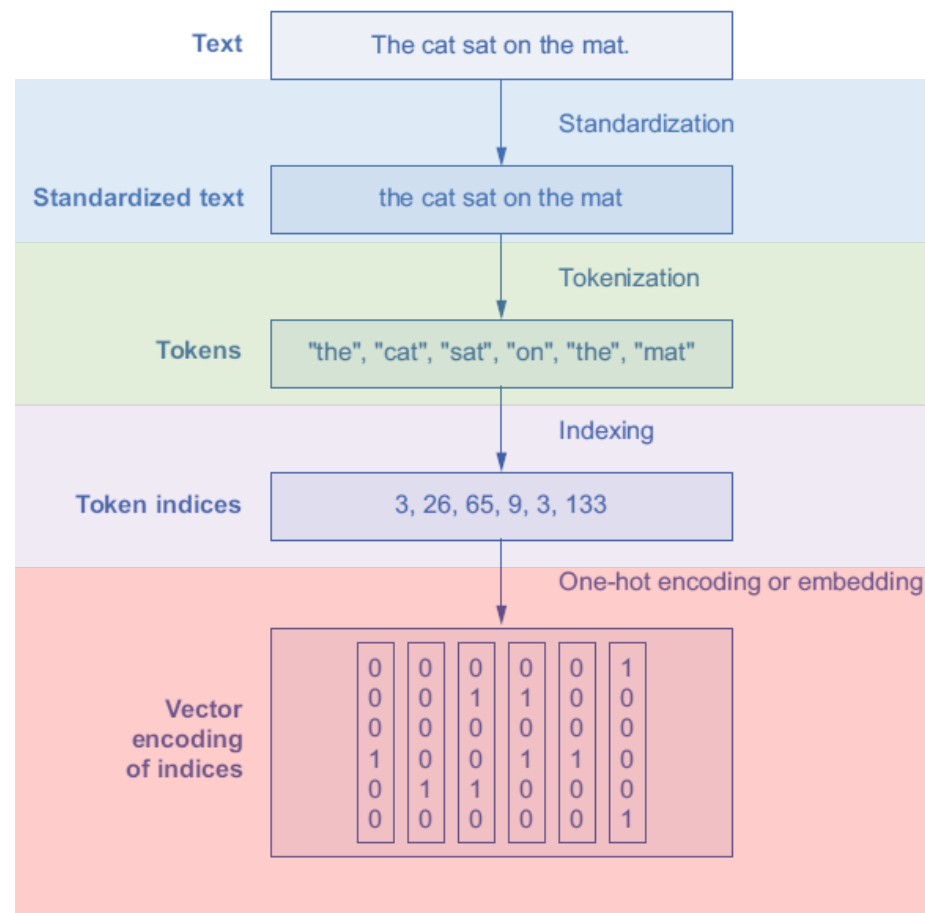
- Making sense of human language through algorithms is a big deal!
- NLP is a subfield of computer science and AI that deals with the interaction between computers and human language
- Traditional NLP started with finding **handcrafted rules** to **understand** human language (1960-1990)
- Modern NLP is all about using machine learning to automate the search for these rules and enabling computers to do the followings:
 - Sentiment analysis
 - Machine translation
 - Content filtering
 - Text classification and ...



Preparing text data

- **Text Vectorization**: the process of transforming text into numeric values.

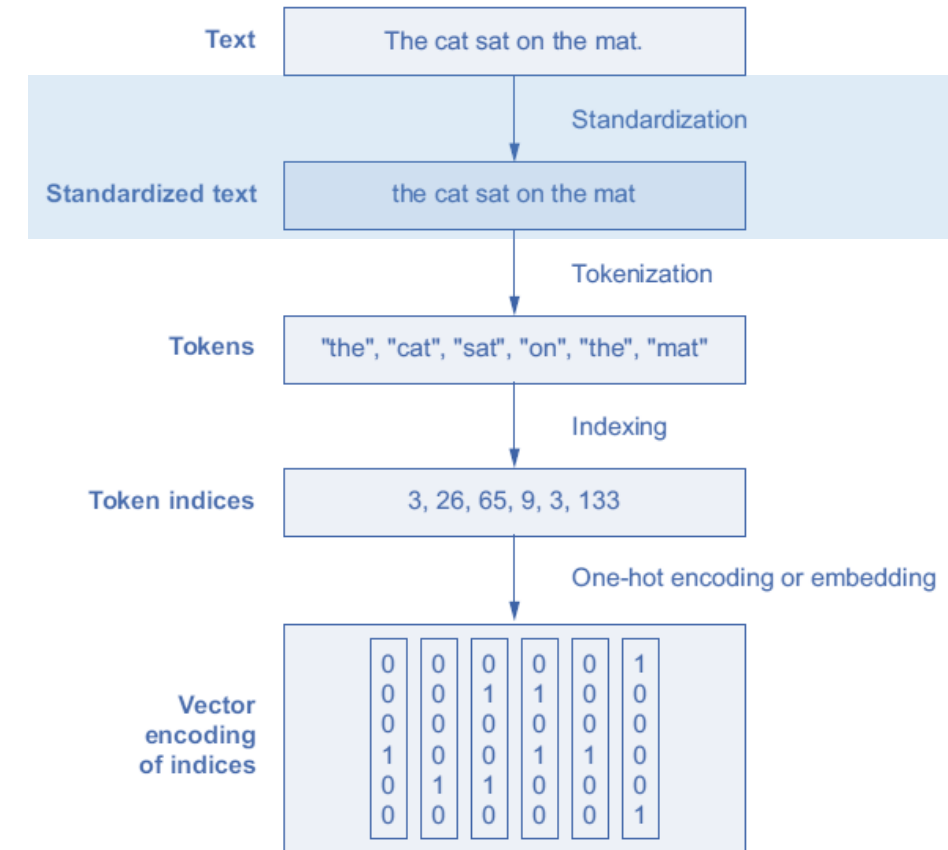
1. Standardization
2. Tokenization
3. Indexing
4. Encoding





Text Vectorization: **standardization**, tokenization, indexing, encoding

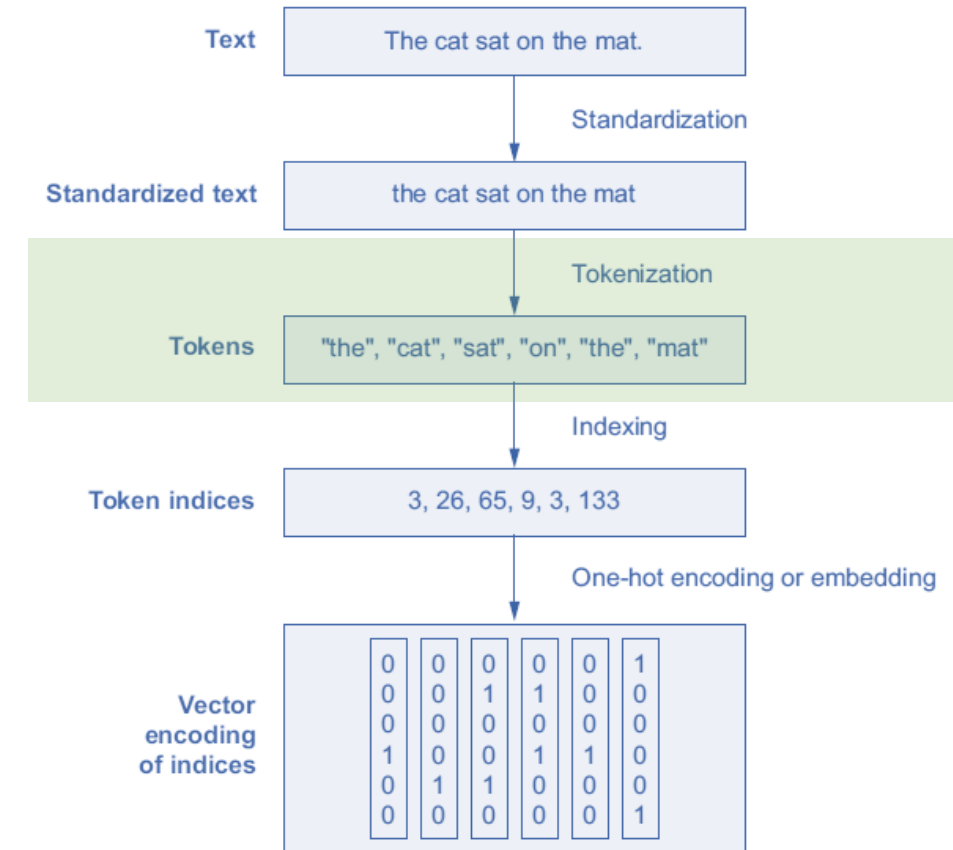
- **Standardization:**
 - Converting to *lowercase*,
 - Removing *punctuations*,
 - Converting *special characters*,
 - **Stemming** (converting variations of a term into a single shared representation)
- Standardization is a basic form of **feature engineering**
- With standardization, your model requires **less training data** and **generalize better**.





Text Vectorization: standardization, **tokenization**, indexing, encoding

- **Tokenization: Splitting text into units/token**
 - **Word-level** tokenization: space-separated tokens
 - **N-gram** tokenization: groups of N (or fewer) consecutive words
 - **Character-level** tokenization: each character is a token
- Two kinds of text processing models:
 - **Bag-of-words** models (discarding original order)
→ N-gram tokenization
 - **Sequence models** (word order matters)
→ **word-level** tokenization

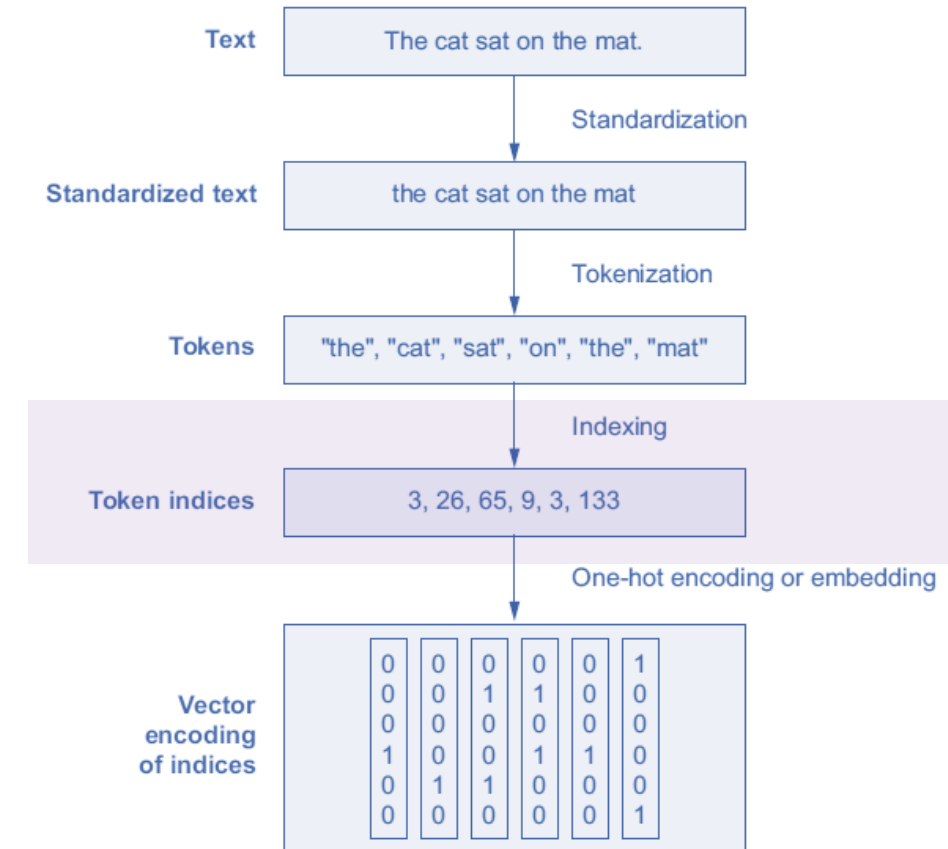




Text Vectorization: standardization, tokenization, **indexing**, encoding

- **Indexing:**

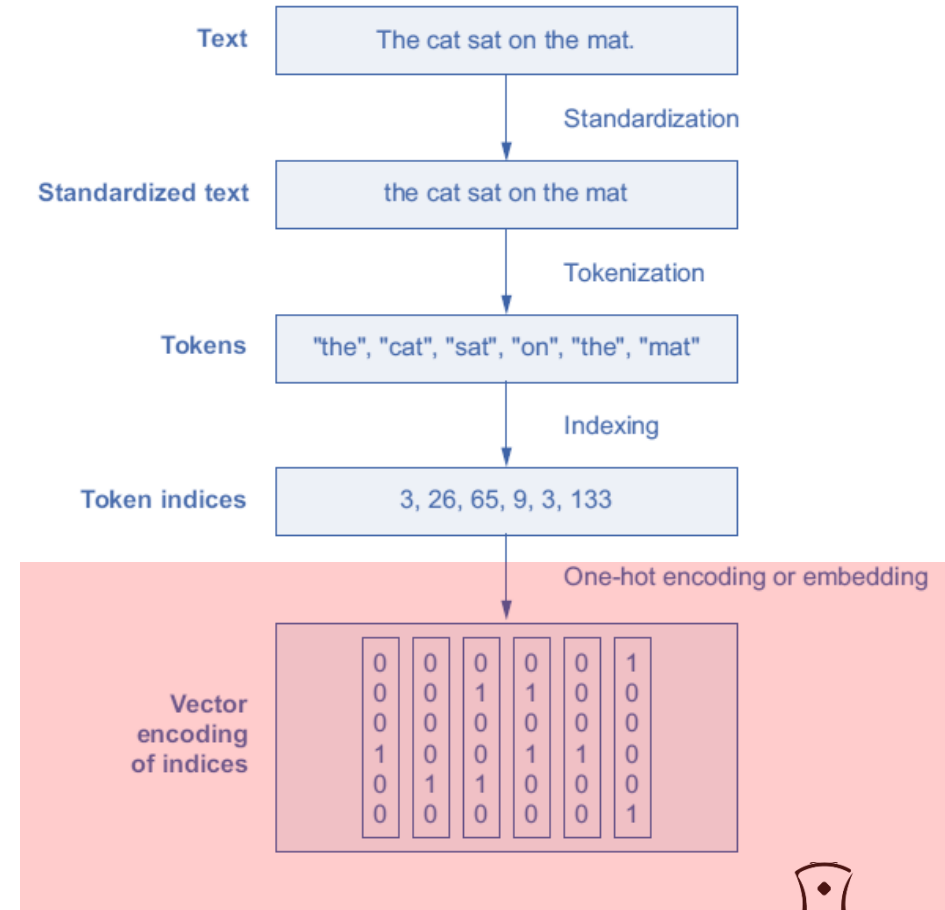
- **Integer**: return sequences of words encoded as integer indices
- Restrict to top 20k or 30k most common words
- **OOV** token (out of vocabulary) index **1**
- **Mask** token index **0** (ignore it, used for padding)



Text Vectorization: standardization, tokenization, indexing, encoding

- **Encoding:**

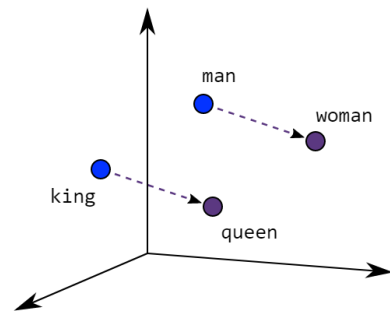
- **Multi-hot (binary):** Encode the tokens as a multi-hot binary vector
- **One-hot:** Encode the tokens as one-hot binary vectors
- **TF-IDF:** term frequency, inverse document frequency



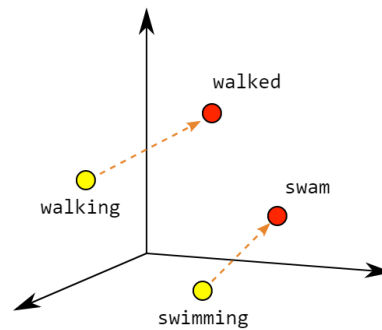


Word Embedding

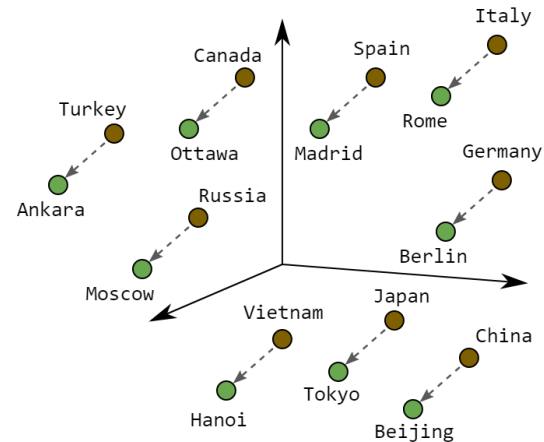
- Word embeddings are **dense**, **structured** representations **learned from data**.



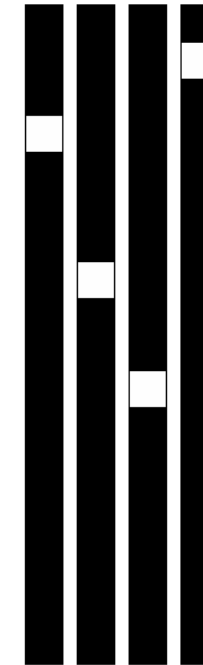
Male-Female



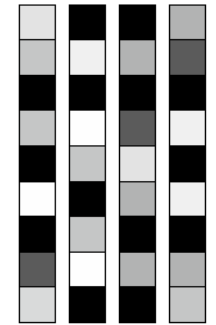
Verb Tense



Country-Capital



One-hot word vectors:
- Sparse
- High-dimensional
- Hardcoded



Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

→ Representing groups of words

	Bag-of-Words Approach	Sequential Modeling Approach
Input	Collection of text documents	Sequence of words
Output	Vector representation of each document	Sequential output, e.g. text classification, machine translation
Order	Ignores the order of words	Captures the order of words
Context	Ignores the context of words	Captures the context of words
Features	Each word is a feature	Features depend on the model architecture
Dimensionality	High dimensionality	Lower dimensionality
Sparsity	High sparsity	Lower sparsity
Performance	Faster training and inference	Slower training and inference
Suitability	Good for simple tasks like document classification	Good for complex tasks like language modeling and machine translation



A simple example

- Historically, most early applications of machine learning to NLP involved bag-of-words models.
- Interest in sequence models started rising in 2015, with the rebirth of RNN
- Today, both approaches remain relevant. Let's look at a simple example: [IMDB movie review](#)
- Task: sentiment-classification of IMDB movie reviews (positive-negative)





IMDB review example (bag-of-word approach)

- Single words (**Unigram**) with binary encoding (**multi-hot**)
- You can represent the entire text as a single vector!
- The cat sat on the mat \rightarrow {"cat", "mat", "sat", "on", "the"} \rightarrow (0,0,1,...,1,0,...,1,0,...,1,..1)
- Bag-of-words model is mainly made of **dense layers** (No recurrent layers)

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(max_tokens=20000, hidden_dim=16):
    inputs = keras.Input(shape=(max_tokens,))
    x = layers.Dense(hidden_dim, activation="relu")(inputs)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])

    return model
```

```
text_vectorization = TextVectorization(
    max_tokens=20000,
    output_mode="multi_hot",
```

- Naïve baseline = **50%** (balance data)
- Test accuracy = **89.2%**





IMDB review example (improving bag-of-word approach)

- We started with Single words (**Unigram**) with binary encoding (**multi-hot**)
- Two words (**Bigrams**) with binary encoding (**multi-hot**): adding **local order** information
- The cat sat on the mat \rightarrow {"the", "the cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the mat", "mat"}

Test accuracy

- Naïve baseline = **50%** (balance data)
- Unigram binary = **89.2%**
- Bigram binary = **90.4%**

```
text_vectorization = TextVectorization(  
    ngrams=2,  
    max_tokens=20000,  
    output_mode="multi_hot",
```



IMDB review example (improving bag-of-word approach)

- Bigrams with TF-IDF encoding

```
text_vectorization = TextVectorization(  
    ngrams=2,  
    max_tokens=20000,  
    output_mode="tf_idf",
```

Test accuracy

- Naïve baseline = 50% (balance data)
- Unigram binary = 89.2%
- Bigram binary = 90.4%
- Bigram TF-IDF = 89.8%
- Typically, TF-IDF increases the model performance by 1%. It wasn't the case for the IMDB dataset.

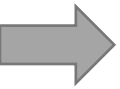


IMDB review example (sequence modeling approach)

1. Starting by representing the input as sequences of **integer** indices (one integer for one word)
2. Map each integer to a vector to get **sequences of vectors** (one-hot or word embedding)
3. Finally, feed these vectors into a stack of layers such as **1D Convnet**, **RNN** or a **Transformer**.

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
```



IMDB review example (sequence modeling approach)

- One-hot encoding approach: Encode the integers into binary 20,000-dimensional vectors.

```
import tensorflow as tf
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = tf.one_hot(inputs, depth=max_tokens)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

Test accuracy

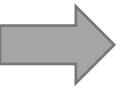
- Naïve baseline = 50% (balance data)
 - Unigram binary = 89.2%
 - Bigram binary = 90.4%
 - Bigram TF-IDF = 89.8%
 - Sequence one-hot encode = 87%
- This model trains very slowly. $600 \times 20,000 = 12$ million floats for a single movie review!
 - There must be a better way than one-hot encoding!



IMDB review example (improving sequence modeling approach)

- **Word-embedding** approach:
 1. **Simultaneously** trained word embedding (trained jointly with the weights of a NN)
 2. **Pretrained** word embedding
- Sometimes it is reasonable to learn a new embedding space for different tasks (for example movie-review is different from legal-document classification)
- Word index → word embedding layer → corresponding word vector

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

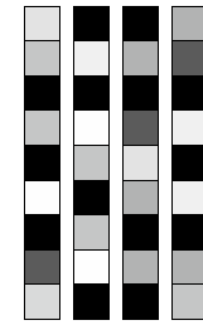


IMDB review example (improving sequence modeling approach)

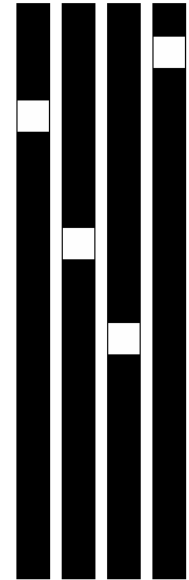
- **Word-embedding** approach performance:
- It trains much faster than the one-hot encoding (256 vs 20k dim)

Test accuracy

- Naïve baseline = **50%** (balance data)
- Unigram binary = **89.2%**
- Bigram binary = **90.4%**
- Bigram TF-IDF = **89.8%**
- Sequence one-hot encode = **87%**
- Sequence word embedding simultaneously trained = **87%**
- Sequence word embedding simultaneously trained with **masks** = **88%**



Word embeddings:
- Dense
- Lower-dimensional
- Learned from data



One-hot word vectors:
- Sparse
- High-dimensional
- Hardcoded

➔ Pretrained word embedding

- Motivation: little training data! The network cannot learn an appropriate embedding.
- Analogous to using pretrained convnets in image classification
- Useful when expecting **generic** features.
- Schemes:
 - The **Word2Vec** algorithm (Tomas MiKolov at Google 2013): a **predictive model** that uses a neural network to predict the **probability of a word given its context**
 - **GloVe** (Stanford researchers 2014): a **count-based model** that computes the **co-occurrence probabilities** between words in a corpus
- Word2Vec is faster to train than GloVe, especially for large corpora



Road map!

- ✓ Module 1- Introduction to Deep Learning
- ✓ Module 2- Setting up Deep Learning Environment
- ✓ Module 3- Machine Learning review (ML fundamentals + models)
- ✓ Module 4- Deep Neural Networks (NN and DNN)
- ✓ Module 5- Deep Computer Vision (CNN, R-CNN, YOLO, FCN)
- ✓ Module 6- Deep Sequence Modeling (RNN, LSTM, NLP)
- Module 7- Transformers (Attention is all you need!)
- Module 8- Deep Generative Modeling (AE, VAE, GAN)
- Module 9- Deep Reinforcement Learning (DQN, PG)

