Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica

Lesson 6

Neural Networks for NLP

Nicola Capuano and Antonio Greco

DIEM – University of Salerno

Outline

- Recurrent Neural Networks
- RNN Variants
- Building a Spam Detector
- Intro to Text Generation
- Building a Poetry Generator

Recurrent Neural Networks

Neural Networks and NLP

Neural networks are widely **used in text processing**

- A limitation of **feedforward networks** is the **lack of memory**
- Each input is **processed independently**, without maintaining any **state**
- To process a text, **the entire sequence of words must be presented at once**

The entire text must become a single data

- This is the approach used with **BoW** or **TF-IDF** vectors
- A similar approach is **averaging the Word Vectors** of a text

Neural Networks with Memory

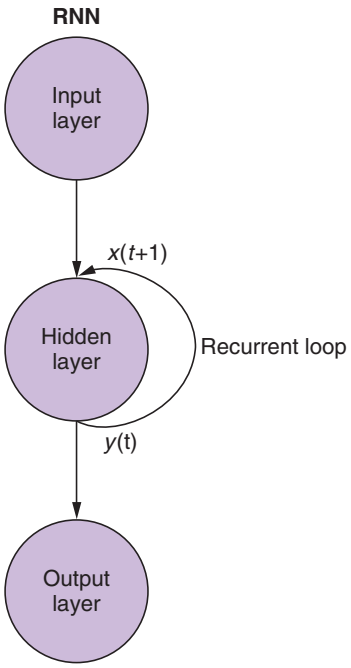
When you **read a text**, you:

- process sentences and words **one by one**
- maintain the **memory** of what you have read previously
  - An **internal model is continuously updated** as new information arrives

A **Recurrent Neural Network** adopts the same principle

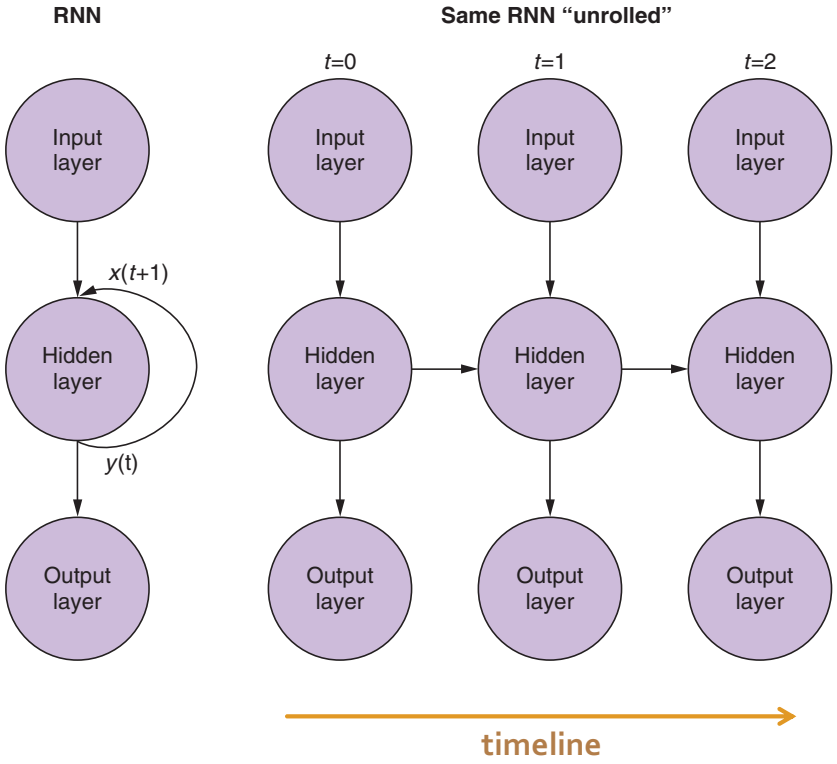
- It processes **sequences** of information by **iterating on the elements** of the sequence
  - E.g., the **words of a text** represented in the form of word embeddings
- It **maintains a state** containing information about what has been seen so far

Recurrent Neural Networks

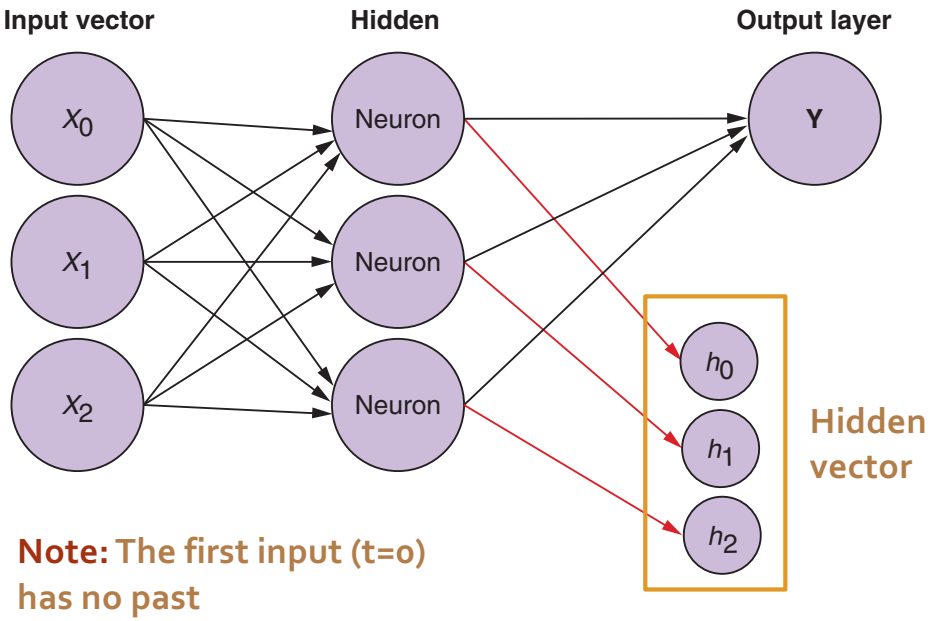


- Circles are **feedforward network layers** composed of one or more neurons
- The **output of the hidden layer** emerges from the network normally to the **output layer**
- In addition, it is also **fed back as input** to the **hidden layer** along with the normal input in the **next time step**

Unrolling the RNN

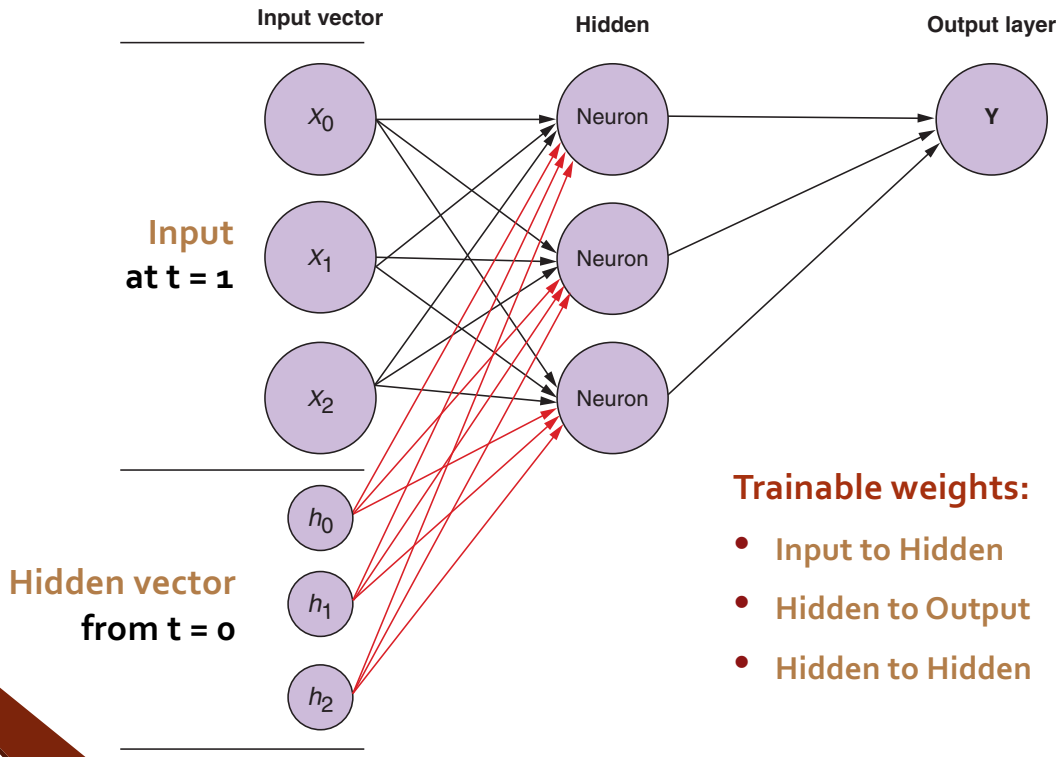


Inside the RNN (t = o)



Note: The first input (t=o) has no past

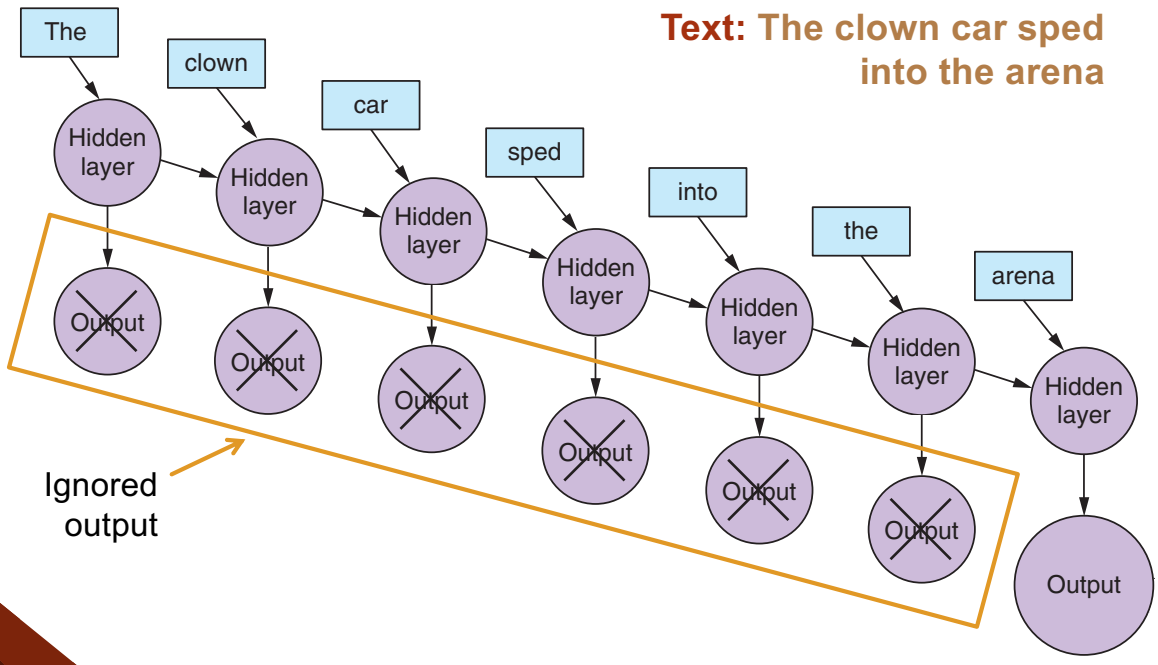
Inside the RNN (t = 1)



Trainable weights:

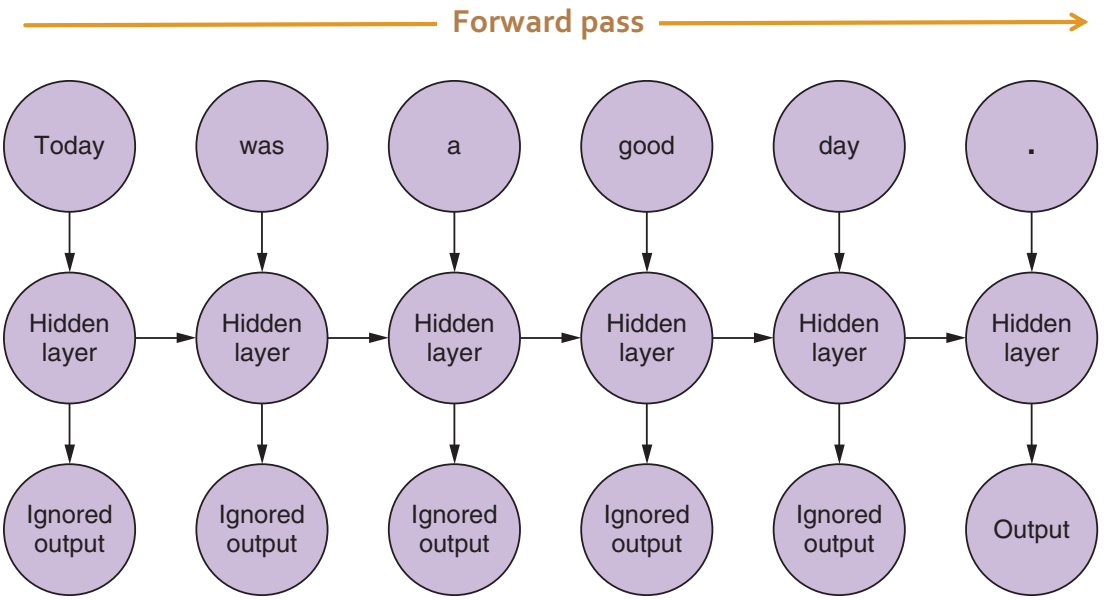
- Input to Hidden
- Hidden to Output
- Hidden to Hidden

Using RNNs with Text



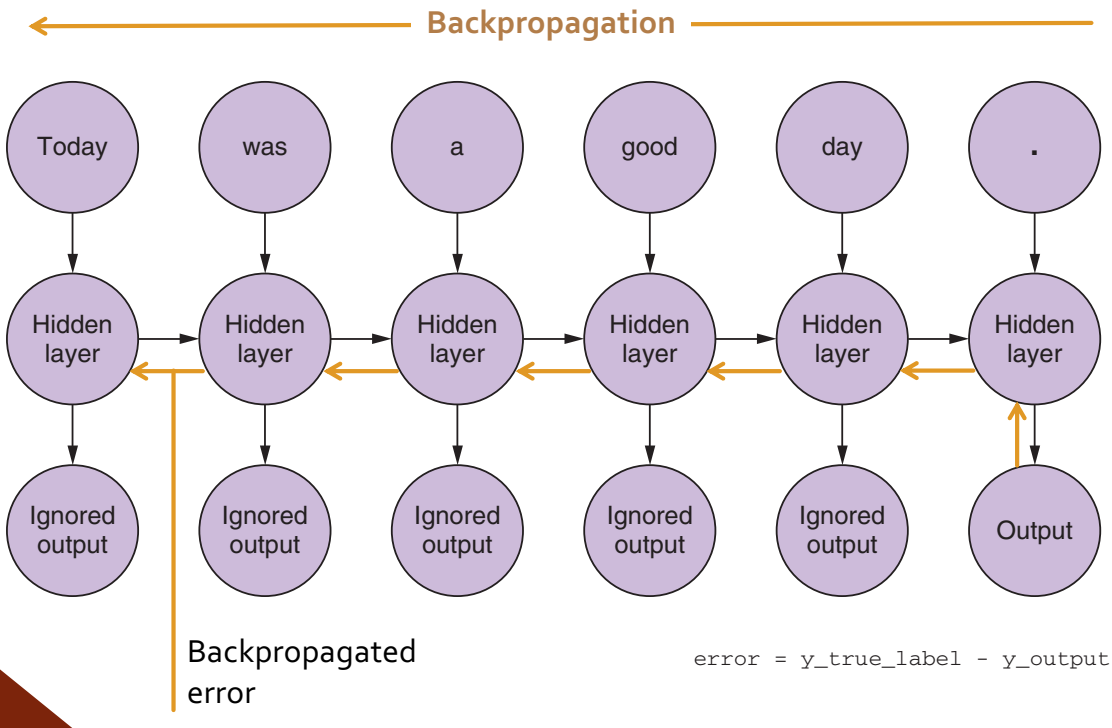
Text: The clown car sped into the arena

RNN Training




```
error = y_true_label - y_output
```

RNN Training



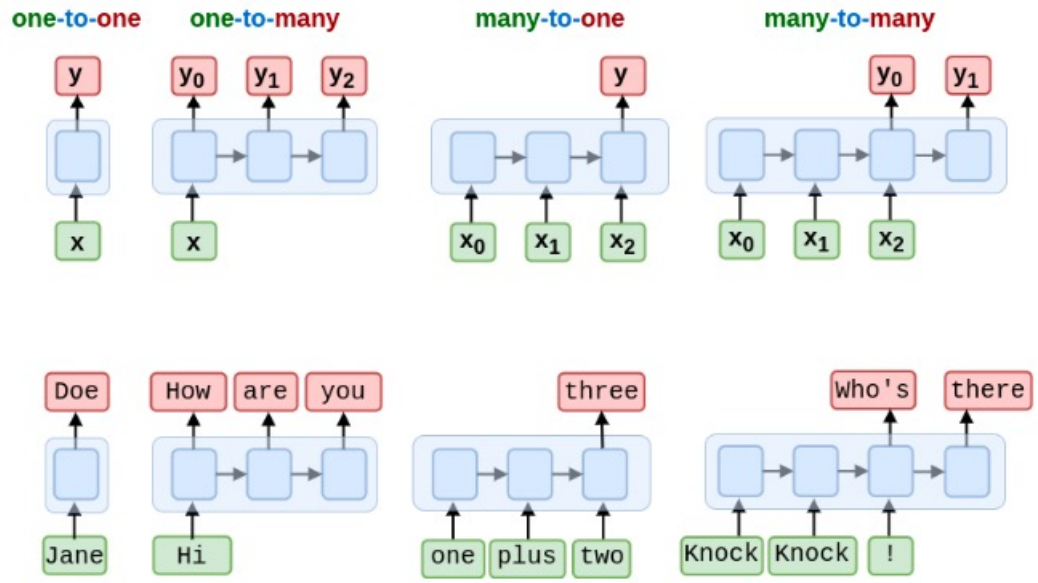

```
error = y_true_label - y_output
```

What are RNNs Good For?

RNNs can be used in several ways:

Type	Description	Applications
One to Many	One input tensor used to generate a sequence of output tensors	Generate chat messages, answer questions, describe images
Many to One	sequence of input tensors gathered up into a single output tensor	Classify or tag text according to its language, intent, or other characteristics
Many to Many	a sequence of input tensors used to generate a sequence of output tensors	Translate, tag, or anonymize the tokens within a sequence of tokens, answer questions, participate in a conversation

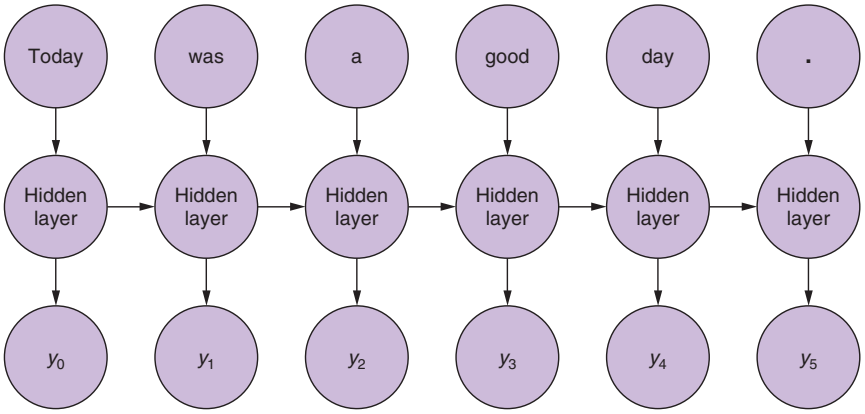
What are RNNs Good For?



RNNs for Text Generation

When used for **text generation**...

- The output of **each time step** is as important as the final output
- **Error** is captured and backpropagated **at each step** to adjust all network weights




```
error = sum([y_true_label[i] - y[i] for i in range(6)])
```

RNN Variants

Bidirectional RNN

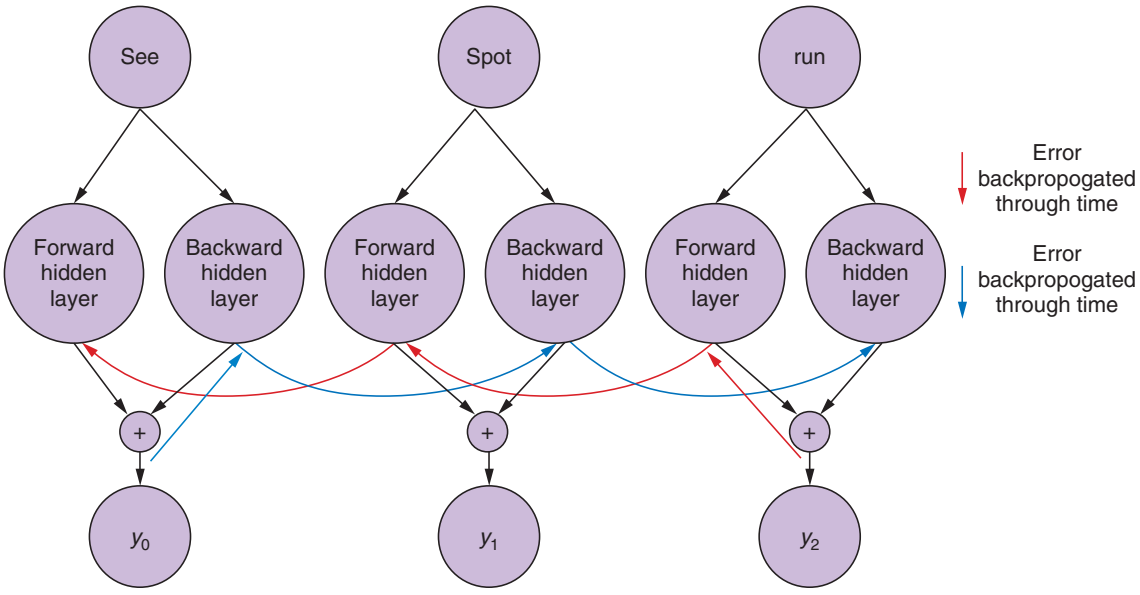
A Bidirectional RNN has **two recurrent hidden layers**

- One processes the input sequence **forward**
- The other processes the input sequence **backward**
- The output of those two are **concatenated** at each time step

By processing a sequence both ways, a bidirectional RNN can **catch patterns that may be overlooked** by a unidirectional RNN

- **Example:** *they wanted to pet the dog whose fur was brown*

Bi-directional RNN



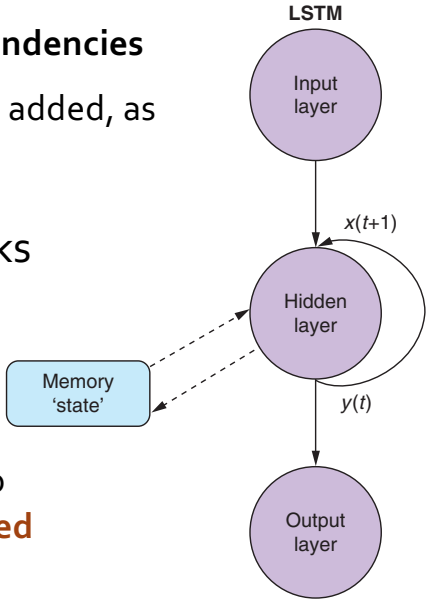
LSTM

RNNs should **theoretically retain** information from inputs seen many timesteps earlier but…

- They **struggle to learn long-term dependencies**
- **Vanishing Gradient:** as more layers are added, as the network becomes difficult to train

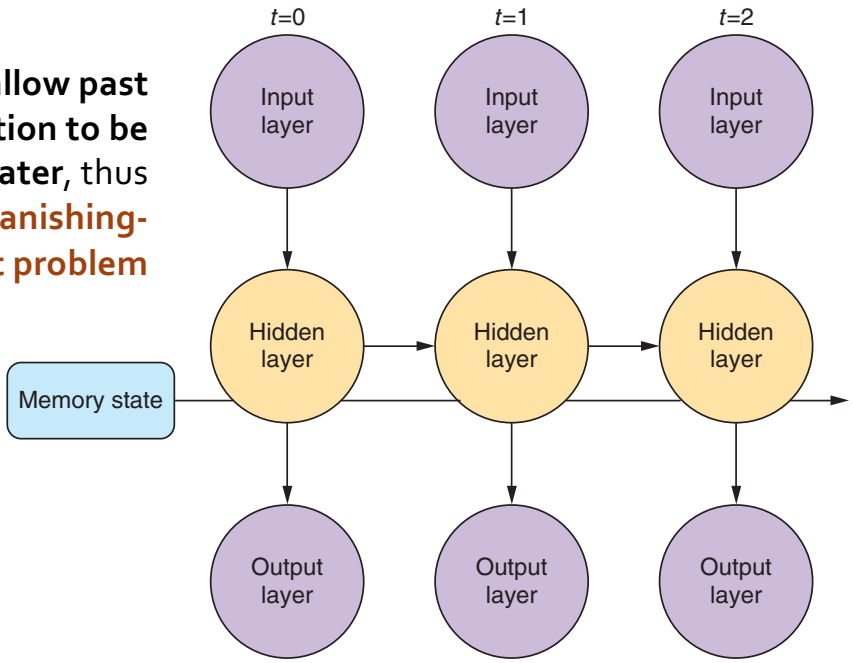
Long Short-Term Memory networks are designed to solve this problem:

- Introduces a **state** updated with each training example
- The **rules** to decide what information to remember and what to forget are **trained**



Unrolling the LSTM

LSTM **allow past information to be reinjected later**, thus fighting the **vanishing-gradient problem**



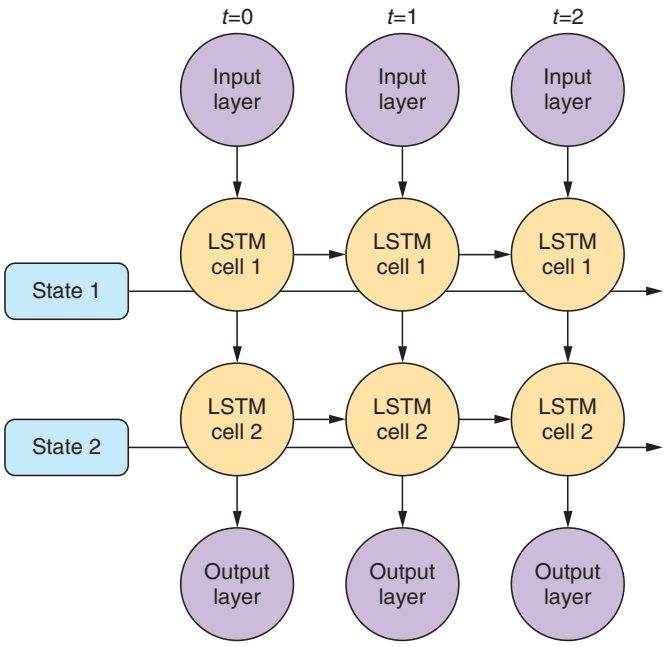
GRU

Gated Recurrent Unit (GRU) is an RNN architecture designed to solve the **vanishing gradient** problem

Main Features

- Like LSTM, but with a **simpler architecture**
- GRU **lacks a separate memory state**, relying solely on the hidden state to store and transfer information across timesteps
- **Fewer parameters than LSTM**, making it faster to train and more computationally efficient
- The **performance is comparable to LSTM**, particularly in tasks with simpler temporal dependencies

Stacked LSTM



Layering enhances the model's ability to **capture complex relationships**

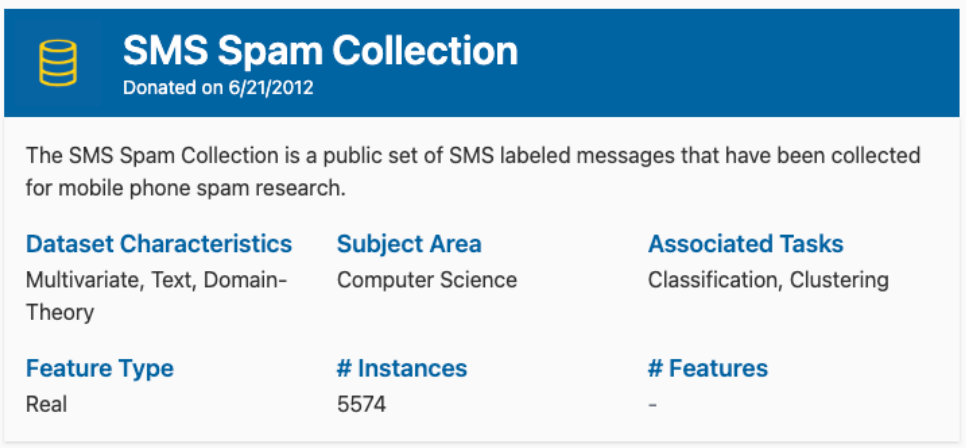
Note: The output at each timestep serves as the input for the corresponding timestep in the next layer

Building a Spam Detector

The Dataset

Download the dataset from:

https://archive.ics.uci.edu/dataset/228/sms+spam+collection



Read the Dataset

```
import pandas as pd

df = pd.read_csv("datasets/sms_spam.tsv", delimiter='\t', \
                 header = None, names = ['label', 'text'])
df
```

	label	text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5567	spam	This is the 2nd time we have tried 2 contact u...
5568	ham	Will ü b going to esplanade fr home?
5569	ham	Pity, * was in mood for that. So...any other s...
5570	ham	The guy did some bitching but I acted like i'd...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Tokenize and Generate WEs

```
import spacy, numpy as np
nlp = spacy.load('en_core_web_md') # loads the medium model with 300-dimensional WEs

# Tokenize the text and save the WEs
corpus = []
for sample in df['text']:
    doc = nlp(sample, disable=["tagger", "parser", "attribute_ruler", \
                               "lemmatizer", "ner"]) # only tok2vec
    corpus.append([token.vector for token in doc])

# Pad or truncate samples to a fixed length
maxlen = 50
zero_vec = [0] * len(corpus[0][0])
for i in range(len(corpus)):
    if len(corpus[i]) < maxlen:
        corpus[i] += [zero_vec] * (maxlen - len(corpus[i])) # pad
    else:
        corpus[i] = corpus[i][:maxlen] # truncate

corpus = np.array(corpus)
corpus.shape
```

```
(5572, 50, 300)
```

Split the dataset

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# Encode the labels
encoder = LabelEncoder()
labels = encoder.fit_transform(df['label'])

# Split the data
X_train, X_test, y_train, y_test = train_test_split(corpus, labels, test_size = 0.2)

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((4457, 50, 300), (1115, 50, 300), (4457,), (1115,))
```

Train an RRN model

```
# pip install tensorflow keras

import keras

model = keras.models.Sequential()
model.add(keras.layers.Input(shape = (X_train.shape[1], X_train.shape[2])))
model.add(keras.layers.SimpleRNN(64))
model.add(keras.layers.Dropout(0.3))
model.add(keras.layers.Dense(1, activation='sigmoid'))
model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
model.summary()

history = model.fit(X_train, y_train, batch_size = 512, epochs = 20,
                    validation_data = (X_test, y_test))
```

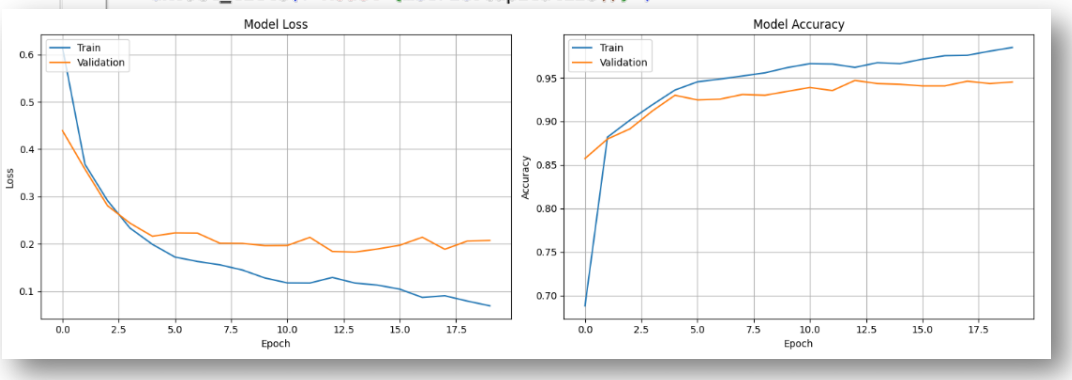
Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 64)	23,360
dropout (Dropout)	(None, 64)	0
dense (Dense)	(None, 1)	65

Plot the Training History

```
from  matplotlib import pyplot as plt

def plot(history, metrics):
    fig, axes = plt.subplots(1, len(metrics), figsize = (15, 5))
    for i, metric in enumerate(metrics):
        ax = axes[i]
        ax.plot(history.history[metric], label='Train')
        ax.plot(history.history['val_' + metric], label='Validation')
        ax.set_title(f'Model {metric.capitalize()}')
        ax.set_ylabel(metric.capitalize())
        ax.set_xlabel('Epoch')
        ax.legend(loc='upper left')
        ax.grid()
    plt.tight_layout()
    plt.show()

plot(history, ['loss', 'accuracy'])
```

Plot the Training History

```
from  matplotlib import pyplot as plt

def plot(history, metrics):
    fig, axes = plt.subplots(1, len(metrics), figsize = (15, 5))
    for i, metric in enumerate(metrics):
        ax = axes[i]
        ax.plot(history.history[metric], label='Train')
        ax.plot(history.history['val_' + metric], label='Validation')
        ax.set_title(f'Model {metric.capitalize()}')
```



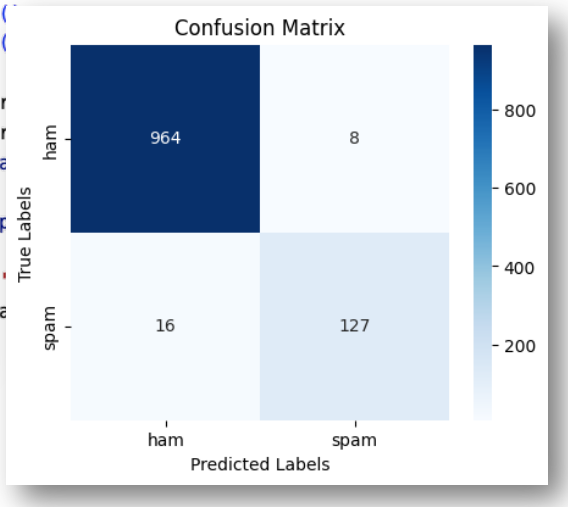
Report and Confusion Matrix

```python
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
import seaborn as sns

def print_report(model, X_test, y_test, encoder):
    y_pred = model.predict(X_test).ravel()
    y_pred_class = (y_pred > 0.5).astype(int) # convert probabilities to classes

    y_pred_lab = encoder.inverse_transform(y_pred_class)
    y_test_lab = encoder.inverse_transform(y_test)
    print(classification_report(y_test_lab, y_pred_lab, zero_division = 0))

    cm = confusion_matrix(y_test_lab, y_pred_lab)
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot = True, fmt = 'd', cmap = 'Blues',
                xticklabels = encoder.classes_, yticklabels = encoder.classes_)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()
```

Report and Confusion Matrix

```python
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
import seaborn as sns

def print_report(model, X_test, y_test, encoder):
    y_pred = model.predict(X_test).ravel(
    y_pred_class = (y_pred > 0.5).astype(

    y_pred_lab = encoder.inverse_transfor
    y_test_lab = encoder.inverse_transfor
    print(classification_report(y_test_lab, y_pred_lab, zero_division = 0))

    cm = confusion_matrix(y_test_lab, y_pred_lab)
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot = True, fmt = 'd', cmap = 'Blues',
                xticklabels = encoder.classes_, yticklabels = encoder.classes_)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()
```

	precision	recall	f1-score	support
ham	0.96	0.97	0.97	972
spam	0.81	0.75	0.78	143
accuracy			0.95	1115
macro avg	0.89	0.86	0.87	1115
weighted avg	0.94	0.95	0.94	1115



Using RNN Variants

- **Bi-directional RRN:**

```
model.add(keras.layers.Bidirectional(keras.layers.SimpleRNN(64)))
```

- **LSTM:**

```
model.add(keras.layers.LSTM(64))
```

- **Bi-directional LSTM:**

```
model.add(keras.layers.Bidirectional(keras.layers.LSTM(64)))
```

- **GRU:**

```
model.add(keras.layers.GRU(64))
```

- **Bi-directional GRU:**

```
model.add(keras.layers.Bidirectional(keras.layers.GRU(64)))
```

Using Ragged Tensors

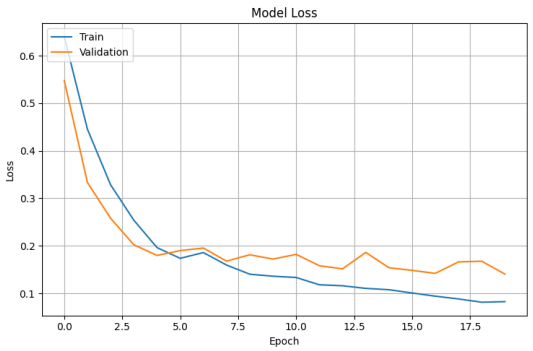
A **Ragged Tensor** is a tensor that allows **rows to have variable lengths**

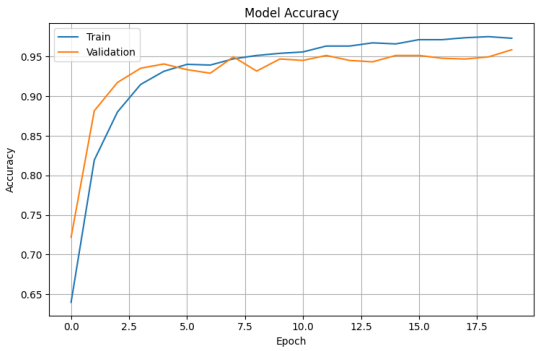
- Useful for handling data like text sequences, where **each input can have a different number of elements** (e.g., sentences with varying numbers of words)
- **Avoids the need for padding/truncating** sequences to a fixed length
- **Reduces overhead and improves computational efficiency** by directly handling variable-length data
- Available in **TensorFlow** since version 2.0
- In **PyTorch**, similar functionality is provided by **Packed Sequences**

Using Ragged Tensors

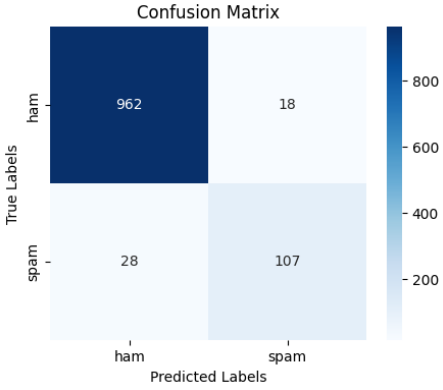
```
import tensorflow as tf

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(original_corpus, labels, test_size=0.2)

# Convert sequences into RaggedTensors to handle variable-length inputs
X_train_ragged = tf.ragged.constant(X_train)
X_test_ragged = tf.ragged.constant(X_test)

# Build the model
model = keras.models.Sequential()
model.add(keras.layers.Input(shape=(None, 300), ragged = True))
model.add(keras.layers.SimpleRNN(64))
model.add(keras.layers.Dropout(0.3))
model.add(keras.layers.Dense(1, activation='sigmoid'))
model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
model.summary()

# Train the model using RaggedTensors
history = model.fit(X_train_ragged, y_train, batch_size=512, epochs=20,
                    validation_data=(X_test_ragged, y_test))

plot(history, ['loss', 'accuracy'])
print_report(model, X_test_ragged, y_test, encoder)
```

Using Ragged Tensors






```
35/35 [==============================] - 0s 6ms/step
              precision    recall  f1-score   support

         ham       0.97      0.98      0.98       980
        spam       0.86      0.79      0.82       135

    accuracy                           0.96      1115
   macro avg       0.91      0.89      0.90      1115
weighted avg       0.96      0.96      0.96      1115
```



Intro to Text Generation

Generative Models

A class of NLP models designed to **generate new text**

- … that is **coherent and syntactically correct**
- … based on **patterns and structures learned** from text corpora

Generative vs Discriminative

- **Discriminative models** are mainly used to **classify or predict categories of data**
- **Generative models** can produce **new and original data**

RNN can be used to generate text

- **Transformers** are better (we will discuss them later)

Applications

- **Machine Translation**
  Automatically translating text from one language to another
- **Question Answering**
  Generating answers to questions based on a given context
- **Automatic Summarization**
  Creating concise summaries of longer texts
- **Text Completion**
  Predicting and generating the continuation of a given text
- **Dialogue Systems**
  Creating responses in conversational agents and chatbots
- **Creative Writing**
  Assisting in generating poetry, stories, and other creative texts

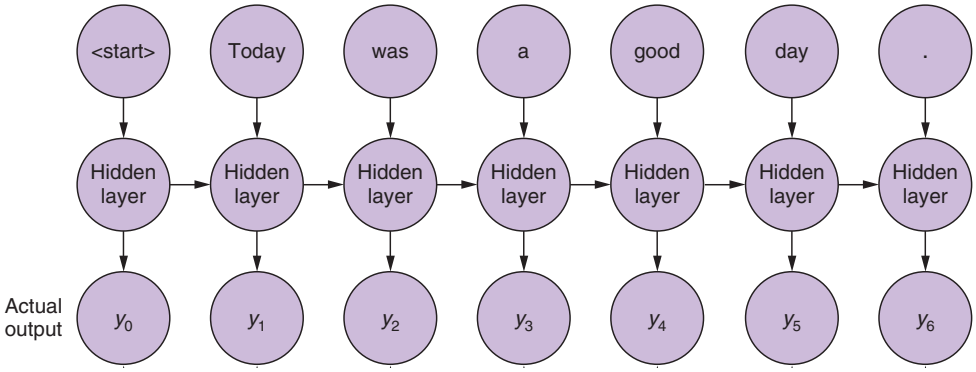
Language Model

A **mathematical model** that determine the **probability of the next token** given the previous ones

- It captures the **statistical structure** of the language **(latent space)**
- Once created, **can be sampled** to generate new sequences
  - An **initial string of text (conditioning data)** is provided
  - The model **generates a new token**
  - The generated token is **added to the input data**
  - the process is **repeated** several times
- This way, sequences of arbitrary length can be generated

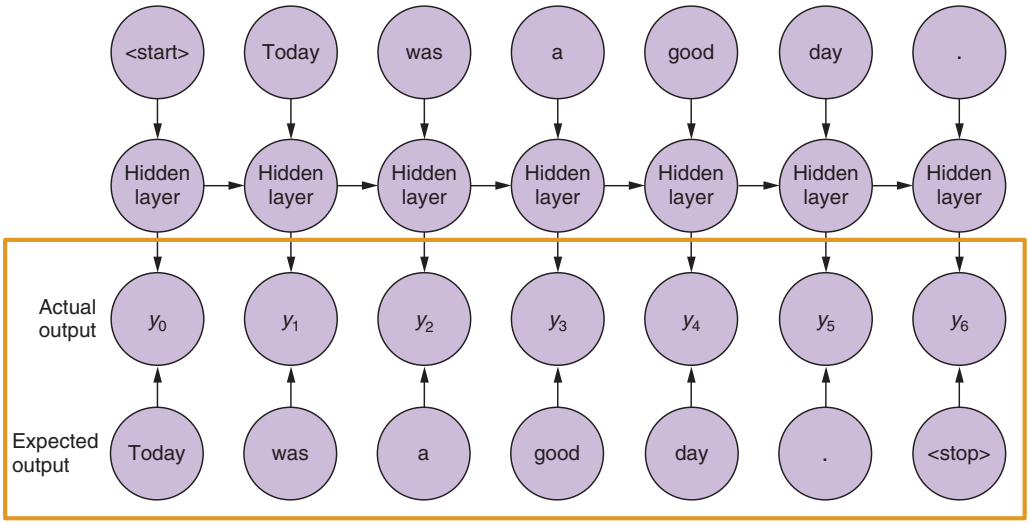
LM Training

At each step, the RNN **receives a token** extracted from a sentence in the corpus and **produces an output**



LM Training

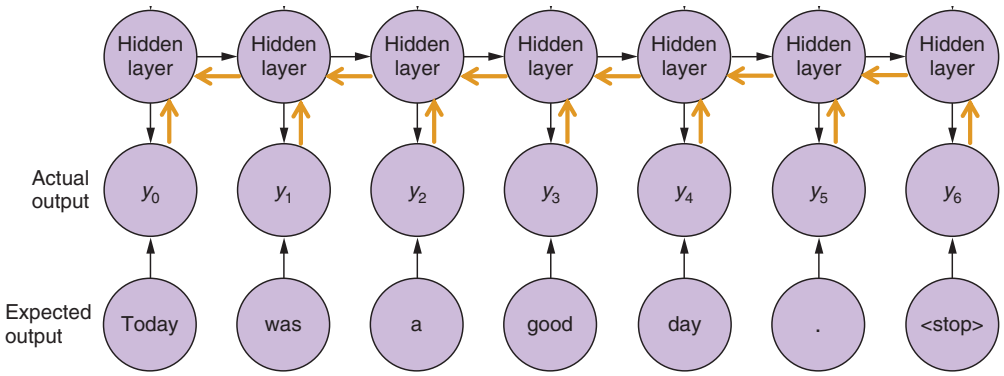
The output is **compared** with the **expected token** (the next one in the sentence)



LM Training

The comparison generates an **error**, which is used to **update the weights of the network** via backpropagation

- **Unlike traditional RNNs**, where backpropagation occurs only at the end of the sequence, **errors are propagated at each step**



Sampling

During utilization:

- **Discriminative models** always select the **most probable output** based on the given input
- **Generative models sample from the possible alternatives**:
  - **Example**: if a word has a probability of 0.3 of being the next word in a sentence, it will be chosen approximately 30% of the time

Temperature: a parameter (T) used to regulate the randomness of sampling

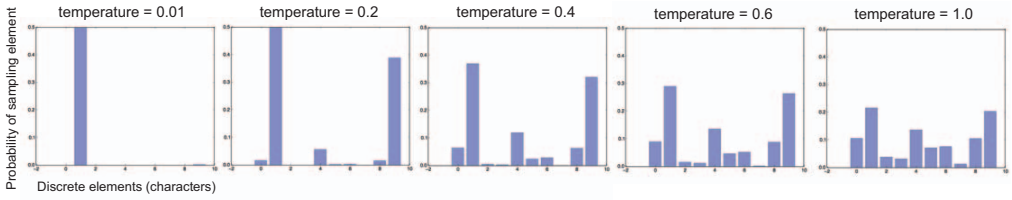
- A **low temperature (T<1)** makes the model more **deterministic**
- A **high temperature (T>1)** makes the model more **creative**

Temperature

$$q_i = exp\left(\frac{log(p_i)}{T}\right); q_i' = \frac{q_i}{\sum_j q_j}.$$

Where…

- p is the original probability distribution
- p_i is the probability of token i
- T>o is the chosen temperature
- q′ is the new distribution affected by the temperature



Building a Poetry Generator

Leopardi Poetry Generator

- Download the **corpus** from https://elearning.unisa.it/

```
# Load the corpus
with open('datasets/leopardi.txt', 'r') as f:
    text = f.read()

# Get the unique characters in the corpus
chars = sorted(list(set(text)))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))

print("Corpus length: {}; total chars: {}".format(len(text), len(chars)))
```

```
Corpus length: 134628; total chars: 77
```

Extract the Training Samples

We will use **characters rather than words** as **tokens**

```
maxlen = 40 # length of the extracted sequences

samples = [] # holds the extracted sequences
targets = [] # holds the next character for each sequence

# Extract sequences of maxlen characters
for i in range(0, len(text) - maxlen):
    samples.append(text[i: i + maxlen])
    targets.append(text[i + maxlen])

print('Number of samples: {}'.format(len(samples)))
```

```
Number of samples: 134588
```

```
import numpy as np

# Initialize the training data
X = np.zeros((len(samples), maxlen, len(chars)), dtype = bool)
y = np.zeros((len(samples), len(chars)), dtype = bool)

# One-hot encode samples and targets
for i, sample in enumerate(samples):
    for j, char in enumerate(sample):
        X[i, j, char_indices[char]] = 1
    y[i, char_indices[targets[i]]] = 1
```

Build and Train the Model

```
model = keras.models.Sequential()
model.add(keras.layers.Input(shape = (X.shape[1], X.shape[2])))
model.add(keras.layers.LSTM(128))
model.add(keras.layers.Dense(len(chars), activation = 'softmax'))
model.compile(loss = 'categorical_crossentropy', optimizer = "adam", metrics = ['accuracy'])
model.summary()

history = model.fit(X, y, batch_size = 128, epochs = 100, shuffle = True)
model.save('models/leopardi.keras')
```

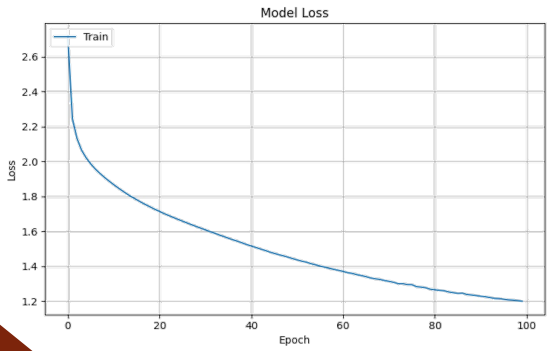
✓ 83m 31.0s

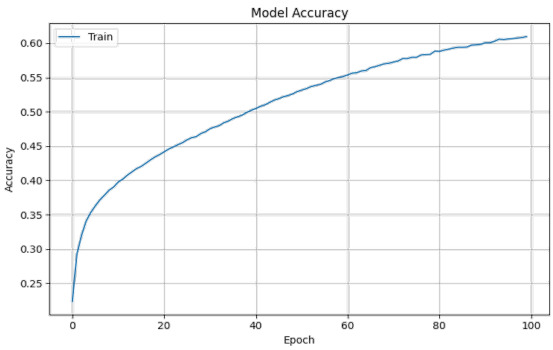
Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 128)	105,472
dense (Dense)	(None, 77)	9,933

Build and Train the Model

```
model = keras.models.Sequential()
model.add(keras.layers.Input(shape = (X.shape[1], X.shape[2])))
model.add(keras.layers.LSTM(128))
model.add(keras.layers.Dense(len(chars), activation = 'softmax'))
model.compile(loss = 'categorical_crossentropy', optimizer = "adam", metrics = ['accuracy'])
model.summary()

history = model.fit(X, y, batch_size = 128, epochs = 100, shuffle = True)
model.save('models/leopardi.keras')
```

✓ 83m 31.0s





Generate a new Poetry

Define **Helper Functions** first

```
import random, sys

# Sample the next character
def sample(preds, temperature):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

# Generate text from a seed
def generate_text(seed, length = 400, temperature = 0.5):
    sys.stdout.write(seed)
    seed = seed[-maxlen:].rjust(maxlen) # adjust the seed length
    for i in range(length):
        x = np.zeros((1, maxlen, len(chars)))
        for t, char in enumerate(seed):
            x[0, t, char_indices[char]] = 1
        preds = model.predict(x, verbose = 0)[0]
        next_char = indices_char[sample(preds, temperature)]
        seed = seed[1:] + next_char
        sys.stdout.write(next_char)
        sys.stdout.flush()
    print("\n")
```

Generate a new Poetry

```
# Generate text from a random two lines Leopardi's seed
lines = text.splitlines()
start_index = random.randint(0, len(lines) - 2)
generate_text("\n".join(lines[start_index:start_index + 2]))

# Generate text from an Ungaretti's seed
generate_text("""Ognuno sta solo sul cuor della terra
trafitto da un raggio di sole:
ed è subito sera.""")
```

```
I fusi delle Parche. Ogni giornale,
Gener vario di lingue e di colonne,
E così senza voce ad altro innoca
Dell'acquianza e l'alta, e la colona.
E tu con soliarti e dilettoso in terra
Di lor voler nelle sua vita, e il corro.
E di gli occhi son o le seco possa,
Con quando a te per lo specia e il vento
Dell'umana cui di luna, e il nuco
Di quest'avira speranza e l'amori,
Che di te per la morte a lei pativa. Un giorno
Dell'armi e delle amor preverea perro
Vai per mi parer la vita io ragio
Di tuo de' corsaggiar, che solo, e chiara
Che di lor vita e vivi in tutto scordo.
```

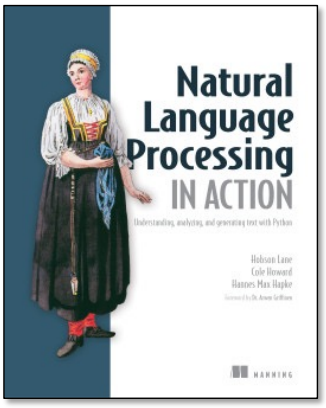
```
Ognuno sta solo sul cuor della terra
trafitto da un raggio di sole:
ed è subito sera.
Era di te stano al volto si cola
Dell'acceli stetto i carsi, o cara si soglie
Seduta il petto dolo
L'altro di rovanta.
Altri in terra di schielata, e la villa
Stal che tu con l'etra colonterati,
Se che il cor tempo, e la terra in tutto,
E di te che te ricolonda
La stravente vendesti in sul fatto estremo
Di colorati suoi l'ora già seggerdo
Del cielo accolor dell'animora, ancora
Quella vita morte, e di corre i migio.
```

References

Natural Language Processing IN ACTION

Understanding, analyzing, and generating text with Python

Chapters 8 and 9



Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica

Lesson 6

Neural Networks for NLP

Nicola Capuano and Antonio Greco

DIEM – University of Salerno