Lesson 01: Introduction to Natural Language Processing

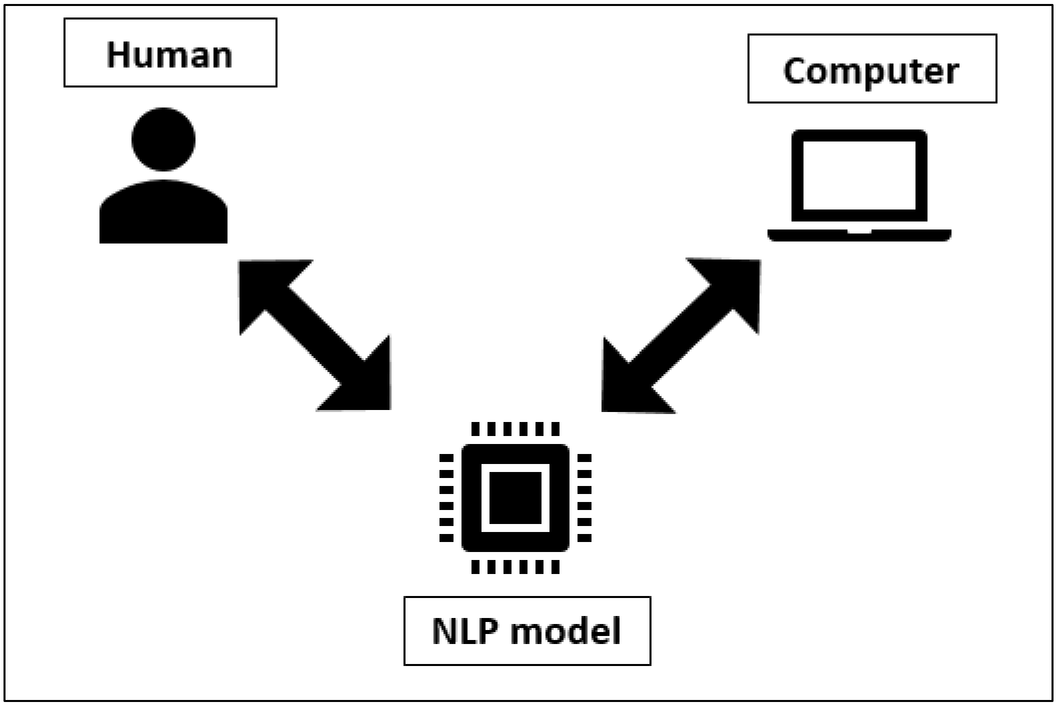

Figure 1.1: Natural language processing

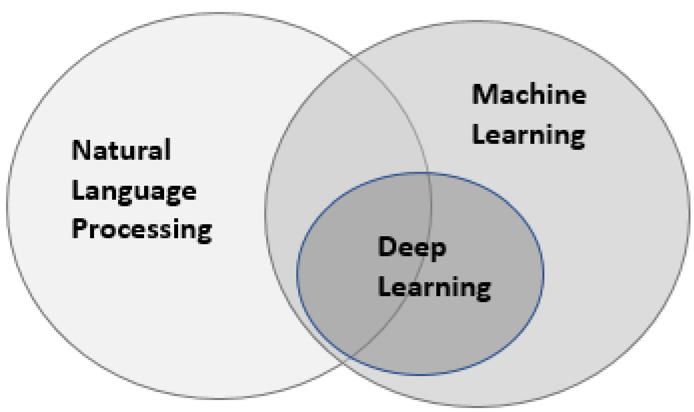

Figure 1.2: Venn diagram for natural language processing

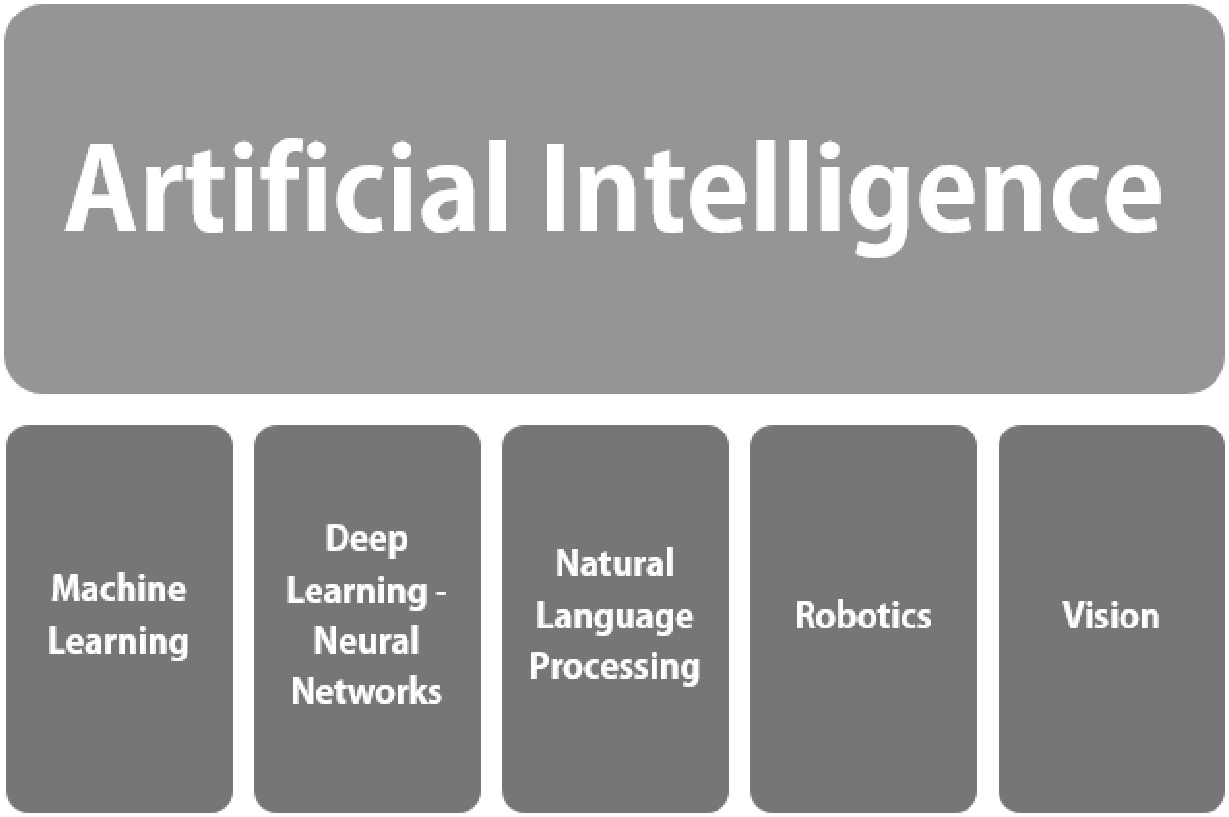

Fig 1.3: Artificial intelligence and some of its subfields

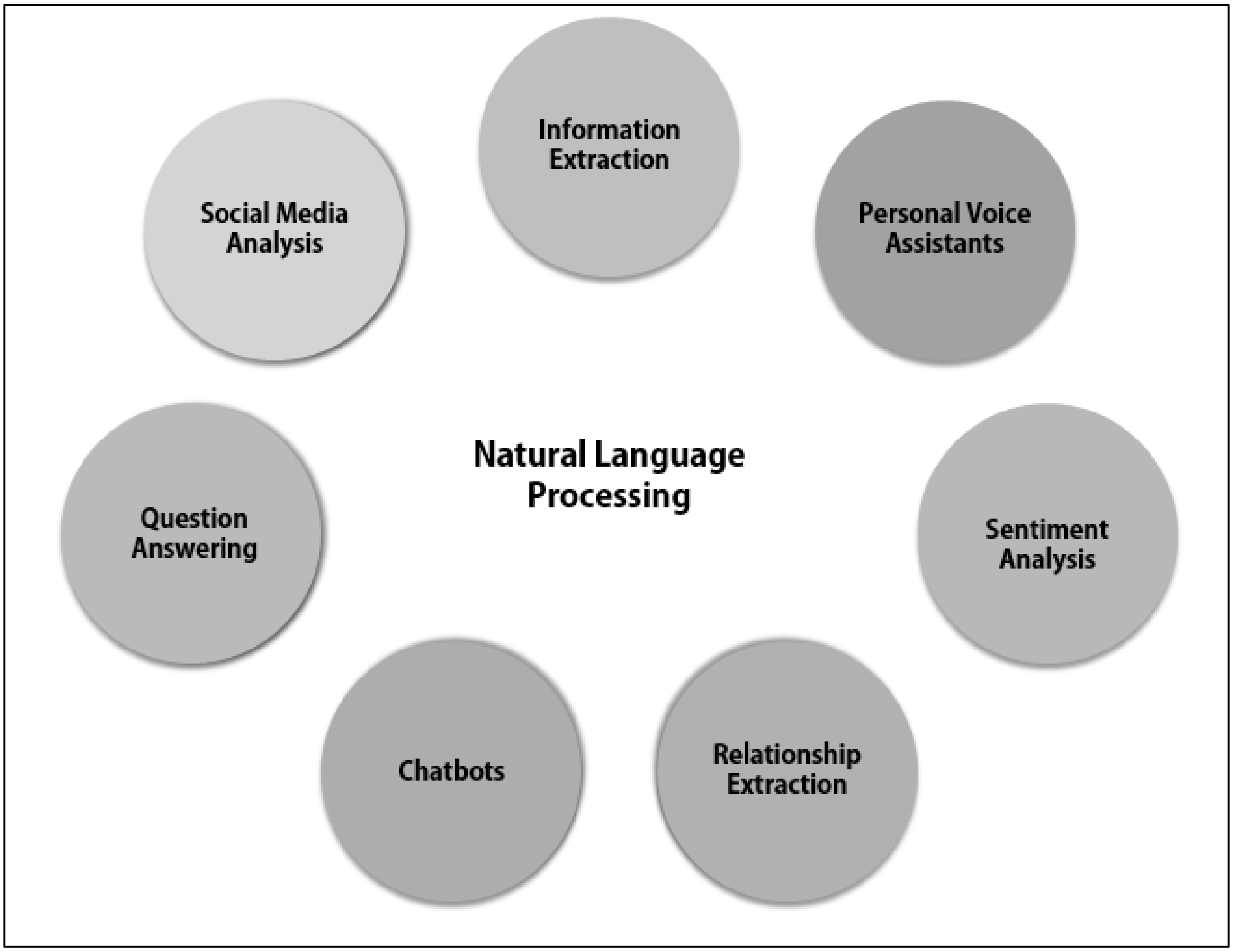

Figure 1.4: Application areas of natural language processing

```
the cities i like most in india are mumbai, bangalore, dharamsala and allahabad.
```

Figure 1.5: Output for lowercasing with mixed casing in a sentence

```
['india', 'india', 'india', 'india']
```

Figure 1.6: Output for lowercasing with mixed casing of words

<table>
<tr><th>With Noise</th><th>Without Noise</th></tr>
<tr><td>..sleepy</td><td rowspan="5">sleepy</td></tr>
<tr><td>sleepy!!</td></tr>
<tr><td>#sleepy</td></tr>
<tr><td>>>>>>sleepy>>>></td></tr>
<tr><td><a>sleepy</td></tr>
</table>

Figure 1.7: Output for noise removal

```
['sleepy', 'sleepy', 'sleepy', 'sleepy', 'sleepy']
```

Figure 1.8: Output for noise removal

<table>
<tr><th>Raw form</th><th>Canonical form</th></tr>
<tr><td>Spaghetti</td><td rowspan="5">Spaghetti</td></tr>
<tr><td>Spagetti</td></tr>
<tr><td>Spageti</td></tr>
<tr><td>Spaghetty</td></tr>
<tr><td>Spagetty</td></tr>
</table>

Figure 1.9: Canonical form for incorrect spellings

Raw form	Canonical form
brb	be right back

Figure 1.10: Canonical form for abbreviations

<table>
<tr><th>Before stemming</th><th>After stemming</th></tr>
<tr><td>Annoying</td><td rowspan="3">Annoy</td></tr>
<tr><td>Annoyed</td></tr>
<tr><td>Annoys</td></tr>
</table>

Figure 1.11: Output for stemming

	raw word	stem
0	annoying	annoy
1	annoys	annoy
2	annoyed	annoy
3	annoy	annoy

Figure 1.12: Output of stemming

	raw word	lemmatized
0	troubling	trouble
1	troubled	trouble
2	troubles	trouble
3	trouble	trouble

Figure 1.13: Output of lemmatization

```
['hi', '!', 'my', 'name', 'is', 'john', '.']
```

Figure 1.14: Output for the tokenization of words

```
['hi!', 'my name is john.']
```

Figure 1.15: Output for tokenizing sentences

```
['weather', 'really', 'hot', 'want', 'go', 'swim']
```

Figure 1.16: Output after removing stopwords

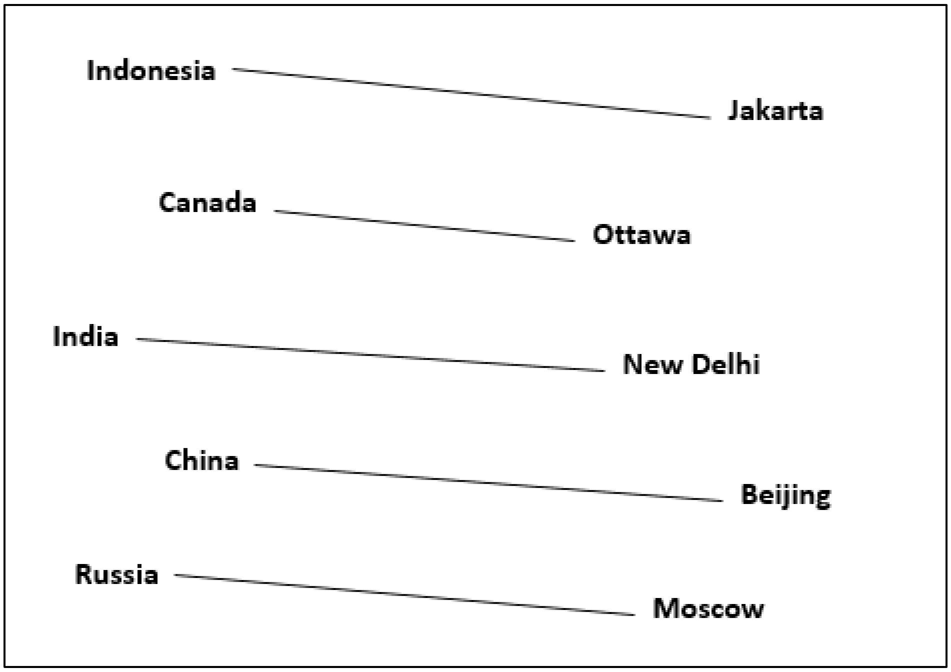

Figure 1.17: Example for word embeddings

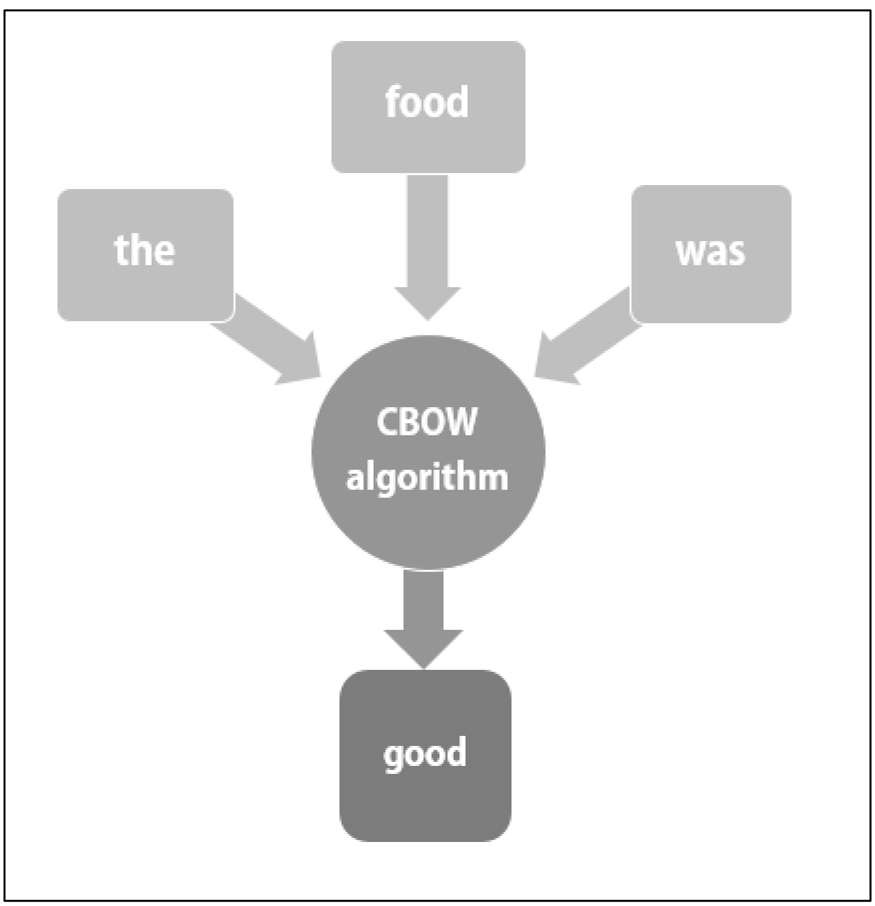

Fig 1.18: The CBOW algorithm

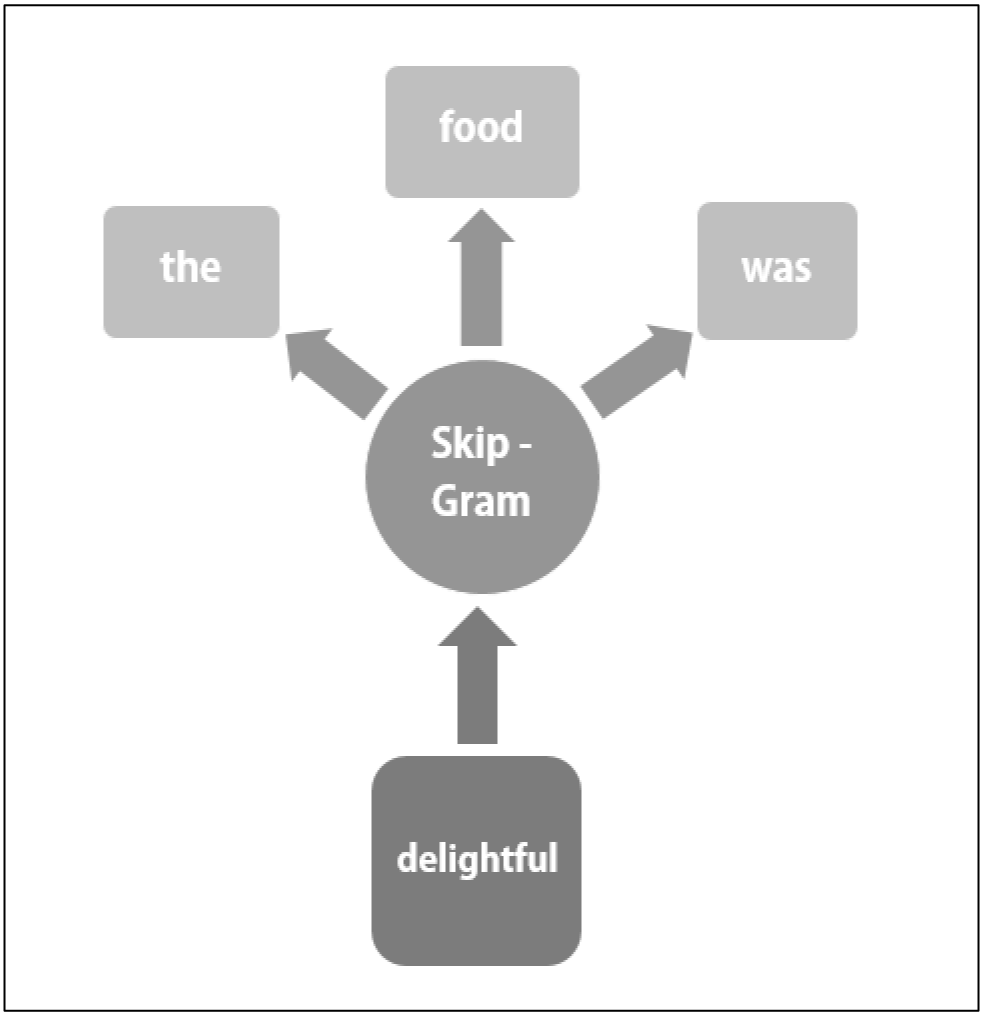

Fig 1.19: The skip-gram algorithm

```
this is the summary of the model:
Word2Vec(vocab=12, size=100, alpha=0.025)
```

Figure 1.20: Output for model summary

```
this is the vocabulary for our corpus:
['Ariana', 'Grande', 'is', 'a', 'singer', 'She', 'has', 'been', 'for', 'many', 'years', 'great']
```

Figure 1.21: Output for the vocabulary of the corpus

```
the vector for the word singer:
[ 3.9150659e-03  2.6659777e-03  1.0298982e-03 -2.7156321e-03
  1.9977870e-03  3.1204436e-03  1.2055682e-04  1.0450699e-03
 -6.4308796e-04  3.0822519e-03  2.1972554e-03  5.1480172e-05
 -3.7099270e-03  3.9439583e-03  6.8276987e-04  7.7137066e-04
  2.3698520e-03 -7.8547641e-04  6.0383842e-04  4.6370425e-03
 -1.6786088e-03  1.7417425e-03  2.4216413e-03  3.6545738e-03
 -1.9871239e-03  2.9489421e-03 -1.2810023e-03 -4.9174053e-04
 -3.9743204e-03 -2.7023794e-03 -3.0541950e-04 -1.5724347e-03
 -2.1029566e-03 -2.1624754e-03  2.1620055e-04 -1.4000515e-03
 -4.0824865e-03  4.6588355e-04  3.5028579e-03  4.8283348e-03
 -2.8737928e-03 -4.5569306e-03 -7.6568732e-04 -3.3311991e-03
  3.5790715e-03  4.2424244e-03  3.3478225e-03 -7.4140396e-04
  1.0030111e-03 -5.2394503e-04  5.8383477e-04 -4.8430995e-03
  2.6972082e-03 -4.8002079e-03 -2.3011414e-03  8.0388715e-04
  3.1952575e-05 -8.1621204e-04 -3.8127291e-03 -6.7428290e-04
 -1.7713077e-03 -3.0159748e-03  1.7178850e-03 -1.9258332e-03
 -2.4637436e-03  3.3779652e-03  2.7676420e-03  1.8853768e-03
 -2.4718521e-03 -1.9754141e-03  2.6104036e-03 -2.1335895e-03
  2.4405334e-03 -3.2013952e-04  3.9961869e-03  4.0419102e-03
  2.0586823e-03  4.9897884e-03  4.5599132e-03 -1.0976522e-03
  1.5563263e-03  3.9063310e-03 -2.9308300e-03 -4.8254002e-03
 -8.7642738e-06  3.9748671e-03  5.2895391e-04  6.3330121e-04
 -1.2614765e-03 -8.5018738e-04  3.7659388e-03  3.0237564e-03
  4.5014662e-03  4.3258793e-03 -4.2659100e-03  4.9081761e-03
 -3.9214552e-03 -2.4262110e-03 -8.1192164e-05 -4.1112076e-03]
```

Figure 1.22: Vector for the word 'singer'

```
[('has', 0.13253481686115265),
 ('been', 0.12117968499660492),
 ('for', 0.10510198771953583),
 ('singer', 0.08586522936820984),
 ('a', 0.08413773775100708),
 ('She', 0.08044794946908951)]
```

Figure 1.23: Word vectors similar to the word 'great'

```
[('for', 0.17918002605438232),
 ('been', 0.12124449759721756),
 ('great', 0.08586522936820984),
 ('is', 0.0768381804227829),
 ('a', 0.03302524611353874),
 ('Ariana', 0.02957470342516899)]
```

Figure 1.24: Word vector similar to word 'singer'

```
[('woman', 0.7866706012658177),
 ('young', 0.7787864197368234),
 ('spider', 0.7728204994207245),
 ('girl', 0.7642560909647501)]
```

Figure 1.25: Output of word embeddings for 'man'

```
[('elizabeth', 0.9290495990532598),
 ('victoria', 0.8600464526851297),
 ('mary', 0.8089403382412337),
 ('anne', 0.7667713770457262),
 ('scotland', 0.6942531928211478),
 ('catherine', 0.6910265819525973),
 ('consort', 0.6906798004149294),
 ('tudor', 0.6686379422061477),
 ('isabella', 0.6666968276614551)]
```

Figure 1.26: Output of word embeddings for 'queen'

```
[('woman', 0.6842043995857239),
 ('girl', 0.5943484306335449),
 ('creature', 0.5780946612358093),
 ('boy', 0.5204570293426514),
 ('person', 0.5135789513587952),
 ('stranger', 0.506704568862915),
 ('beast', 0.504448652267456),
 ('god', 0.5037523508071899),
 ('evil', 0.4990573525428772),
 ('thief', 0.4973783493041992)]
```

Figure 1.27: Output for similar word embeddings

```
[('mother', 0.7770676612854004),
 ('grandmother', 0.7024110555648804),
 ('wife', 0.6916966438293457)]
```

Figure 1.28: Output for top three words for 'x'

Lesson 02: Applications of Natural Language Processing

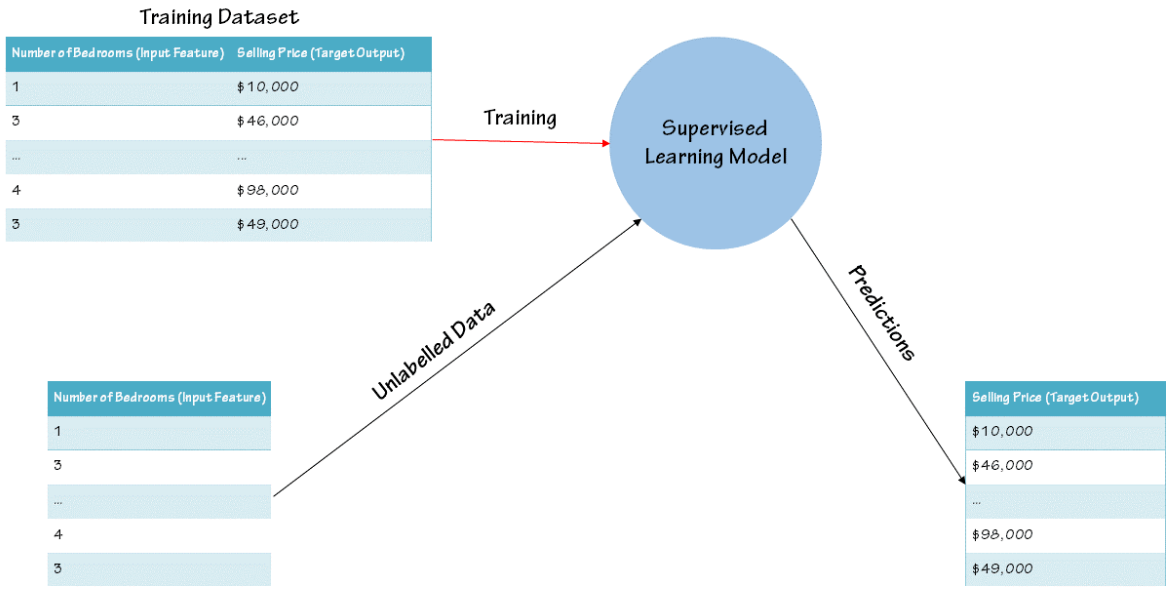

Fig 2.1: Supervised learning

Number	Tag	Description
1	CC	Coordinating conjunction
2	CD	Cardinal number
3	DT	Determiner
4	EX	Existential there
5	FW	Foreign word
6	IN	Preposition or subordinating conjunction
7	JJ	Adjective
8	JJR	Adjective, comparative
9	JJS	Adjective, superlative
10	LS	List item marker
11	MD	Modal
12	NN	Noun, singular or mass
13	NNS	Noun, plural
14	NNP	Proper noun, singular
15	NNPS	Proper noun, plural
16	PDT	Predeterminer
17	POS	Possessive ending
18	PRP	Personal pronoun
19	PRP$	Possessive pronoun
20	RB	Adverb
21	RBR	Adverb, comparative
22	RBS	Adverb, superlative
23	RP	Particle
24	SYM	Symbol
25	TO	*To*
26	UH	Interjection
27	VB	Verb, base form
28	VBD	Verb, past tense
29	VBG	Verb, gerund or present participle
30	VBN	Verb, past participle
31	VBP	Verb, non-3rd person singular present
32	VBZ	Verb, 3rd person singular present
33	WDT	Wh-determiner
34	WP	Wh-pronoun
35	WP$	Possessive wh-pronoun
36	WRB	Wh-adverb

Figure 2.2: POS tags with descriptions

```
[('i', 'NN'),
 ('enjoy', 'VBP'),
 ('playing', 'VBG'),
 ('the', 'DT'),
 ('piano', 'NN')]
```

Fig 2.3: Tagged output

```
NN: noun, common, singular or mass
    common-carrier cabbage knuckle-duster Casino afghan shed thermostat
    investment slide humour falloff slick wind hyena override subhumanity
    machinist ...
```

Fig 2.4: Noun details

```
[('and', 'CC'),
 ('so', 'RB'),
 ('i', 'JJ'),
 ('said', 'VBD'),
 ('im', 'NN'),
 ('going', 'VBG'),
 ('to', 'TO'),
 ('play', 'VB'),
 ('the', 'DT'),
 ('piano', 'NN'),
 ('for', 'IN'),
 ('the', 'DT'),
 ('play', 'NN'),
 ('tonight', 'NN')]
```

Fig 2.5: Tagged output

```
and CCONJ CC
so ADV RB
i PRON PRP
said VERB VBD
i PRON PRP
'm VERB VBP
going VERB VBG
to PART TO
play VERB VB
the DET DT
piano NOUN NN
for ADP IN
the DET DT
play NOUN NN
tonight NOUN NN
```

Figure 2.6: Output for POS tags

Figure 2.7: Parse tree.

Figure 2.8: Output for chunking.

```
the beautiful butterfly butterfly nsubj
the night sky sky pobj
```

Figure 2.9: Output for chunking with spaCy

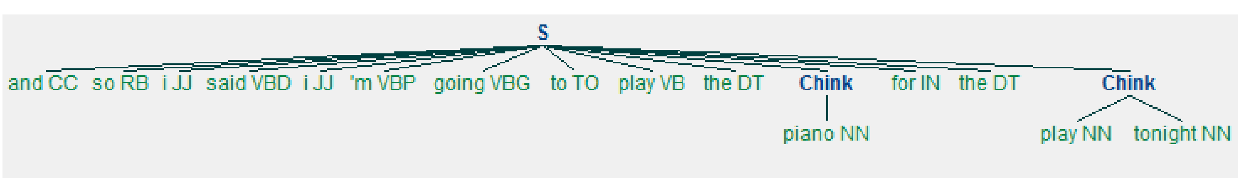

Figure 2.10: Output for chinking

```
Training completed
Accuracy: 0.8959505061867267
```

Figure 2.11: Expected accuracy score.

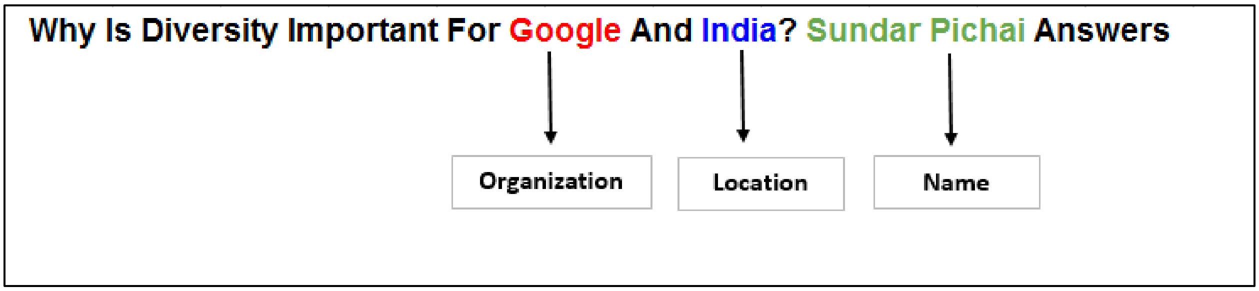

Figure 2.12: Example for named entity recognition

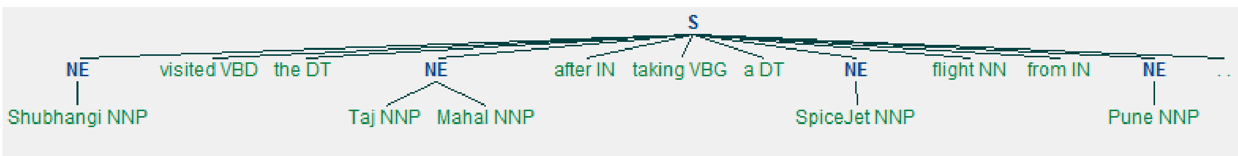

Figure 2.13: Output for named entity recognition with POS tags

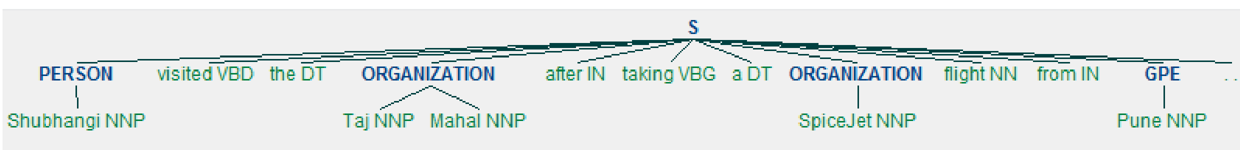

Figure 2.14: Output with named entities

PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Companies, agencies, institutions, etc.
GPE	Countries, cities, states.
LOC	Non-GPE locations, mountain ranges, bodies of water.
PRODUCT	Objects, vehicles, foods, etc. (Not services.)
EVENT	Named hurricanes, battles, wars, sports events, etc.
WORK_OF_ART	Titles of books, songs, etc.
LAW	Named documents made into laws.
LANGUAGE	Any named language.
DATE	Absolute or relative dates or periods.
TIME	Times smaller than a day.
PERCENT	Percentage, including "%".
MONEY	Monetary values, including unit.
QUANTITY	Measurements, as of weight or distance.
ORDINAL	"first", "second", etc.
CARDINAL	Numerals that do not fall under another type.

Figure 2.15: Categories of spaCy

```
SpiceJet ORG
Pune GPE
```

Figure 2.16: Output for named entity

```
Shubhangi Hora PERSON
the Taj Mahal WORK_OF_ART
SpiceJet ORG
Pune GPE
```

Figure 2.17: Output for named entity recognition with spaCy.

```
(S
  (PERSON Rudolph/NNP)
  (GPE Agnew/NNP)
  ,/,
  55/CD
  years/NNS
  old/JJ
  and/CC
  former/JJ
  chairman/NN
  of/IN
  (ORGANIZATION Consolidated/NNP Gold/NNP Fields/NNP)
  PLC/NNP
  ,/,
  was/VBD
  named/VBN
  *-1/-NONE-
  a/DT
  nonexecutive/JJ
  director/NN
  of/IN
  this/DT
  (GPE British/JJ)
  industrial/JJ
  conglomerate/NN
  ./.)
```

Figure 2.18: Expected output for NER on tagged corpus

Lesson 03: Introduction to Neural Networks

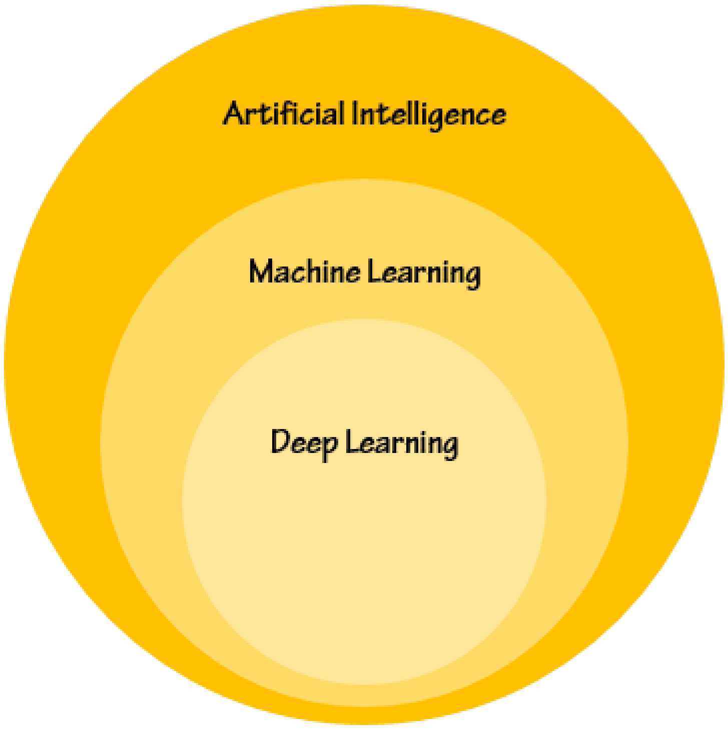

Fig 3.1: Deep Learning as a subfield of Machine Learning

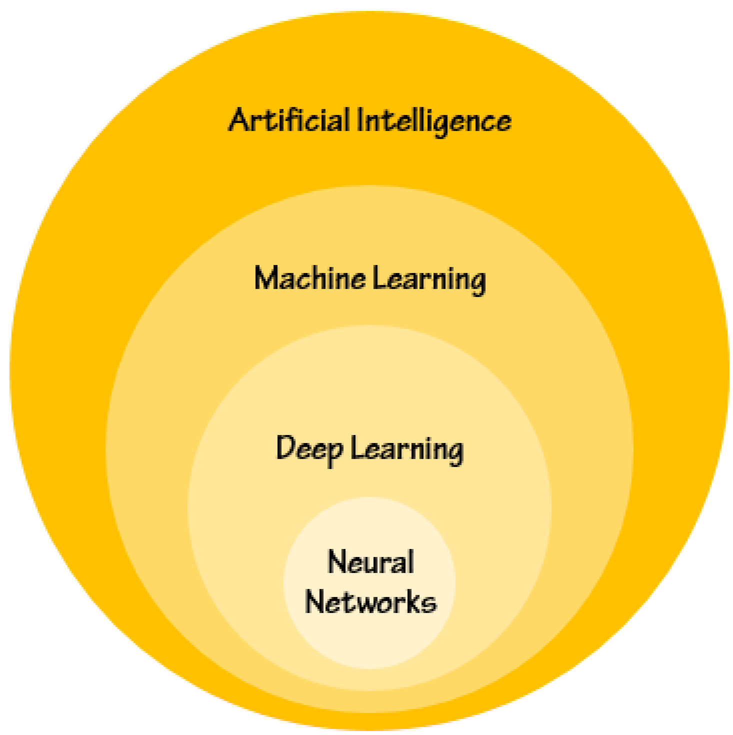

Fig 3.2: Neural Networks as a part of the Deep Learning Approach

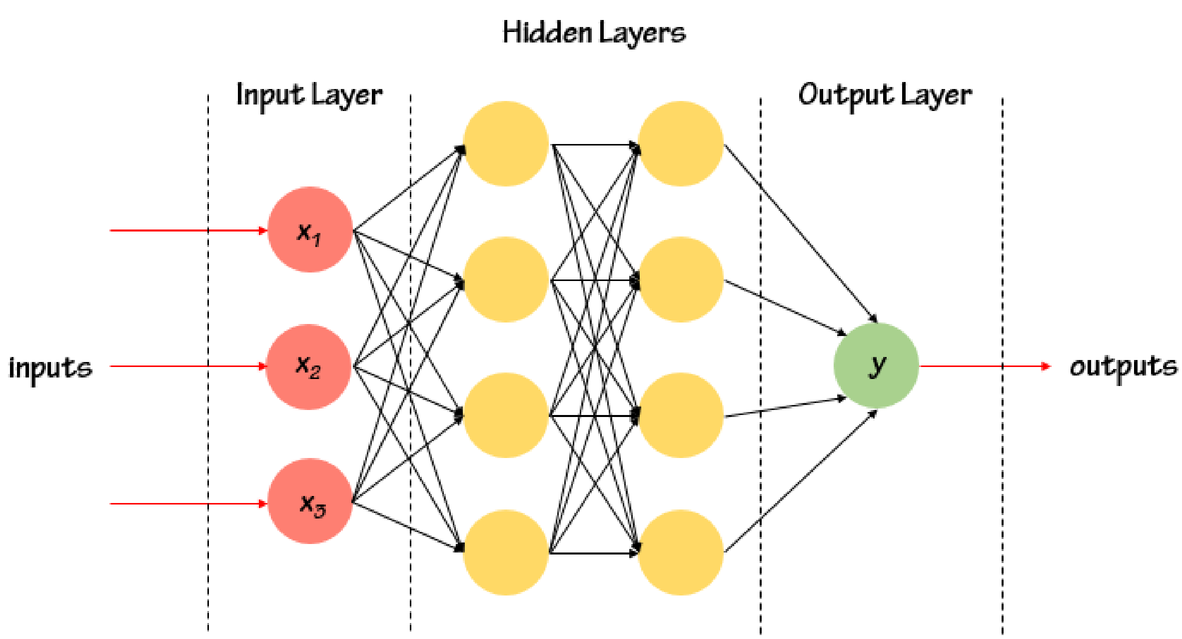

Fig 3.3: A Neural Network with 2 Hidden Layers

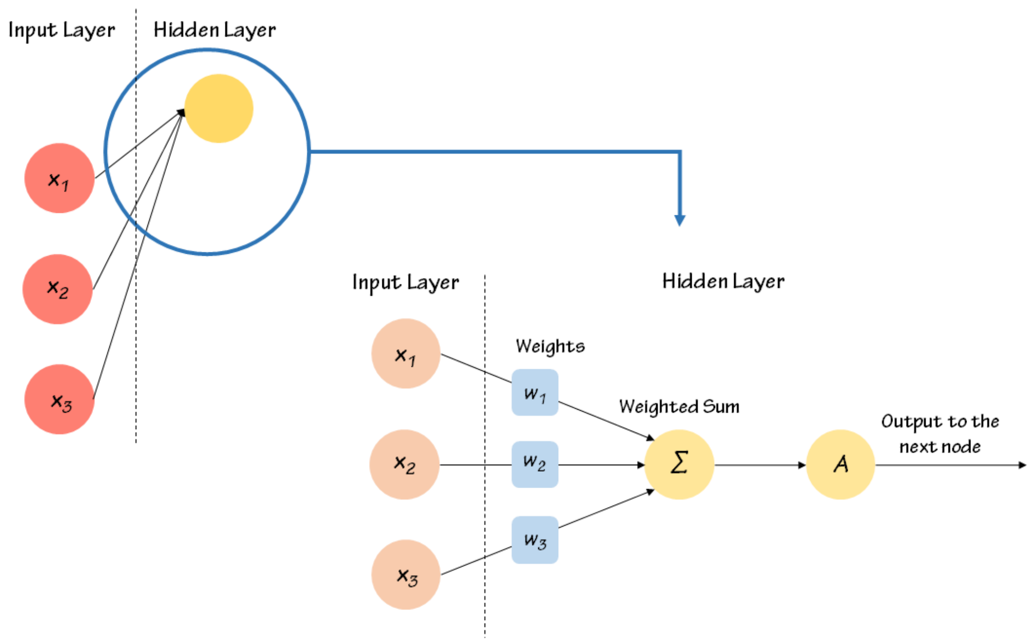

Fig 3.4: The Weighted Connections of a Neural Network

$$f(x) = \frac{1}{1 + e^{-x}}$$

Figure 3.5: Expression for sigmoid function

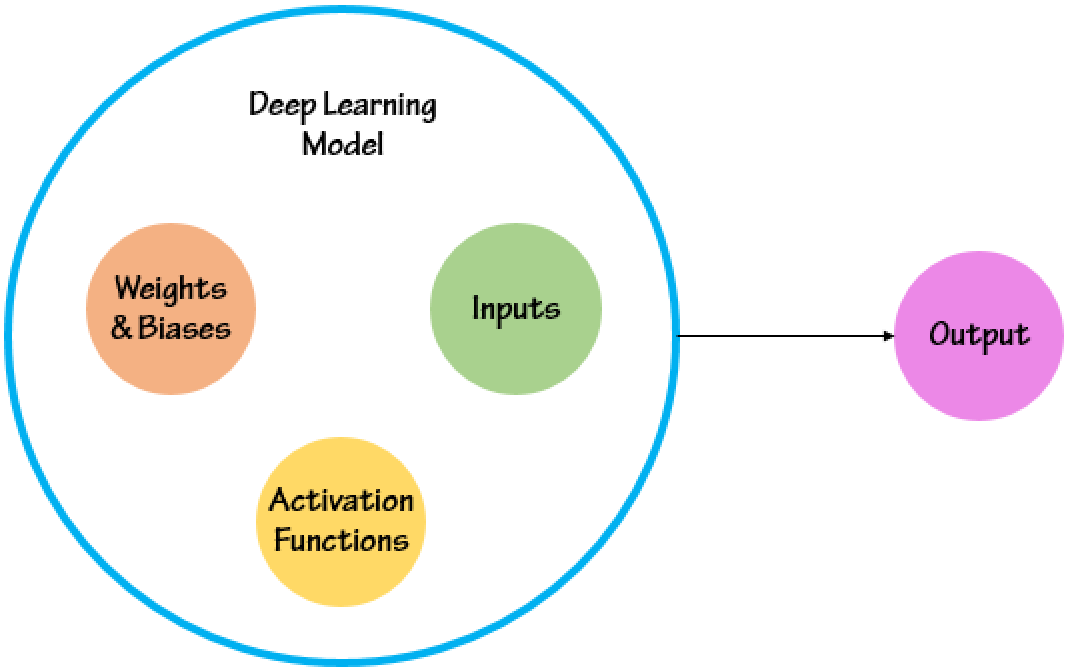

Figure 3.6: Aspects of a deep learning model that impact the output

$$y = c + mx$$

Figure 3.7: Expression for linear regression

Number of Bedrooms (Input Feature)	Selling Price (Target Output)
1	$10, 000
3	$46, 000
...	...
4	$98, 000
3	$49, 000

Fig 3.8: Sample Dataset for Linear Regression

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - f(x_i))^2$$

Figure 3.9: Expression for mean squared error function

$$Log\ Loss = -\frac{1}{N}\sum_{i=1}^{N} y_i\,(\log(p(y_i)) + (1 - y_i))(\log(1 - p(y_i)))$$

Figure 3.10: Expression for log loss function

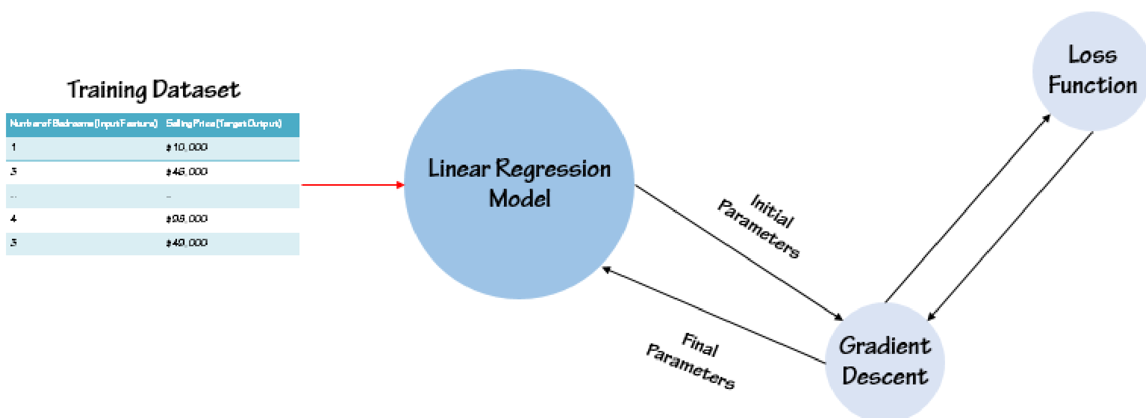

Fig 3.11: Updating Parameters

$$f(w,b) = \frac{1}{n}\sum_{i}^{n}(y_i - f(wx_i + b))^2$$

Figure 3.12: Expression for gradient of loss function

$$f'(w,b) = \begin{bmatrix} \frac{df}{dw} \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N}\sum -2x_i(y_i - (wx_i + b)) \\ \frac{1}{N}\sum -2x_i(y_i - (wx_i + b)) \end{bmatrix}$$

Figure 3.13: Expression of gradient with partial derivaive of loss function

$$w = w - \left(\frac{{df}/{dw}}{N} \right) * \alpha$$

Figure 3.14: Expression for learning rate multipled with gradient

$$b = b - \left(\frac{{df}/{db}}{N} \right) * \alpha$$

Figure 3.15: Expression for learning rate multipled with gradient at each step

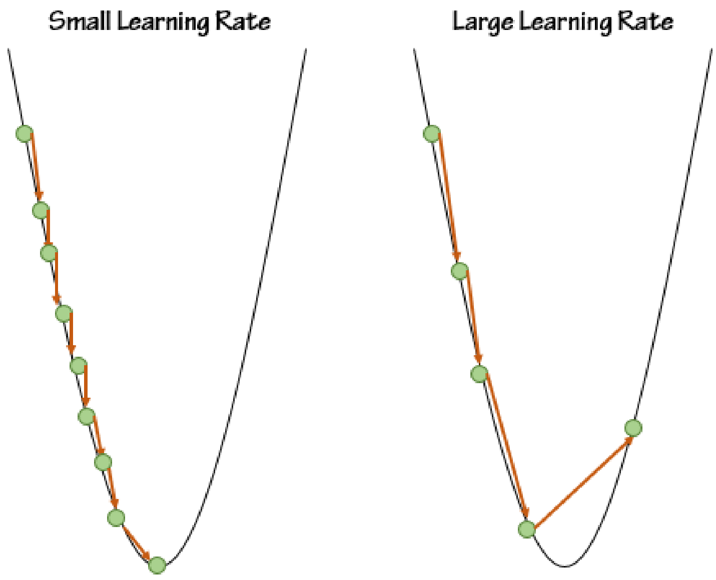

Fig 3.16: Learning Rate

$$f(x) = X(Y(Z(x)))$$

Figure 3.17: Expression for backpropagation function

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 500)               28500
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 501
=================================================================
Total params: 29,001
Trainable params: 29,001
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.18: Model summary

```
10/10 [==============================] - 0s 135us/step
Accuracy: 0.8999999761581421
```

Figure 3.19: Expected accuracy score

```
20/20 [==============================] - 0s 160us/step
Accuracy: 1.0
[1.192093321833454e-07, 1.0]
```

Figure 3.20: Accuracy score

Lesson 04: Foundations of Convolutional networks

Figure 4.1: Examples of spatial variance

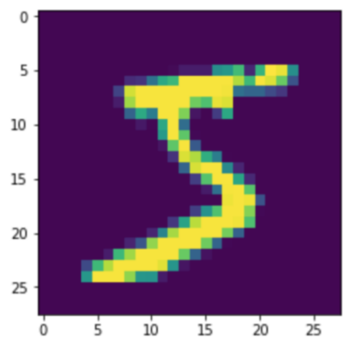

Figure 4.2: Visualization of an image

253
170
127
154

Figure 4.3: Numerical representation of an image

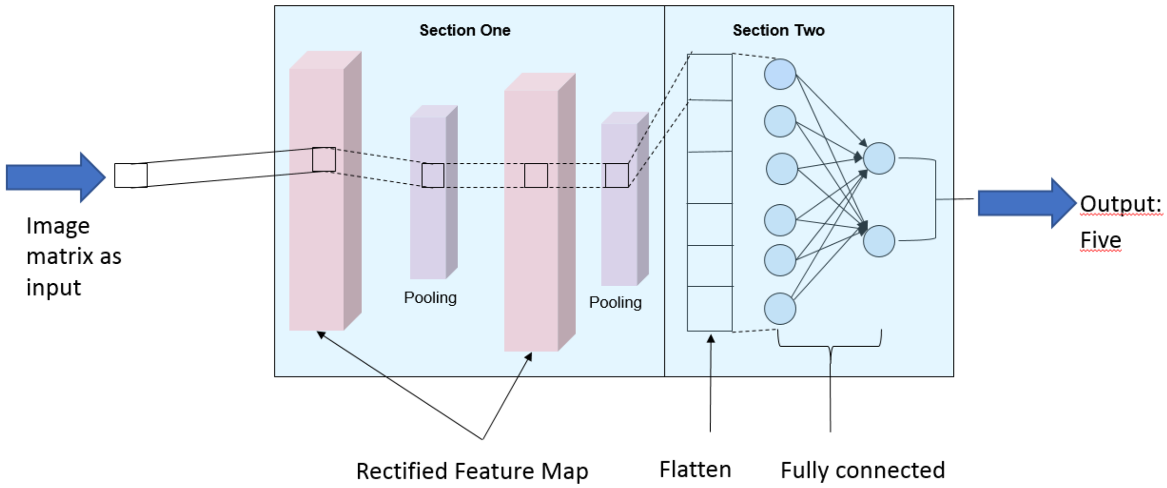

Figure 4.4: Application of convolution and ReLU operations

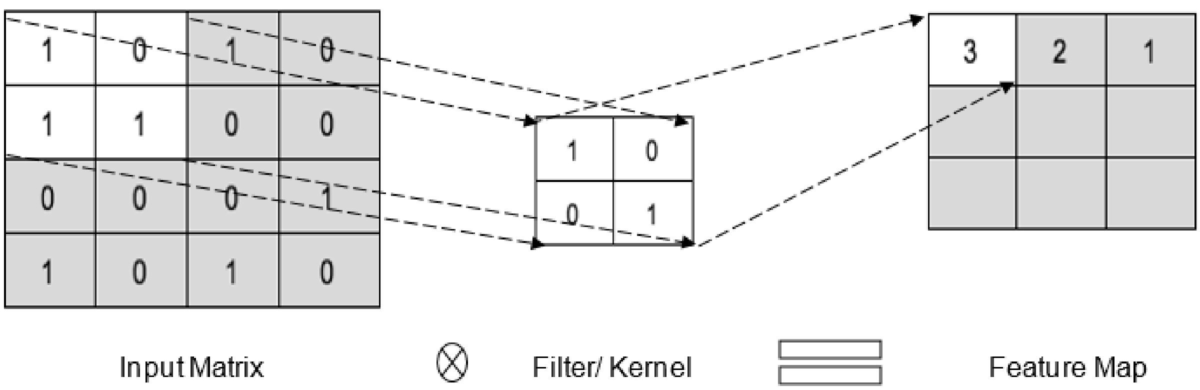

Figure 4.5: Filter application to images

Figure 4.6: Image after applying ReLU function

Figure 4.8: Max pool

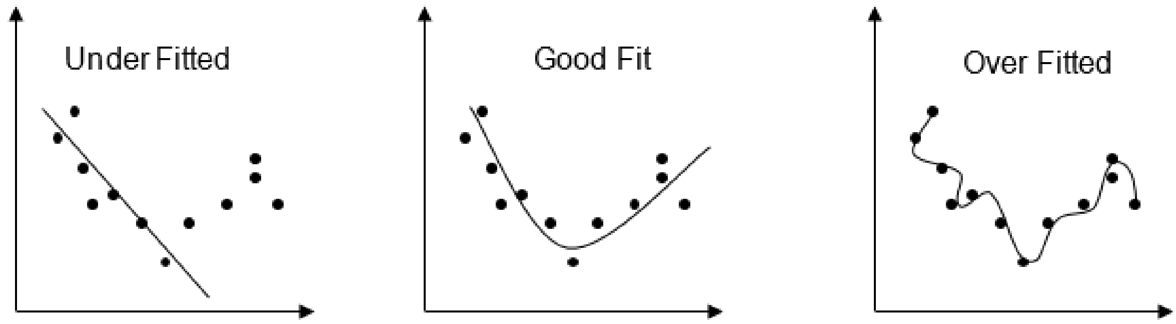

Figure 4.9: Regularization

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 26, 26, 64)        640
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 24, 24, 32)        18464
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 12, 12, 32)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 4608)              0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                46090
=================================================================
Total params: 65,194
Trainable params: 65,194
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.10: Model summary

$$softmax(\, y_i\,) \;=\; \frac{exp(\, y_i\,)}{\sum_j^C \; exp(y_j)} \; where\ i = is\ the\ class\ 0{,}1.\ldots.9$$

Figure 4.12: Expression to calculate probability

$$H(y', y) \;=\; -\sum_i \; y' log(softmax(y_i))$$

Figure 4.13: Expression to calculate loss

Figure 4.14: Cross-entropy loss vs. predicted probability

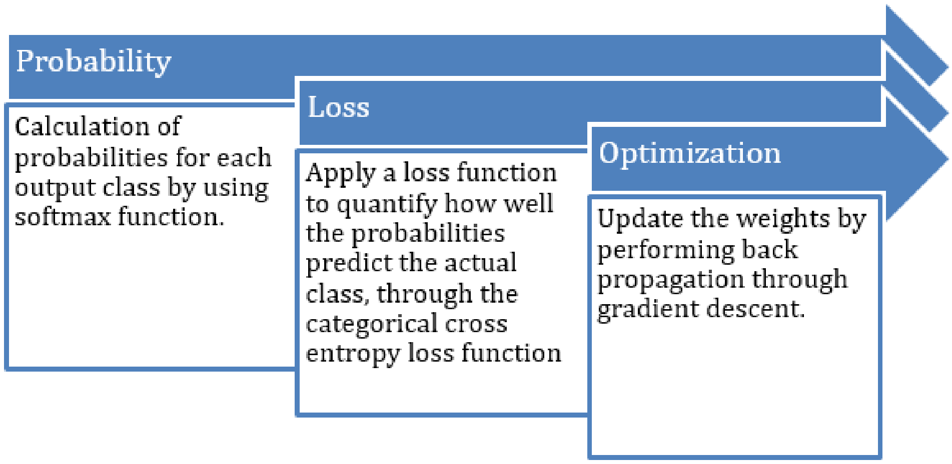

Figure 4.15: Steps for task of classification

```
array([[0., 1., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 1., 0.]])
```

Figure 4.16: Array output

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 209s 3ms/step - loss: 11.8406 - acc: 0.2646 - val_loss: 11.0491 - val_
acc: 0.3130
Epoch 2/12
60000/60000 [==============================] - 197s 3ms/step - loss: 9.8795 - acc: 0.3867 - val_loss: 9.8567 - val_ac
c: 0.3884
Epoch 3/12
60000/60000 [==============================] - 199s 3ms/step - loss: 9.8271 - acc: 0.3901 - val_loss: 9.7647 - val_ac
c: 0.3940
Epoch 4/12
60000/60000 [==============================] - 227s 4ms/step - loss: 9.6686 - acc: 0.4000 - val_loss: 9.6117 - val_ac
c: 0.4033
```

Figure 4.17: Training the model

```
Test loss: 6.17029175567627
Test accuracy: 0.6169
```

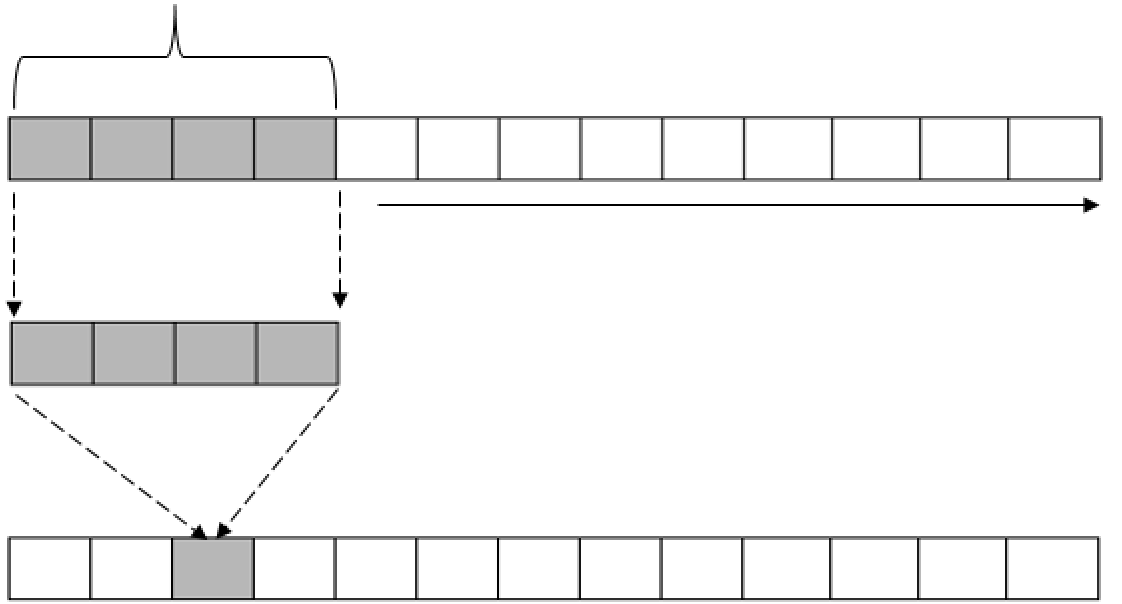

Figure 4.19: One-dimensional convolution

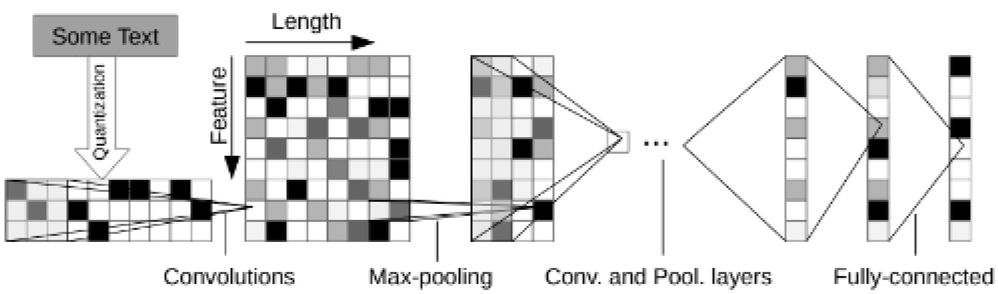

Figure 4.20: CNN with 6 convolutional and 3 fully connected layers

```
Test loss: 2.2279047027615064
Test accuracy: 0.43232413178984863
```

Figure 4.21: Accuracy score

Figure 4.22: Facial Recognition

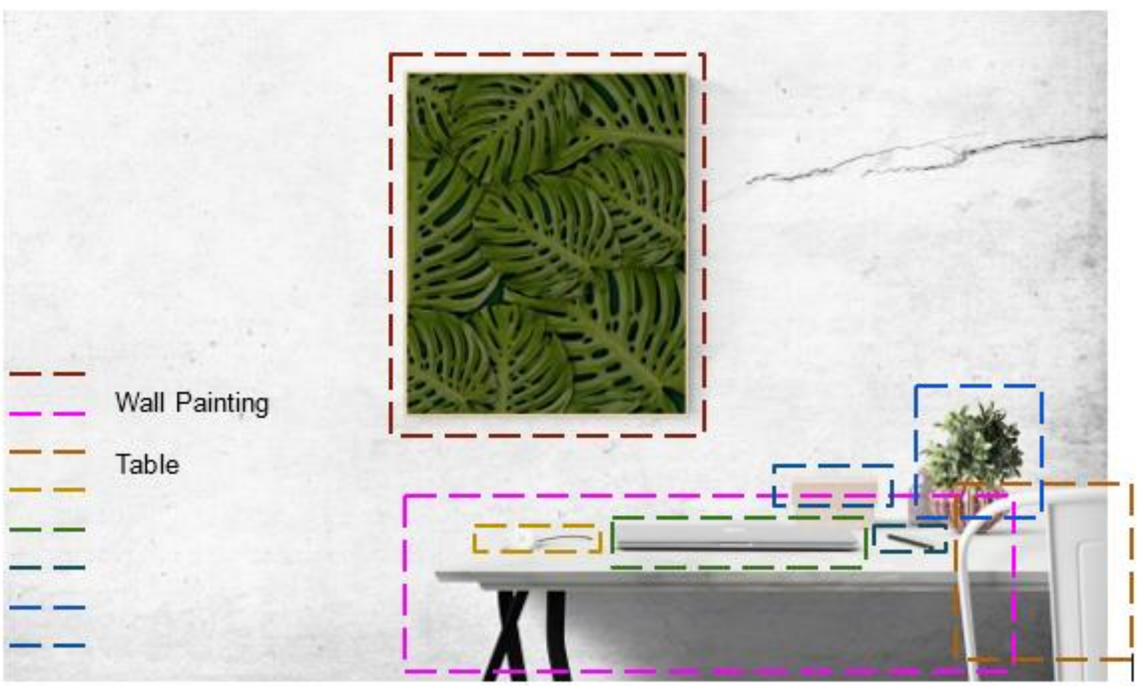

Figure 4.23: Object detection

Figure 4.24: Image captioning

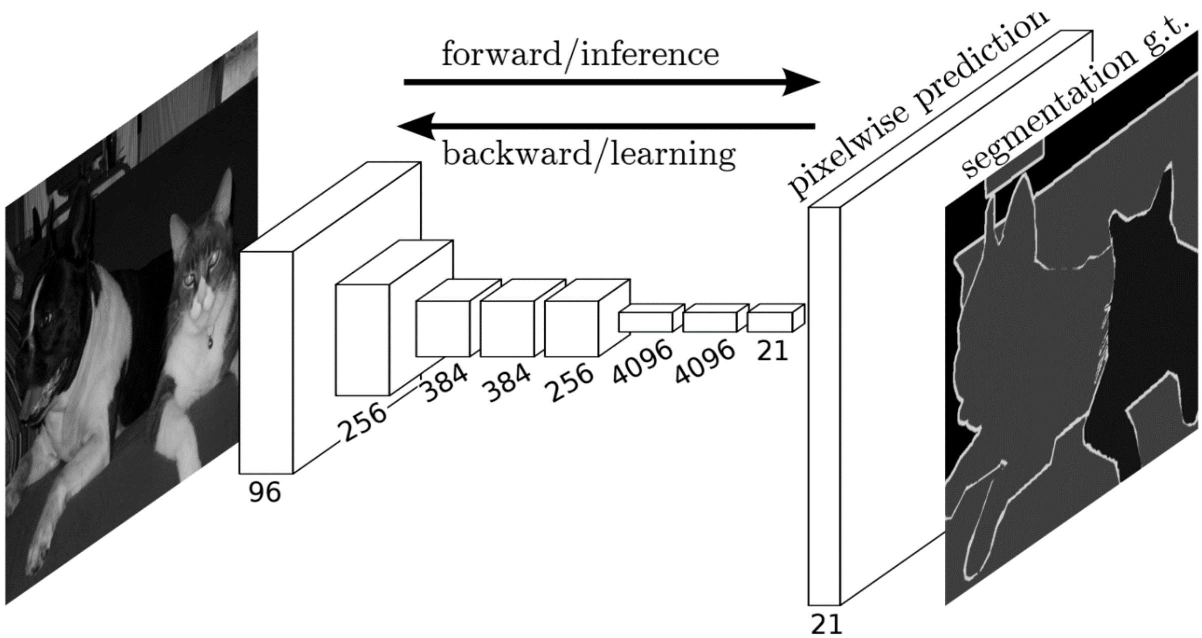

Figure 4.25: Semantic segmentation

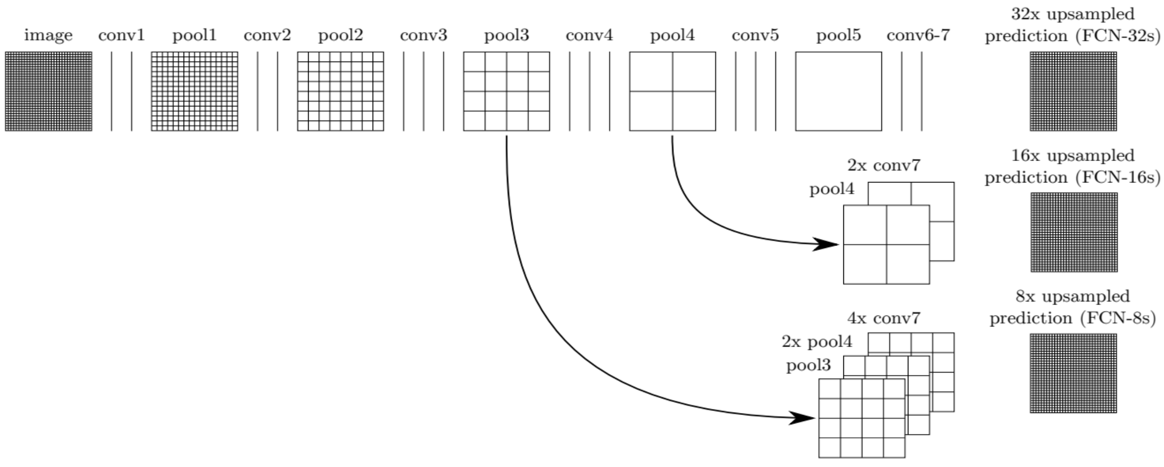

Figure 4.26: Sample architecture of semantic segmentation

```
Training Accuracy: 1.0000
Testing Accuracy:  0.8167
```

Figure 4.27: Accuracy scores

Lesson 05: Recurrent Neural Networks

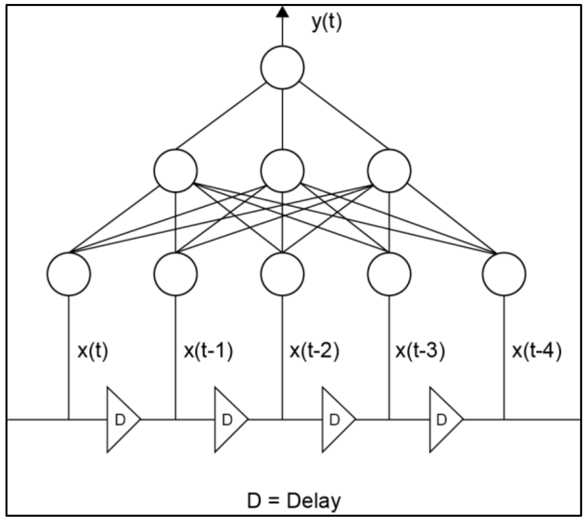

Figure 5.1: TDNN structure

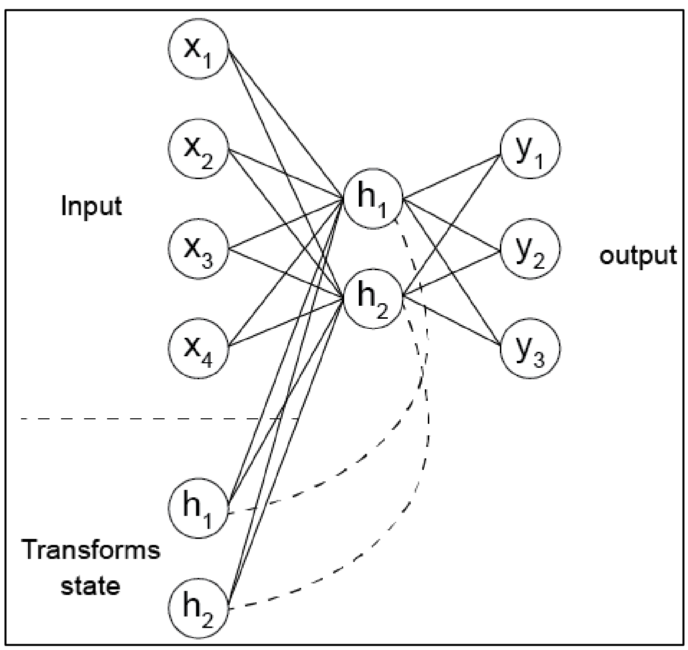

Figure 5.2: SimpleRNN structure

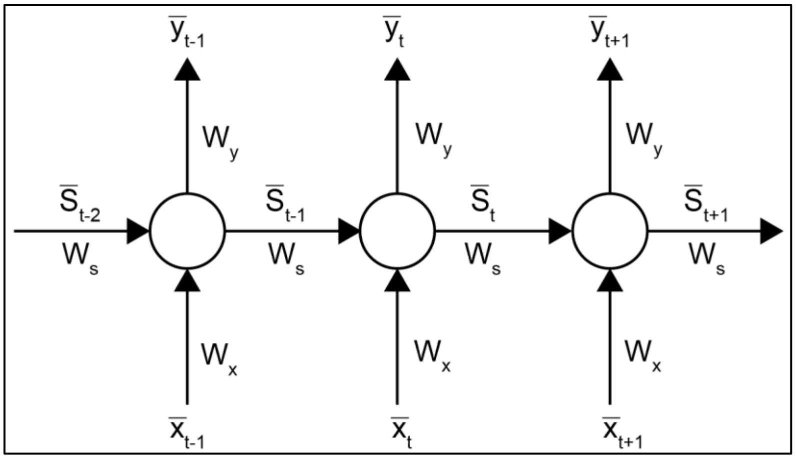

Figure 5.3: RNN structure

$$\boxed{\bar{y}_t = F(\bar{x}_t, W)}$$

Figure 5.4: Expression for the output of an RNN

$$\boxed{\bar{y}_t = F(\bar{x}_t, \bar{x}_{t-1}, \bar{x}_{t-2}, \cdots, \bar{x}_{t-t_0}, W)}$$

Figure 5.5: Expression for the output of an RNN at time t

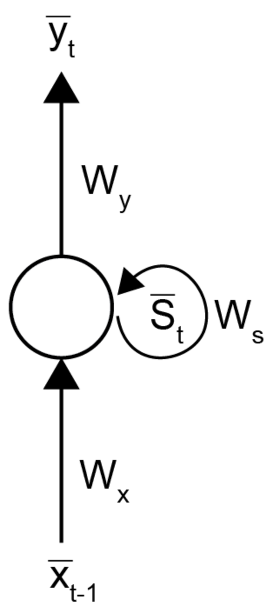

Figure 5.6: Folded model of an RNN

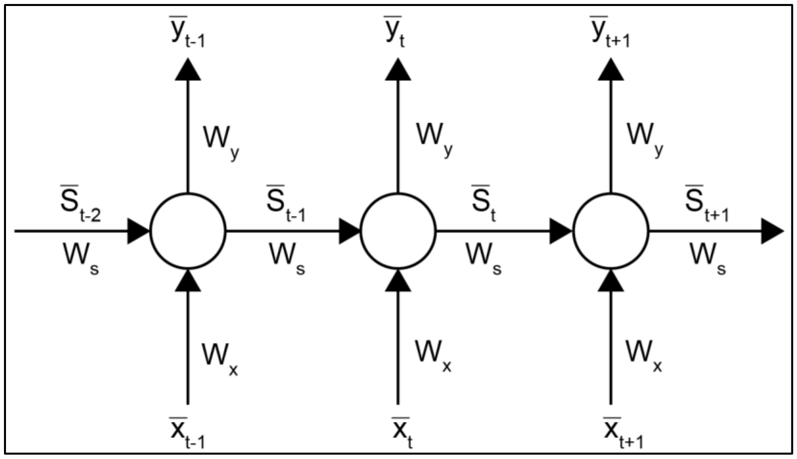

Figure 5.7: Unfolding of an RNN

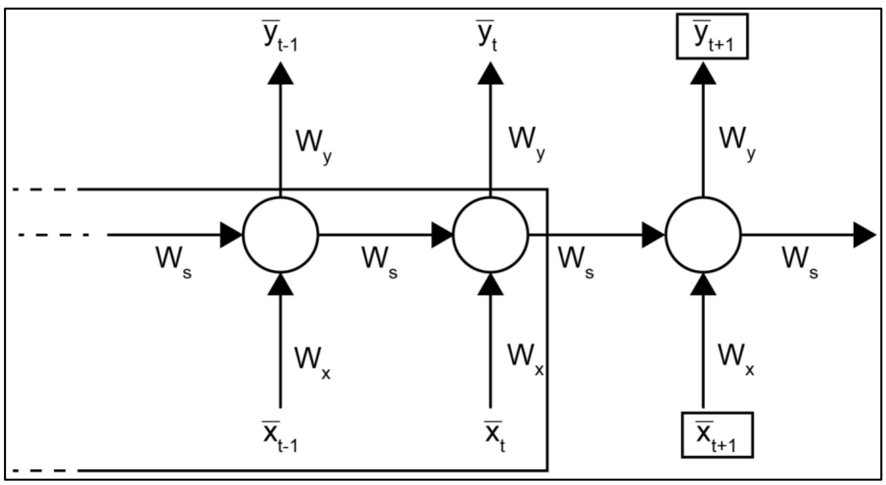

Figure 5.8: Unfolded RNN

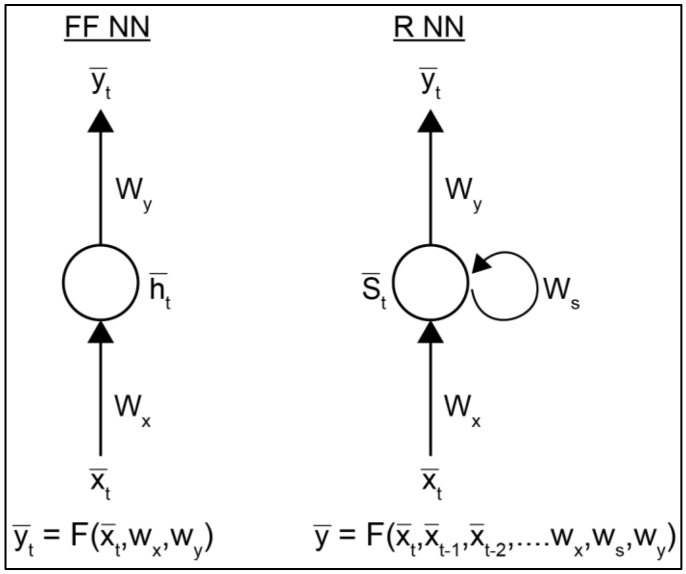

Figure 5.9: Differences between FFNNs and RNNs

$$\overline{h}_t = \overline{x}_t . w_x \qquad \overline{S}_t = \overline{X}_t . W_x + \overline{S}_{t-1} . W_s$$

$$\overline{x}_t w_x + (\overline{x}_{t-1} w_x, \overline{S}_{t-2} w_s)\, w_s$$

$$\overline{y}_t = \overline{h}_t . w_y \qquad \overline{y}_t = s_t . w_y$$

Figure 5.10: Output expressions for FFNNs and RNNs

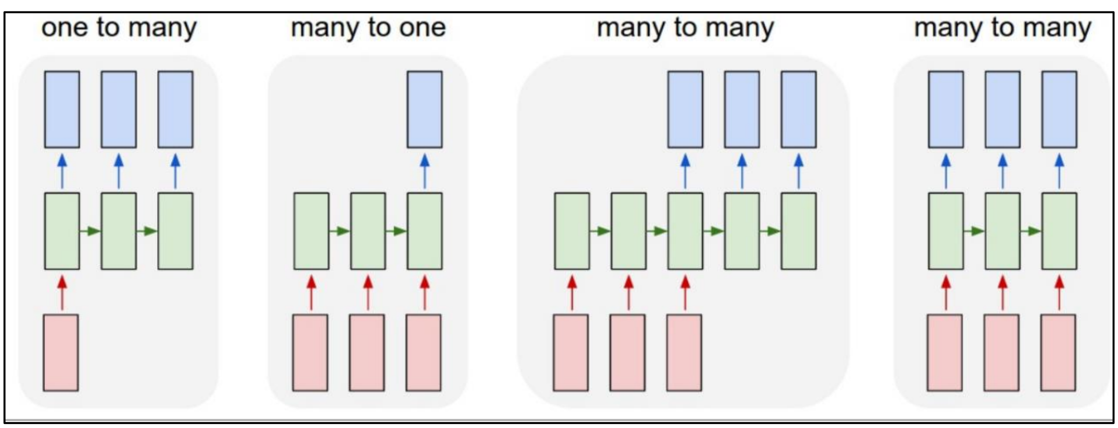

Figure 5.11 Different architectures of RNNs

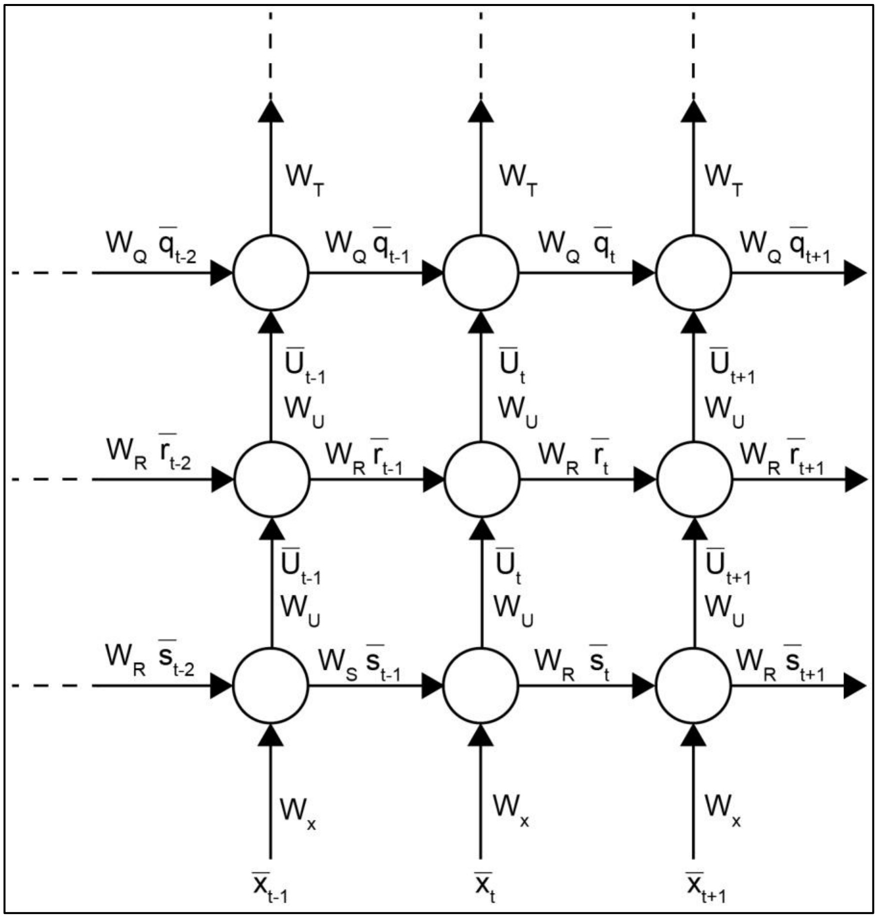

Figure 5.12: Stacked RNNs

$$\boxed{W_{new} = W_{previous} + \Delta W}$$

Figure 5.13: Expression for weight update

$$\boxed{\Delta W \ = -\alpha \frac{\partial E}{\partial W}}$$

Figure 5.14 Partial derivative of error with regards to weight

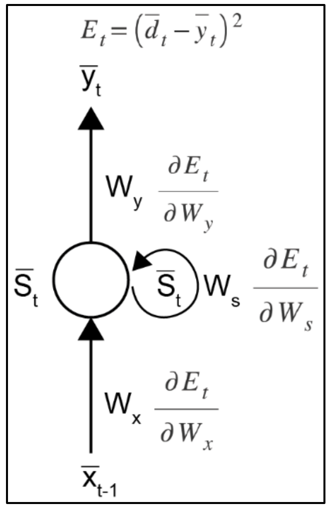

Figure 5.15: Loss function

$$E_3 = (d_3 - y_3)^2$$

Figure 5.16 Loss at time t=3

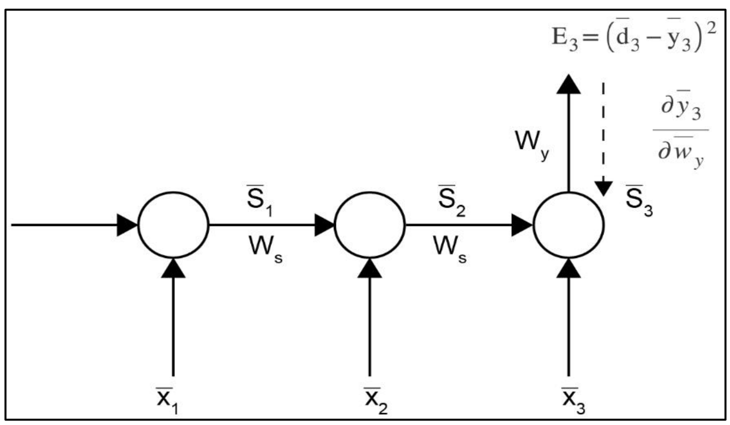

Figure 5.18: Back propagation of loss through weight matrix Wy

$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial W_y}$$

Figure 5.19: Expression for weight matrix Wy

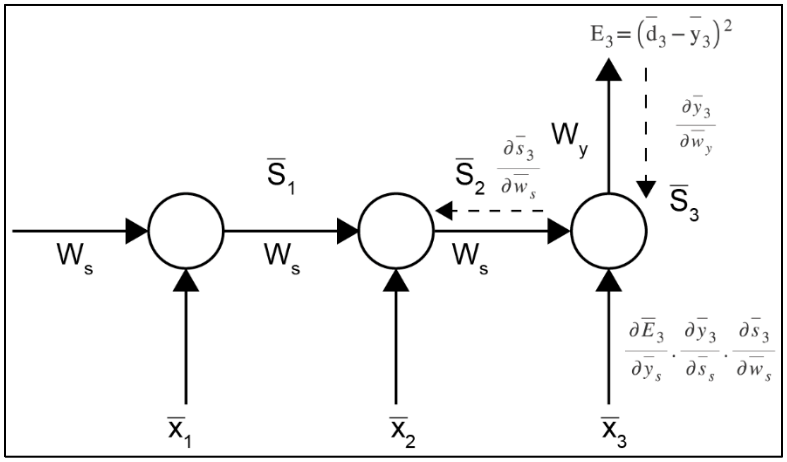

Figure 5.20: Back propagation of loss through weight matrix Ws with respect to S_3

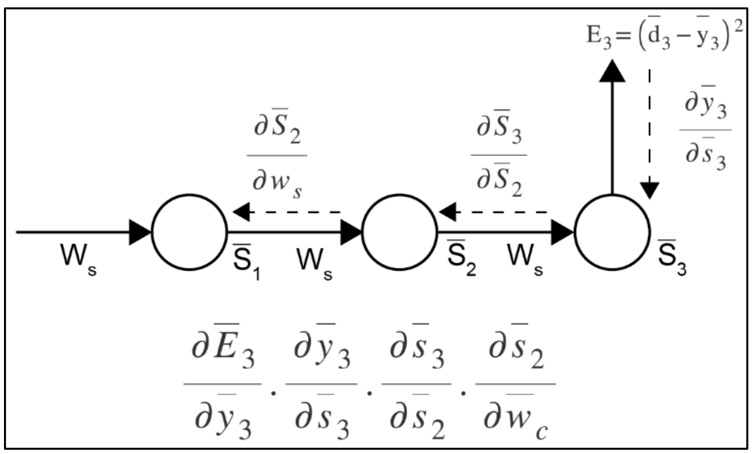

Figure 5.21: Back propagation of loss through weight matrix Ws with respect to S2

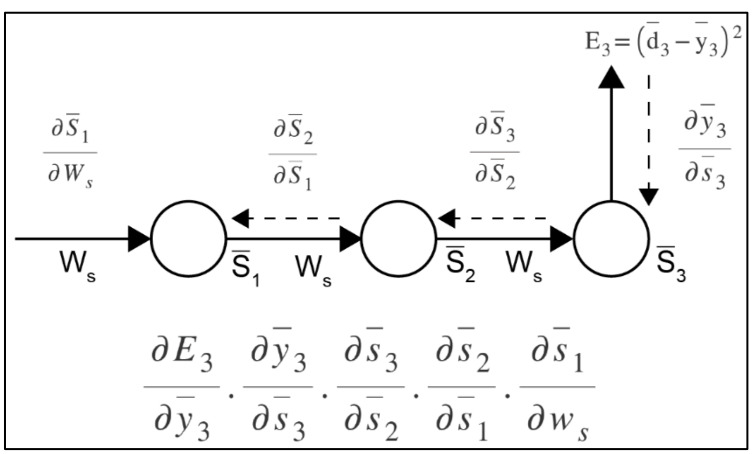

Figure 5.22: Back propagation of loss through weight matrix Ws with respect to S_1

$$
\begin{aligned}
\frac{\partial E_3}{\partial W_s} = \; & \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial w_s} \\
& + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial w_s} \\
& + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial w_s}
\end{aligned}
$$

Figure 5.23: Sum of all derivatives of error with respect to Ws at t=3

$$
\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \cdot \frac{\partial \bar{y}_N}{\partial \bar{S}_i} \cdot \frac{\partial \bar{S}_i}{\partial W_s}
$$

Figure 5.24: General expression for the derivative of error with respect to Ws

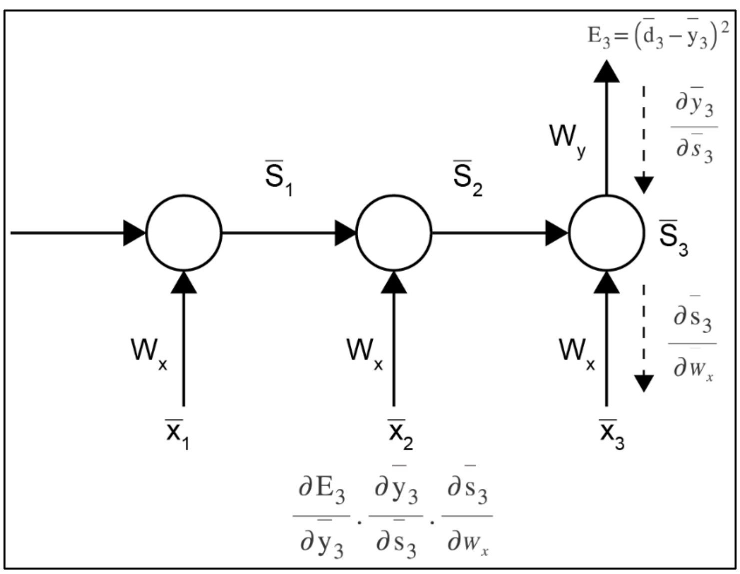

Figure 5.25: Back propagation of loss through weight matrix Wx with respect to S_2

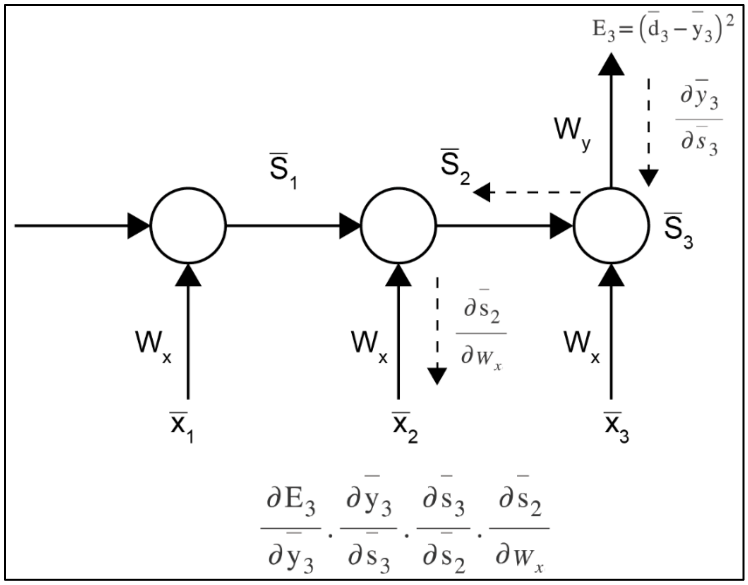

Figure 5.26: Back propagation of loss through weight matrix Wx with respect to S_2

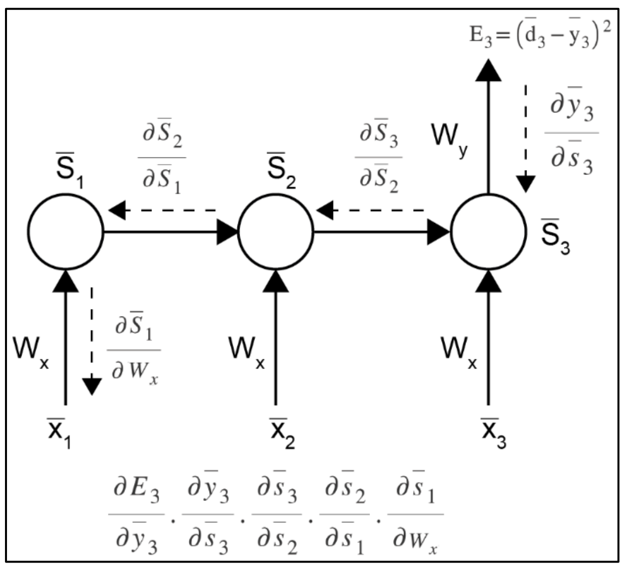

Figure 5.27: Back propagation of loss through weight matrix Wx with respect to S_1

$$
\begin{aligned}
\frac{\partial E_3}{\partial W_x} = \; & \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W_x} \\
& + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W_x} \\
& + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W_x}
\end{aligned}
$$

Figure 5.28: Sum of all derivatives of error with respect to Wx at t=3

$$
\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \cdot \frac{\partial \bar{y}_N}{\partial \bar{S}_i} \cdot \frac{\partial \bar{S}_i}{\partial W_x}
$$

Figure 5.29: General expression of derivative of error with respect to Wx

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 64)                10560
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_2 (Dense)              (None, 100)               6500
=================================================================
Total params: 21,220
Trainable params: 21,220
Non-trainable params: 0
_________________________________________________________________
```

Figure 5.30: Model summary for model layers

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_3 (SimpleRNN)     (None, 10, 64)            10560
_________________________________________________________________
dense_5 (Dense)              (None, 10, 64)            4160
_________________________________________________________________
dense_6 (Dense)              (None, 10, 100)           6500
=================================================================
Total params: 21,220
Trainable params: 21,220
Non-trainable params: 0
_________________________________________________________________
```

Figure 5.31: Model summary of sequence-returning model

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_5 (SimpleRNN)     (None, 1000, 64)          10560
_________________________________________________________________
dense_9 (Dense)              (None, 1000, 64)          4160
_________________________________________________________________
dense_10 (Dense)             (None, 1000, 100)         6500
=================================================================
Total params: 21,220
Trainable params: 21,220
Non-trainable params: 0
_________________________________________________________________
```

Figure 5.32: Model summary for timesteps

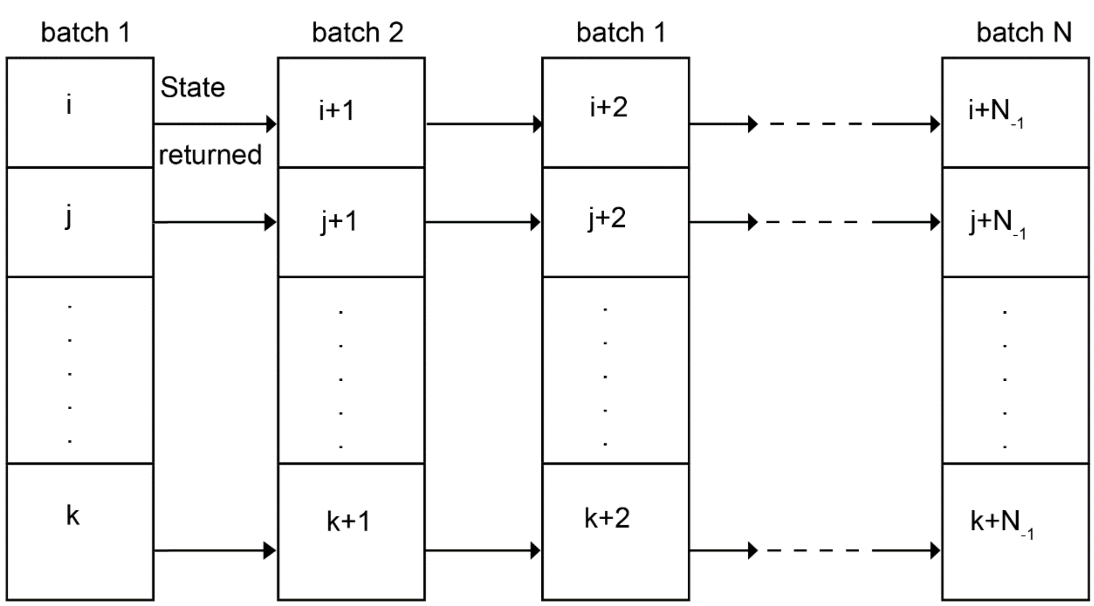

Figure 5.33 Batch formations for stateful RNN

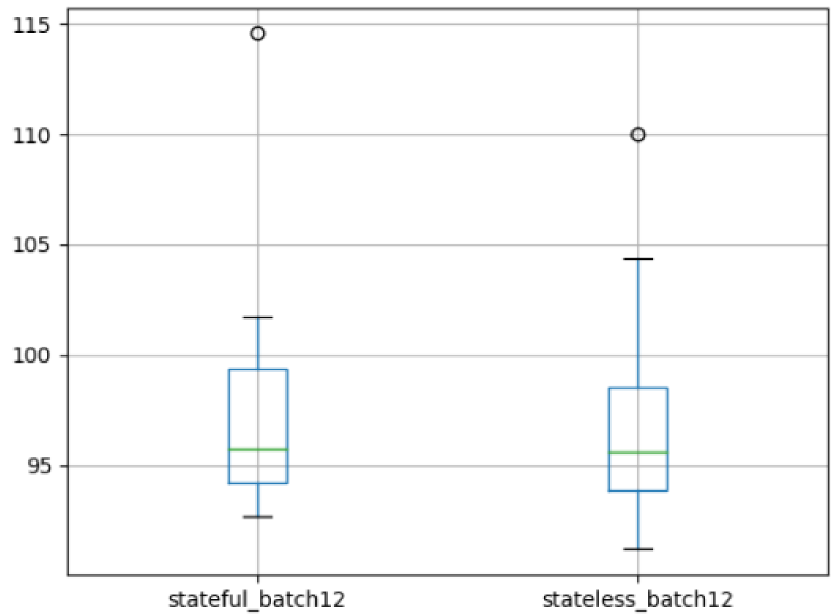

Figure 5.34: Box and whisker plot for stateful vs stateless

```
Paper 5 is predicted to have been written by Author A, 6142 to 5612
Paper 4 is predicted to have been written by Author B, 5215 to 4558
Paper 1 is predicted to have been written by Author B, 13924 to 6850
Paper 3 is predicted to have been written by Author B, 7620 to 5764
Paper 2 is predicted to have been written by Author B, 12840 to 6806
```

Figure 5.35: Output for author attribution

Lesson 06: Gated Recurrent Units (GRUs)

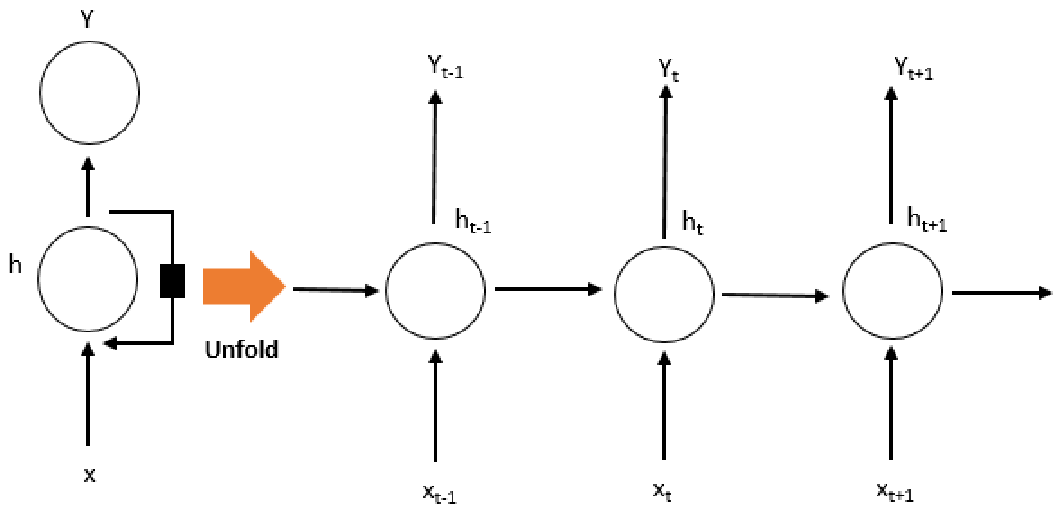

Figure 6.1: A basic RNN

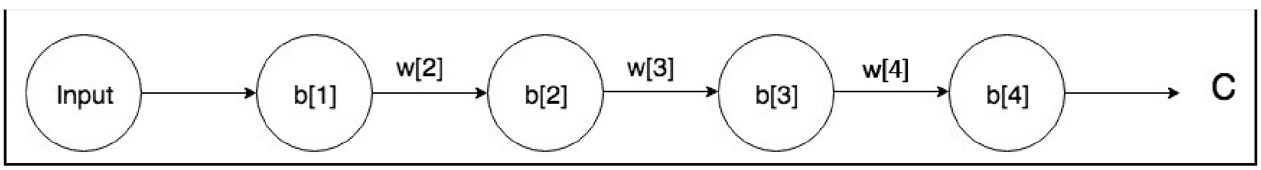

Figure 6.2: A simple neural network

grad(C, b[1]) = d(z[1]) * w[2] * d(z[2]) * w[3] * d(z[3]) * w[4] * d(z[4]) * grad(C, a [4])

Figure 6.3: Gradient calculation using chain rule

b[1] = b[1] + lambda*grad(C, b[1])

Figure 6.4: Updating value of b[1] using the gradient

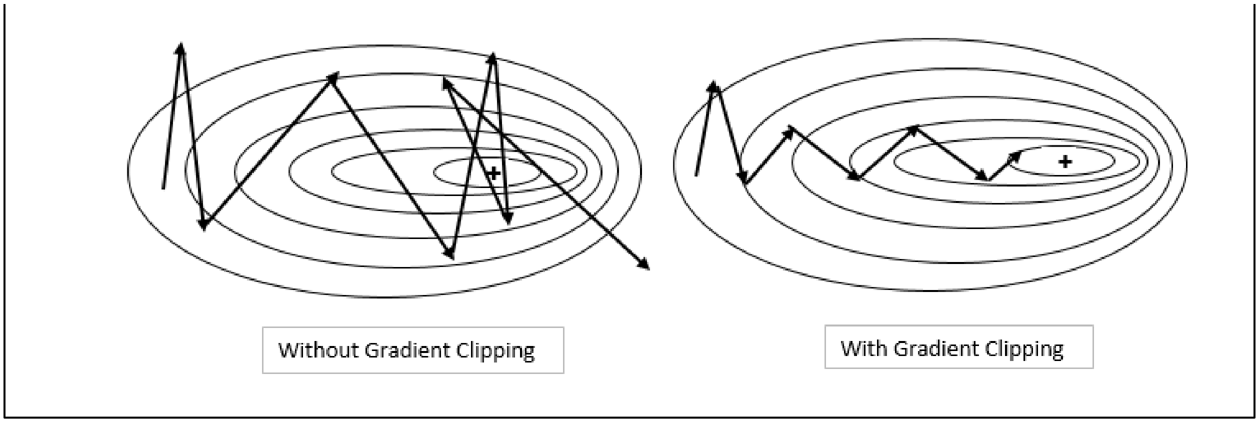

Figure 6.5: Clipping gradients to combat the explosion of gradients

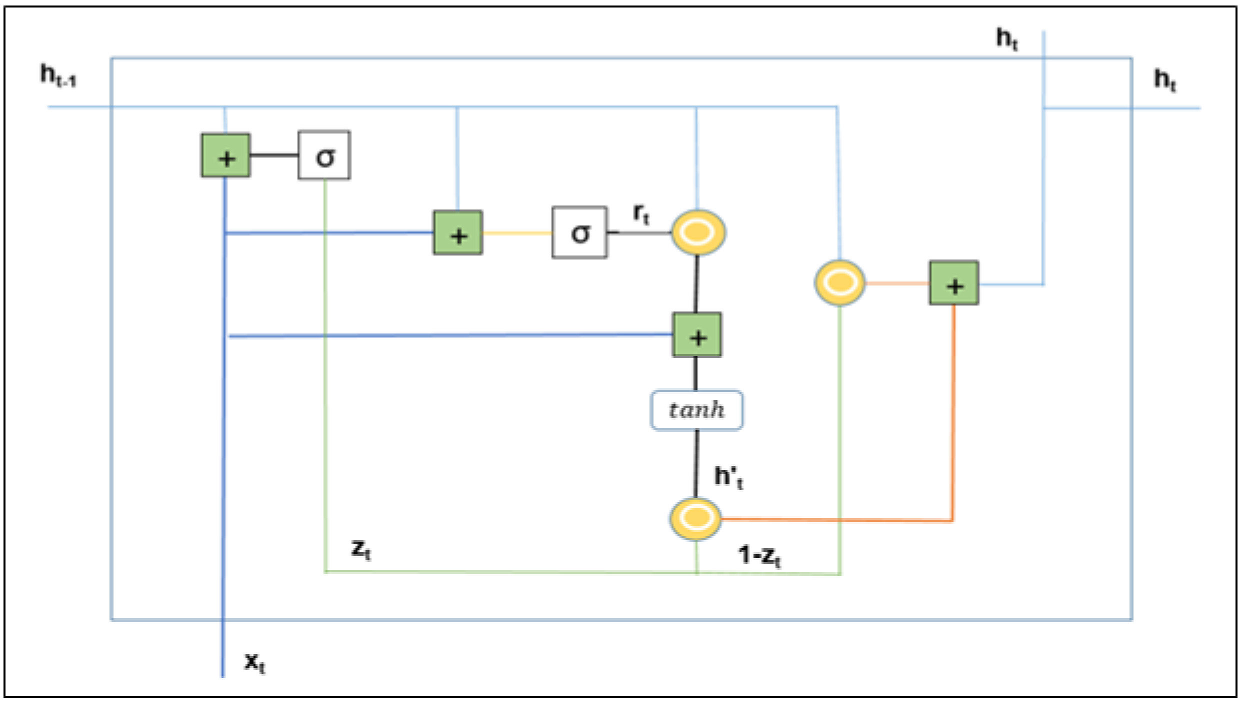

Figure 6.6: The full GRU structure

Figure 6.7: The meanings of the different signs in the GRU diagram

h[t] = hadamard{z[t], h[t-1]} + hadamard{(1 - z[t]) * h_candidate[t]}

Figure 6.8: The expression for the activation function for the next layer in terms of the candidate activation function

```
z[t] = sigmoid(W_z * x[t] + U_z * h[t-1])
```

Figure 6.9: The expression for calculating the update gate

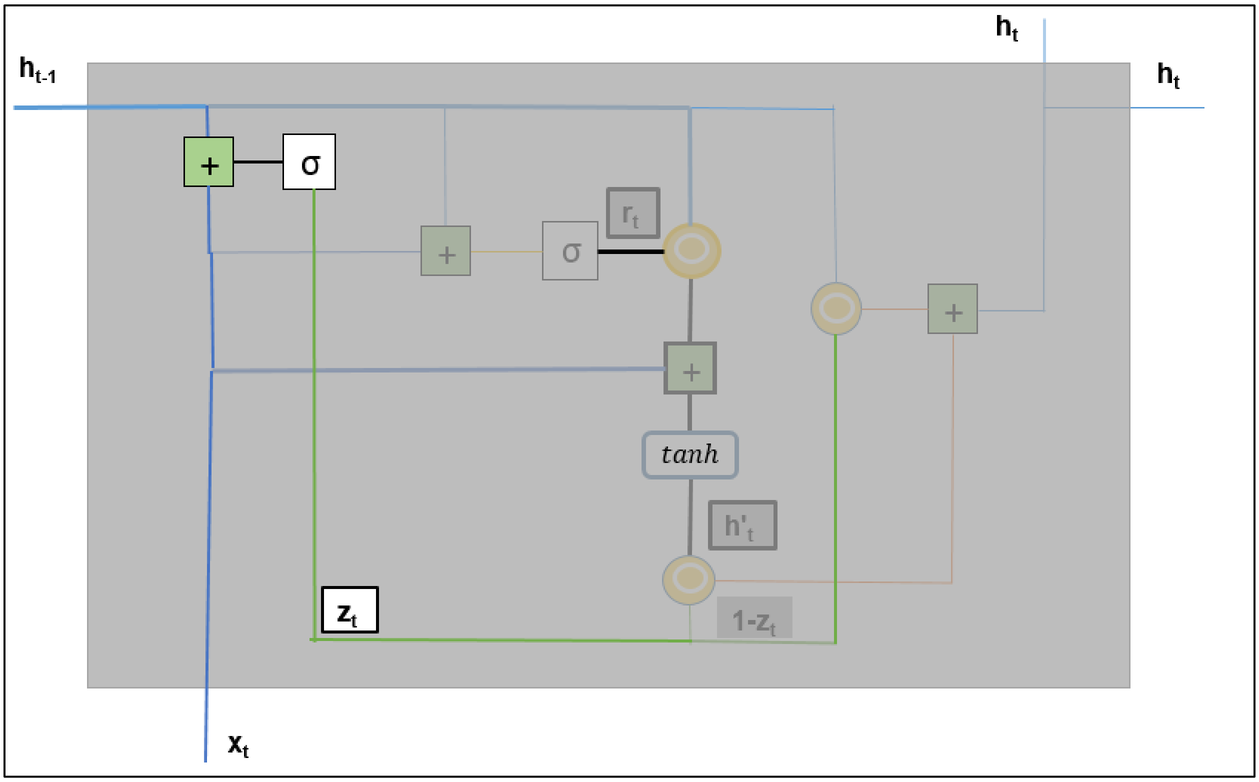

Figure 6.10: The update gate in a full GRU diagram

```
x_t
```

```
array([[-0.93576943],
       [-0.26788808],
       [ 0.53035547],
       [-0.69166075],
       [-0.39675353]])
```

```
h_prev
```

```
array([[ 0.90085595],
       [-0.68372786],
       [-0.12289023]])
```

```
W_z
```

```
array([[ 1.62434536, -0.61175641, -0.52817175, -1.07296862,  0.86540763],
       [-2.3015387 ,  1.74481176, -0.7612069 ,  0.3190391 , -0.24937038],
       [ 1.46210794, -2.06014071, -0.3224172 , -0.38405435,  1.13376944]])
```

```
U_z
```

```
array([[-1.09989127, -0.17242821, -0.87785842],
       [ 0.04221375,  0.58281521, -1.10061918],
       [ 1.14472371,  0.90159072,  0.50249434]])
```

Figure 6.11: A screenshot displaying the weights and activation functions

r[t] = sigmoid(W_r * x[t] + U_r * h[t-1])

Figure 6.12: The expression for calculating the reset gate

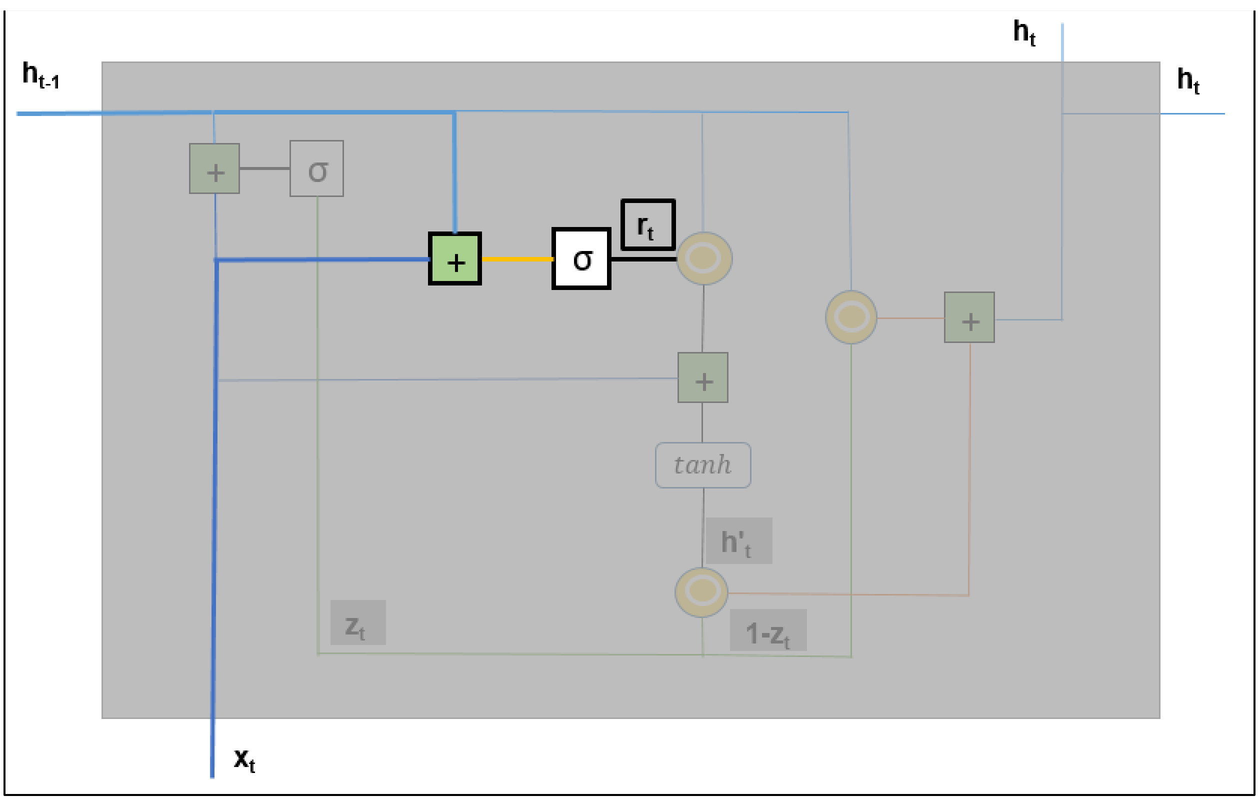

Figure 6.13: The reset gate

```
W_r
```

```
array([[-0.6871727 , -0.84520564, -0.67124613, -0.0126646 , -1.11731035],
       [ 0.2344157 ,  1.65980218,  0.74204416, -0.19183555, -0.88762896],
       [-0.74715829,  1.6924546 ,  0.05080775, -0.63699565,  0.19091548]])
```

```
U_r
```

```
array([[ 2.10025514,  0.12015895,  0.61720311],
       [ 0.30017032, -0.35224985, -1.1425182 ],
       [-0.34934272, -0.20889423,  0.58662319]])
```

Figure 6.14: A screenshot displaying the values of the weights

```
r_t
```

```
array([[0.93699927],
       [0.70392511],
       [0.5971474 ]])
```

Figure 6.15: A screenshot displaying the r_t output

h_candidate[t] = tanh(W * x[t] + U * hadamard{r[t], h[t-1]})

Figure 6.16: The expression for calculating the candidate activation function

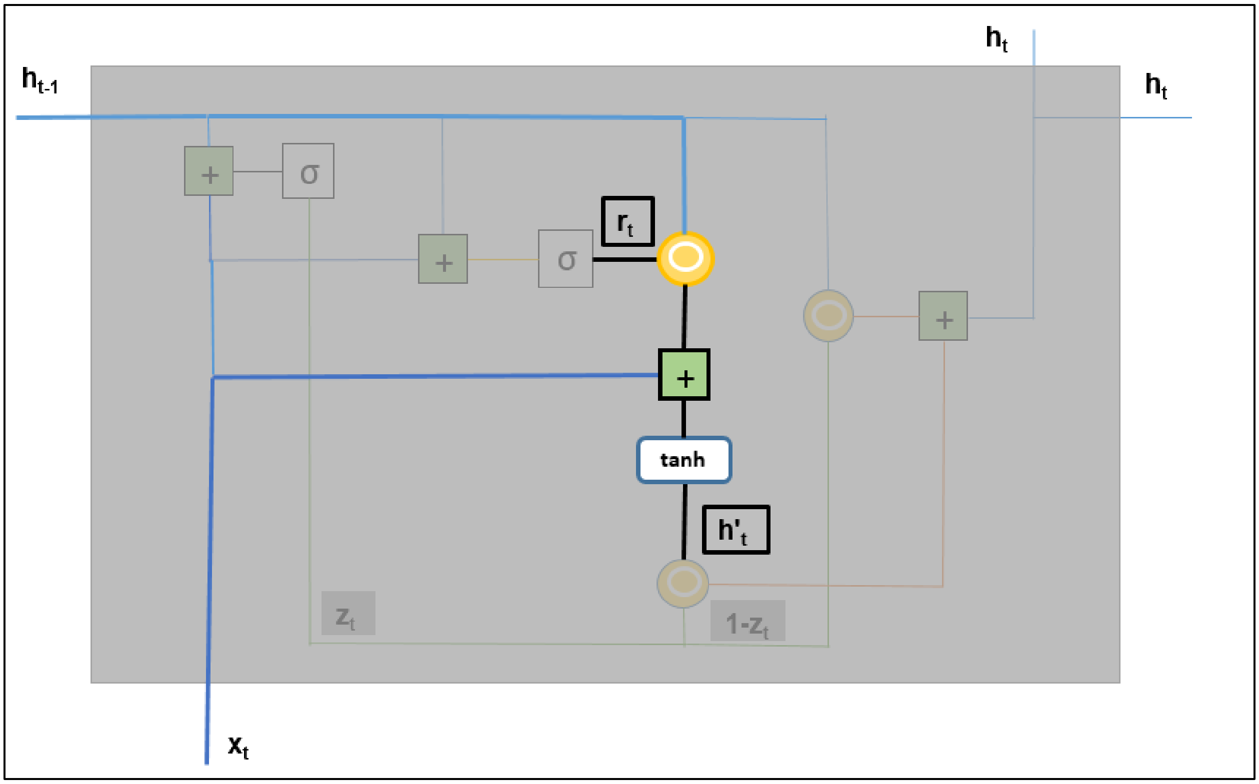

Figure 6.17: The candidate activation function

```
W
```

```
array([[ 0.83898341,  0.93110208,  0.28558733,  0.88514116, -0.75439794],
       [ 1.25286816,  0.51292982, -0.29809284,  0.48851815, -0.07557171],
       [ 1.13162939,  1.51981682,  2.18557541, -1.39649634, -1.44411381]])
```

```
U
```

```
array([[-0.50446586,  0.16003707,  0.87616892],
       [ 0.31563495, -2.02220122, -0.30620401],
       [ 0.82797464,  0.23009474,  0.76201118]])
```

Figure 6.18: A screenshot displaying how the W and U weights are defined

```
h_candidate
```

```
array([[-0.94284959],
       [-0.47277196],
       [ 0.9429634 ]])
```

Figure 6.19: A screenshot displaying the value of h_candidate

```
h_new
```

```
array([[-0.72356608],
       [-0.62428489],
       [ 0.61671542]])
```

Figure 6.20: A screenshot displaying the value of the current activation function

```
Number of train sequences:  25000
Number of test sequences:  25000
train_data shape: (25000, 500)
test_data shape: (25000, 500)
```

Figure 6.21: A screenshot showing the train and test sequences

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 53s 3ms/step - loss: 0.5382 - acc: 0.7286 - val_loss: 0.4796 - val_ac
c: 0.7620
Epoch 2/10
20000/20000 [==============================] - 53s 3ms/step - loss: 0.3120 - acc: 0.8701 - val_loss: 0.3218 - val_ac
c: 0.8732
Epoch 3/10
20000/20000 [==============================] - 51s 3ms/step - loss: 0.2503 - acc: 0.9025 - val_loss: 0.3644 - val_ac
c: 0.8720
Epoch 4/10
20000/20000 [==============================] - 51s 3ms/step - loss: 0.2187 - acc: 0.9184 - val_loss: 0.3092 - val_ac
c: 0.8740
Epoch 5/10
20000/20000 [==============================] - 51s 3ms/step - loss: 0.1937 - acc: 0.9290 - val_loss: 0.3130 - val_ac
c: 0.8792
Epoch 6/10
20000/20000 [==============================] - 51s 3ms/step - loss: 0.1747 - acc: 0.9350 - val_loss: 0.3299 - val_ac
c: 0.8710
Epoch 7/10
20000/20000 [==============================] - 52s 3ms/step - loss: 0.1600 - acc: 0.9434 - val_loss: 0.3599 - val_ac
c: 0.8500
Epoch 8/10
20000/20000 [==============================] - 53s 3ms/step - loss: 0.1498 - acc: 0.9458 - val_loss: 0.3378 - val_ac
c: 0.8792
Epoch 9/10
20000/20000 [==============================] - 53s 3ms/step - loss: 0.1389 - acc: 0.9512 - val_loss: 0.5470 - val_ac
c: 0.8308
Epoch 10/10
20000/20000 [==============================] - 53s 3ms/step - loss: 0.1284 - acc: 0.9541 - val_loss: 0.3599 - val_ac
c: 0.8672
```

Figure 6.22: A screenshot displaying the variable history output of the training model

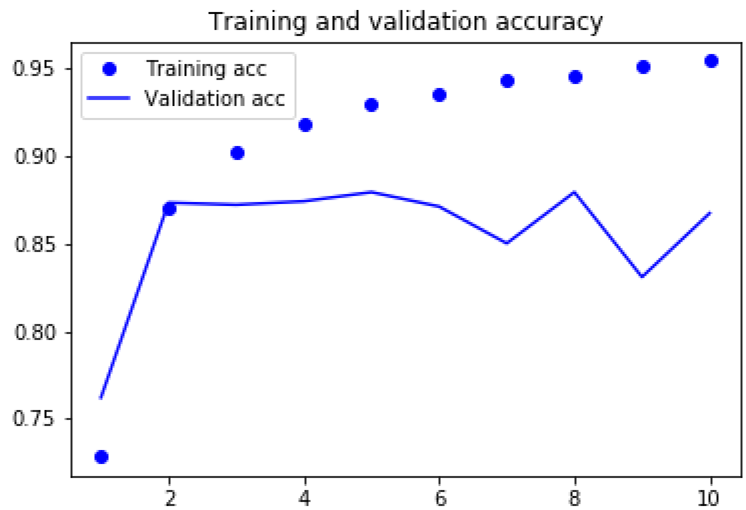

Figure 6.23: The training and validation accuracy for the sentiment classification task

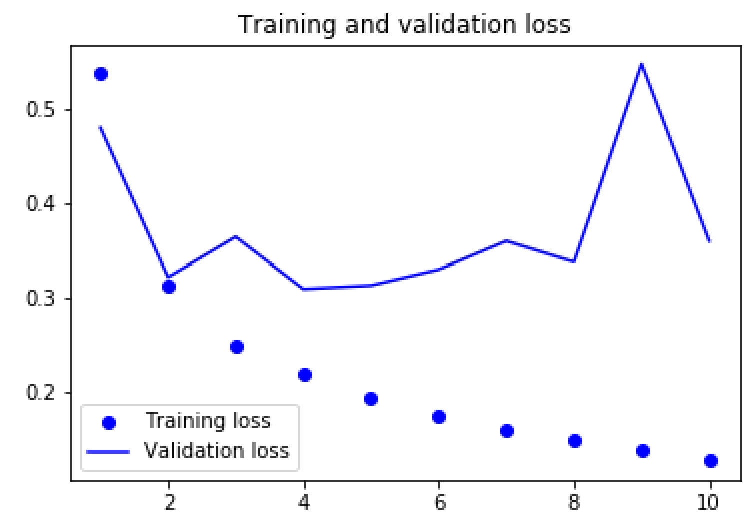

Figure 6.24: The training and validation loss for the sentiment classification task

```
THE SONNETS

by William Shakespeare

From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his mem
```

Figure 6.25: A screenshot of THE SONNETS

```
'\n\nFrom fairest creatures we desire incre',
'rom fairest creatures we desire increase',
' fairest creatures we desire increase,\nT',
'irest creatures we desire increase,\nThat',
'st creatures we desire increase,\nThat th',
'creatures we desire increase,\nThat there',
'atures we desire increase,\nThat thereby ',
'res we desire increase,\nThat thereby bea',
' we desire increase,\nThat thereby beauty',
" desire increase,\nThat thereby beauty's ",
"sire increase,\nThat thereby beauty's ros",
"e increase,\nThat thereby beauty's rose m",
"ncrease,\nThat thereby beauty's rose migh",
"ease,\nThat thereby beauty's rose might n",
"e,\nThat thereby beauty's rose might neve",
"That thereby beauty's rose might never d",
"t thereby beauty's rose might never die,",
```

Figure 6.26: A screenshot of the training sequences

```
Epoch 1/10
31327/31327 [==============================] - 12s 374us/step - loss: 2.2844
Epoch 2/10
31327/31327 [==============================] - 11s 335us/step - loss: 1.8985
Epoch 3/10
31327/31327 [==============================] - 11s 339us/step - loss: 1.7675
Epoch 4/10
31327/31327 [==============================] - 12s 372us/step - loss: 1.6757
Epoch 5/10
31327/31327 [==============================] - 11s 353us/step - loss: 1.5984
Epoch 6/10
31327/31327 [==============================] - 11s 341us/step - loss: 1.5479
Epoch 7/10
31327/31327 [==============================] - 12s 382us/step - loss: 1.5083
Epoch 8/10
31327/31327 [==============================] - 11s 346us/step - loss: 1.4803
Epoch 9/10
31327/31327 [==============================] - 11s 354us/step - loss: 1.4648
Epoch 10/10
31327/31327 [==============================] - 11s 356us/step - loss: 1.4428
```

Figure 6.27: A screenshot displaying epochs

```
' thou viewest,\nNow is the time that faced padince thy fete,\njevery bnuping griats I have liking dispictreessedg.\n
\nThy such thy sombeliner  h'
```

Figure 6.28: A screenshot displaying the output of the generated poem sequence

Lesson 07: Long Short-Term Memory (LSTM)

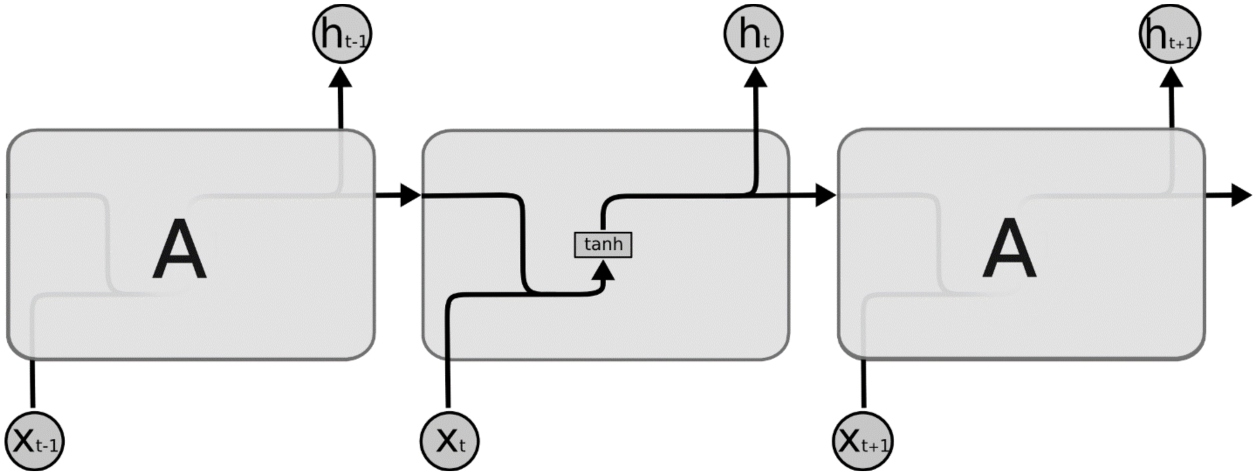

Figure 7.1: The repeating module in a standard RNN

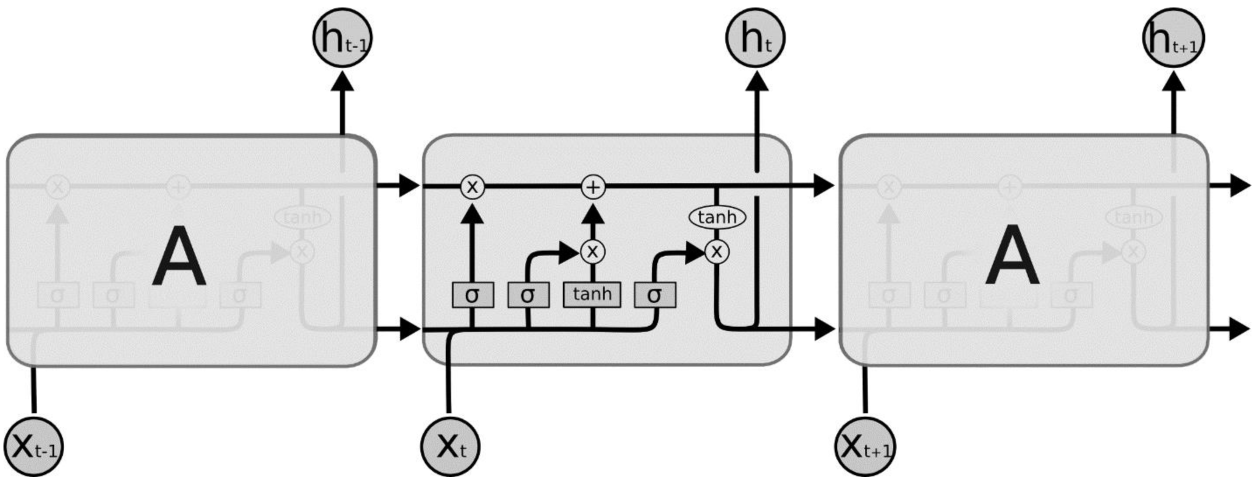

Figure 7.2: The LSTM unit

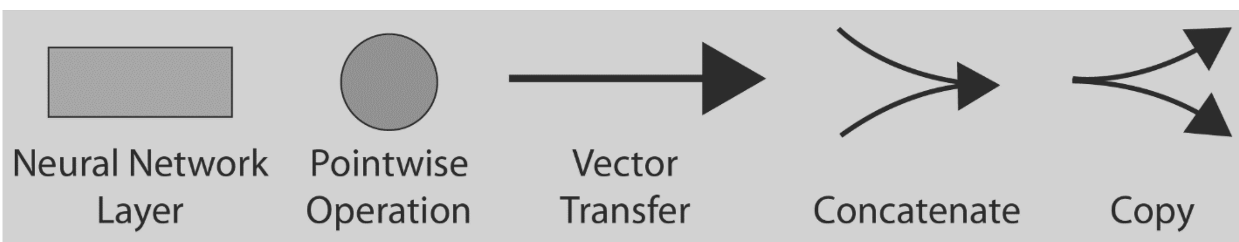

Figure 7.3: Notations used in the model

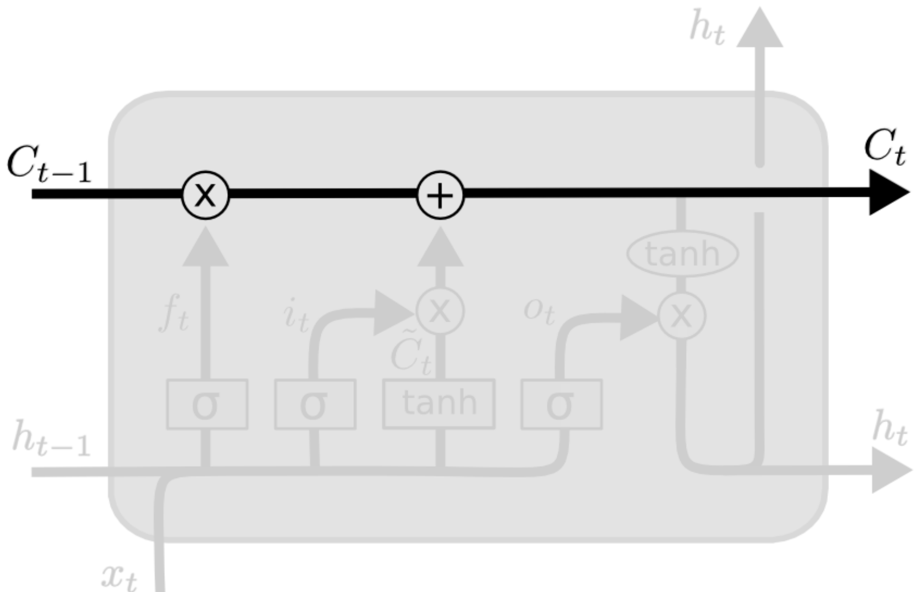

Figure 7.4: Cell state

$$f[t] = \text{sigmoid}(w\_f * x[t] + U\_f * h[t-1])$$

Figure 7.5: Expression for the forget gate

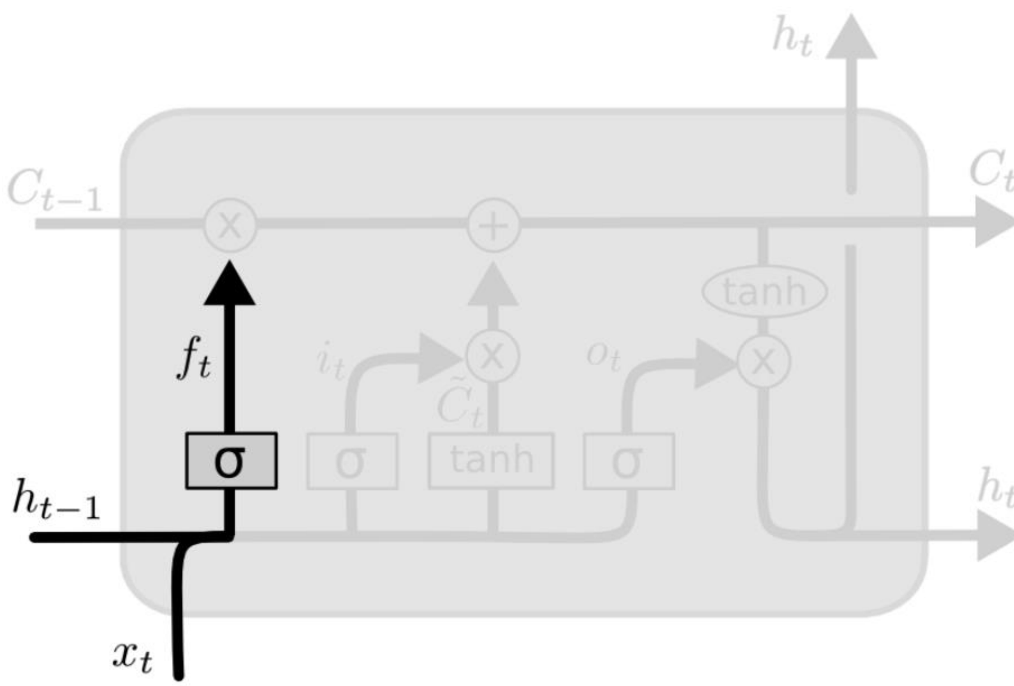

Figure 7.6: The forget gate

```
h_prev
```

```
array([[1.76405235],
       [0.40015721],
       [0.97873798]])
```

```
x
```

```
array([[ 2.2408932 ],
       [ 1.86755799],
       [-0.97727788],
       [ 0.95008842],
       [-0.15135721]])
```

Figure 7.7: Output for the previous state, 'h_prev,' and the current input, 'x'

```
W_f
```

```
array([[-0.10321885,  0.4105985 ,  0.14404357,  1.45427351,
0.76103773],
       [ 0.12167502,  0.44386323,  0.33367433,  1.49407907, -
0.20515826],
       [ 0.3130677 , -0.85409574, -2.55298982,  0.6536186 ,
0.8644362 ]])
```

```
U_f
```

```
array([[-0.74216502,  2.26975462, -1.45436567],
       [ 0.04575852, -0.18718385,  1.53277921],
       [ 1.46935877,  0.15494743,  0.37816252]])
```

Figure 7.8: Output of the matrix values

```
f
```

```
array([[0.45930054],
       [0.97661676],
       [0.99403442]])
```

Figure 7.9: Output of the forget gate, f[t]

$$C\_candidate = \tanh(W\_c * h[t-1] + U\_c * x[t])$$

Figure 7.10: Expression for candidate cell state

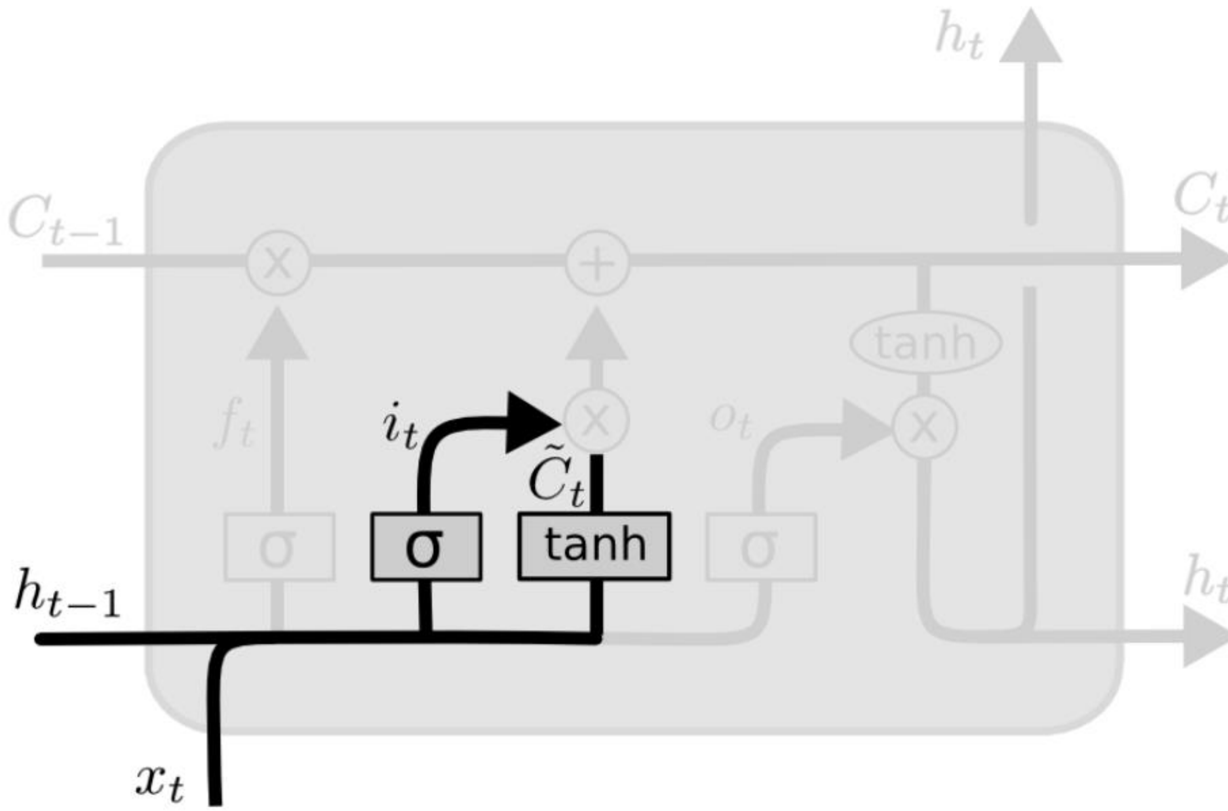

Figure 7.11: Input gate and candidate state

$$i[t] = \mathrm{sigmoid}(W\_i * x[t] + U\_i * h[t-1])$$

Figure 7.12: Expression for the input gate value

```
W_i
```

```
array([[-0.88778575, -1.98079647, -0.34791215,  0.15634897,
1.23029068],
       [ 1.20237985, -0.38732682, -0.30230275, -1.04855297, -
1.42001794],
       [-1.70627019,  1.9507754 , -0.50965218, -0.4380743 , -
1.25279536]])
```

```
U_i
```

```
array([[ 0.77749036, -1.61389785, -0.21274028],
       [-0.89546656,  0.3869025 , -0.51080514],
       [-1.18063218, -0.02818223,  0.42833187]])
```

Figure 7.13: Screenshot of values of matrices for candidate cell state and input gate

```
i
```

```
array([[0.00762368],
       [0.39184172],
       [0.17027909]])
```

Figure 7.14: Screenshot of output of input gate

```
W_c
```

```
array([[ 0.06651722,  0.3024719 , -0.63432209, -0.36274117, -
0.67246045],
       [-0.35955316, -0.81314628, -1.7262826 ,  0.17742614, -
0.40178094],
       [-1.63019835,  0.46278226, -0.90729836,  0.0519454 ,
0.72909056]])
```

```
U_c
```

```
array([[ 0.12898291,  1.13940068, -1.23482582],
       [ 0.40234164, -0.68481009, -0.87079715],
       [-0.57884966, -0.31155253,  0.05616534]])
```

Figure 7.15: Screenshot for values of matrices W_c and U_c

```
c_candidate
```

```
array([[ 0.51233992],
       [-0.67747899],
       [-0.99555958]])
```

Figure 7.16: Screenshot of the candidate cell state

C[t]=hadamard(f[t], C[t-1]) + hadamard(i[t], C_candidate[t])

Figure 7.17: Expression for cell state update

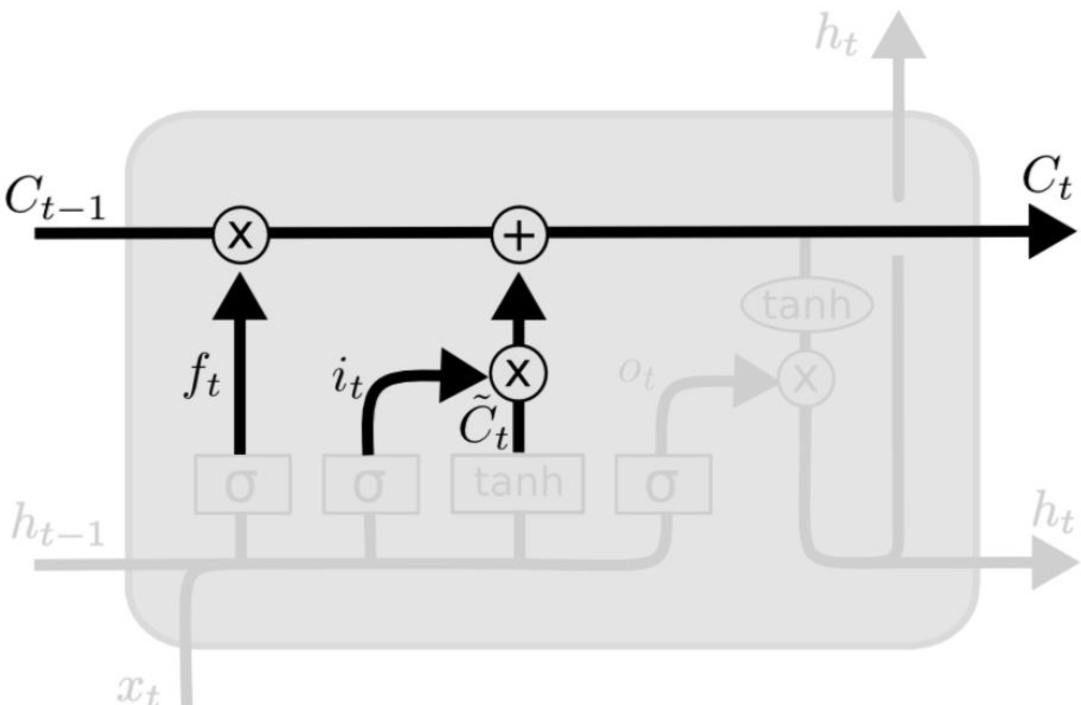

Figure 7.18: Updated cell state

```
c_new
```

```
array([[-0.53124803],
       [ 0.61429771],
       [ 0.29336152]])
```

Figure 7.19: Screenshot for output of updated cell state

$$o[t] = sigmoid(W\_o*x[t] + U\_o*h[t-1])$$

Figure 7.20: Expression for output gate.

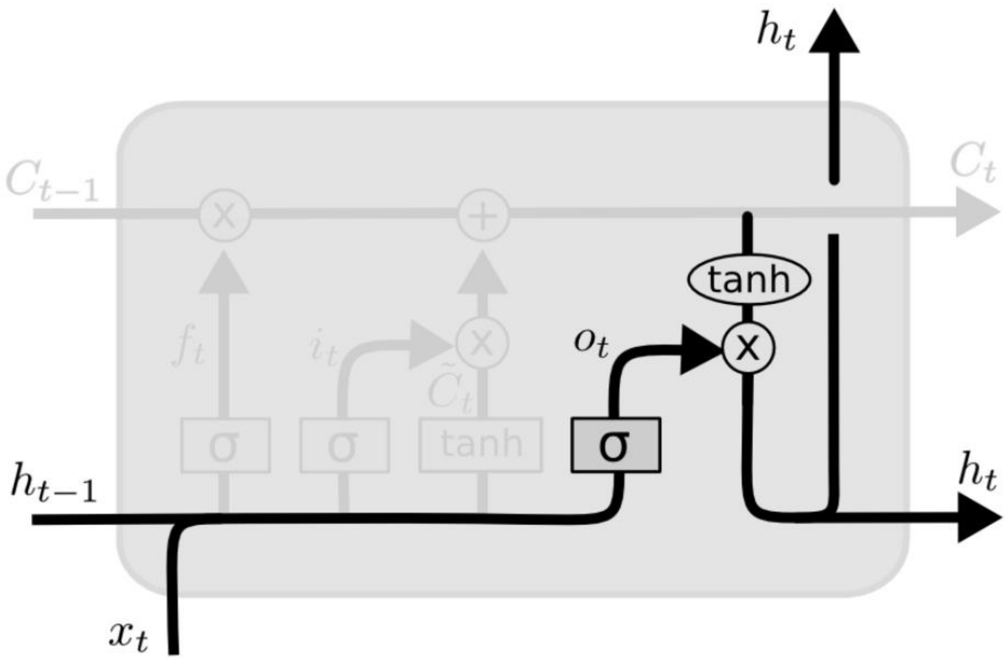

Figure 7.21: Output gate and current activation

```
W_o
```

```
array([[-1.16514984,  0.90082649,  0.46566244, -1.53624369,
1.48825219],
       [ 1.89588918,  1.17877957, -0.17992484, -1.07075262,
1.05445173],
       [-0.40317695,  1.22244507,  0.20827498,  0.97663904,
0.3563664 ]])
```

```
U_o
```

```
array([[ 0.70657317,  0.01050002,  1.78587049],
       [ 0.12691209,  0.40198936,  1.8831507 ],
       [-1.34775906, -1.270485  ,  0.96939671]])
```

Figure 7.22: Screenshot for output of matrices W_o and U_o

```
o
```

```
array([[-0.06989015],
       [ 0.99999957],
       [ 0.11232103]])
```

Figure 7.23: Screenshot of the value of the output gate

$$h[t] = hadamard(o[t], tanh(C[t]))$$

Figure 7.24: Expression to calculate the value of the next activation

```
h_new
```

```
array([[-0.04695679],
       [ 0.12468345],
       [ 0.07479682]])
```

Figure 7.25: Screenshot for the current timestep activation

```
df.head()
```

	v1	v2	Unnamed: 2	Unnamed: 3	Unnamed: 4
0	ham	Go until jurong point, crazy.. Available only ...	NaN	NaN	NaN
1	ham	Ok lar... Joking wif u oni...	NaN	NaN	NaN
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	NaN	NaN	NaN
3	ham	U dun say so early hor... U c already then say...	NaN	NaN	NaN
4	ham	Nah I don't think he goes to usf, he lives aro...	NaN	NaN	NaN

Figure 7.26: Screenshot of the output for spam classification

```
df.head()
```

	v1	v2
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...

Figure 7.27: Screenshot for columns with text and labels

```
df["v1"].value_counts()
```

```
ham     4825
spam     747
Name: v1, dtype: int64
```

Figure 7.28: Screenshot for label distribution

```
X
```

```
array(['Go until jurong point, crazy.. Available only in bugi
s n great world la e buffet... Cine there got amore wat...',
       'Ok lar... Joking wif u oni...',
       "Free entry in 2 a wkly comp to win FA Cup final tkts
21st May 2005. Text FA to 87121 to receive entry question(std
txt rate)T&C's apply 08452810075over18's",
       ..., 'Pity, * was in mood for that. So...any other sug
gestions?',
       "The guy did some bitching but I acted like i'd be int
erested in buying something else next week and he gave it to
us for free",
       'Rofl. Its true to its name'], dtype=object)
```

Figure 7.29: Screenshot for output X

```
Y
```

```
array([0, 0, 1, ..., 0, 0, 0])
```

Figure 7.30: Screenshot for output Y

```
In [24]: text_tokenized
```

```
Out[24]: [[50, 64, 8, 89, 67, 58],
          [46, 6],
          [47, 8, 19, 4, 2, 71, 2, 2, 73],
          [6, 23, 6, 57],
          [1, 98, 69, 2, 69],
          [67, 21, 7, 38, 87, 55, 3, 44, 12, 14, 85, 46, 2, 68, 2],
          [11, 9, 25, 55, 2, 36, 10, 10, 55],
          [72, 13, 72, 13, 12, 51, 2, 13],
          [72, 4, 3, 17, 2, 2, 16, 64],
          [13, 96, 26, 6, 81, 2, 2, 5, 36, 12, 47, 16, 5, 96, 47, 18],
          [30, 32, 77, 7, 1, 98, 70, 2, 80, 40, 93, 88],
          [2, 48, 2, 73, 7, 68, 2, 65, 92, 42],
          [3, 17, 4, 47, 8, 91, 73, 5, 2, 38],
          [12, 5, 2, 3, 12, 40, 1, 1, 97, 13, 12, 7, 33, 11, 3, 17, 7,
         4, 29, 51],
          [1, 17, 4, 18, 36, 33],
          [2, 13, 5, 8, 5, 73, 26, 89],
          [93, 30],
          [6, 49, 19, 1, 69, 1],
          [34, 5, 6, 5, 61],
```

Figure 7.31: Screenshot for the output of tokenized values

```
sequences
```

```
array([[ 0,  0,  0, ..., 89, 67, 58],
       [ 0,  0,  0, ...,  0, 46,  6],
       [ 0,  0,  0, ...,  2,  2, 73],
       ...,
       [ 0,  0,  0, ..., 12, 20, 23],
       [ 0,  0,  0, ...,  2, 12, 47],
       [ 0,  0,  0, ..., 61,  2, 61]], dtype=int32)
```

Figure 7.32: Screenshot for padded sequences

```
model.fit(sequences,Y,batch_size=128,epochs=10,
          validation_split=0.2)
```

```
Train on 4457 samples, validate on 1115 samples
Epoch 1/10
4457/4457 [==============================] - 2s 539us/step -
loss: 0.4885 - acc: 0.8548 - val_loss: 0.3700 - val_acc: 0.87
00
Epoch 2/10
4457/4457 [==============================] - 2s 374us/step -
loss: 0.3425 - acc: 0.8652 - val_loss: 0.2649 - val_acc: 0.87
71
Epoch 3/10
4457/4457 [==============================] - 2s 381us/step -
loss: 0.2028 - acc: 0.9226 - val_loss: 0.1489 - val_acc: 0.95
34
Epoch 4/10
4457/4457 [==============================] - 2s 367us/step -
loss: 0.1348 - acc: 0.9547 - val_loss: 0.1271 - val_acc: 0.95
16
Epoch 5/10
4457/4457 [==============================] - 2s 404us/step -
loss: 0.1157 - acc: 0.9605 - val_loss: 0.1073 - val_acc: 0.95
78
Epoch 6/10
4457/4457 [==============================] - 2s 368us/step -
loss: 0.1061 - acc: 0.9632 - val_loss: 0.1027 - val_acc: 0.96
14
Epoch 7/10
4457/4457 [==============================] - 2s 371us/step -
loss: 0.0998 - acc: 0.9657 - val_loss: 0.1046 - val_acc: 0.95
78
Epoch 8/10
4457/4457 [==============================] - 2s 372us/step -
loss: 0.0955 - acc: 0.9672 - val_loss: 0.1004 - val_acc: 0.95
96
```

Figure 7.33: Screenshot of model fitting to 10 epochs

```
model.predict(test_sequences_matrix)
```

```
array([[0.96648586]], dtype=float32)
```

Figure 7.34: Screenshot of the output of model prediction

```
array([[0.979119]], dtype=float32)
```

Figure 7.35: Output for mail category prediction

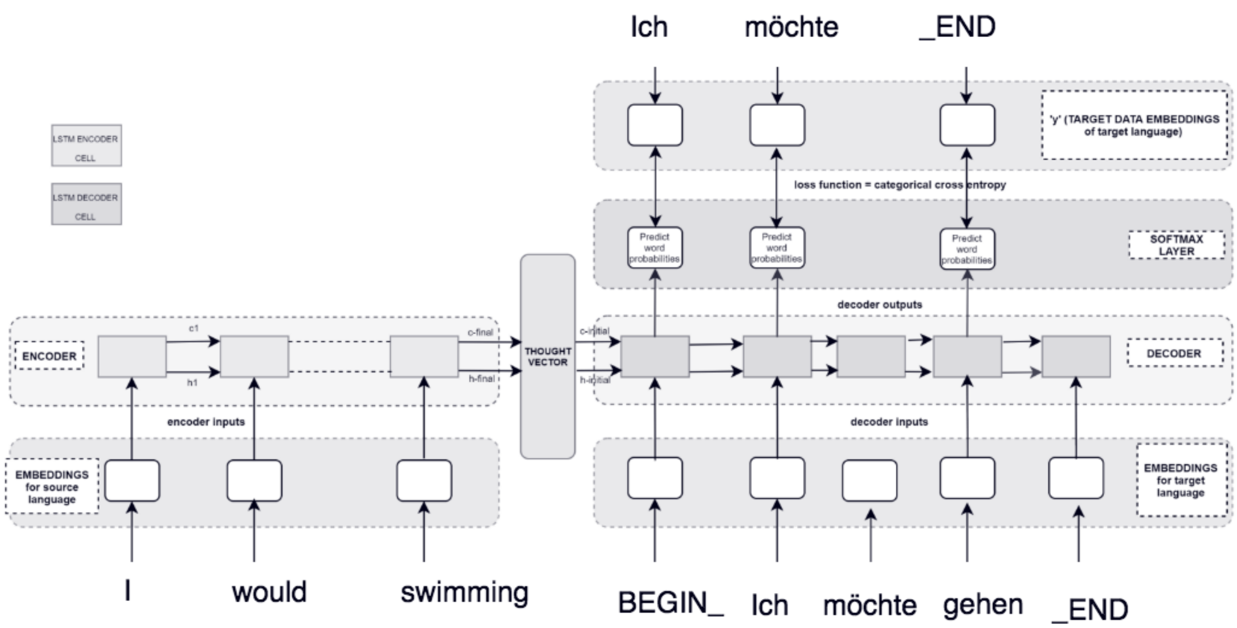

Figure 7.36: Neural translation model

```
lines_to_use
```

```
['Hi.\tHallo!',
 'Hi.\tGrüß Gott!',
 'Run!\tLauf!',
 'Wow!\tPotzdonner!',
 'Wow!\tDonnerwetter!',
 'Fire!\tFeuer!',
 'Help!\tHilfe!',
 'Help!\tZu Hülf!',
 'Stop!\tStopp!',
 'Wait!\tWarte!',
 'Go on.\tMach weiter.',
 'Hello!\tHallo!',
 'I ran.\tIch rannte.',
 'I see.\tIch verstehe.',
 'I see.\tAha.',
 'I try.\tIch probiere es.',
 'I won!\tIch hab gewonnen!',
```

Figure 7.37: Screenshot for the English-to-German translation of sentence pairs

```
input_texts
```

```
['Hi.',
 'Hi.',
 'Run!',
 'Wow!',
 'Wow!',
 'Fire!',
 'Help!',
 'Help!',
 'Stop!',
 'Wait!',
 'Go on.',
 'Hello!',
 'I ran.',
 'I see.',
 'I see.',
 'I try.',
 'I won!',
 'I won!',
 'Smile.',
 'Cheers!',
```

```
target_texts
```

```
['BEGIN_ Hallo! _END',
 'BEGIN_ Grüß Gott! _END',
 'BEGIN_ Lauf! _END',
 'BEGIN_ Potzdonner! _END',
 'BEGIN_ Donnerwetter! _END',
 'BEGIN_ Feuer! _END',
 'BEGIN_ Hilfe! _END',
 'BEGIN_ Zu Hülf! _END',
 'BEGIN_ Stopp! _END',
 'BEGIN_ Warte! _END',
 'BEGIN_ Mach weiter. _END',
 'BEGIN_ Hallo! _END'
```

Figure 7.38: Screenshot for input and output texts after mapping

```
input_words
```

```
['"Look,"',
 '"aah."',
 '$',
 '%',
 ',',
 '-',
 '.',
 '...',
 ':',
 '?',
 'A',
 'A.',
 'ATM?',
 'AWOL.',
 'Abandon',
 'About',
 'Act',
 'Add',
 'Admission',
 'After',
```

```
target_words
```

```
['"Schau!"',
 '$.',
 '%',
 "'ne",
 ',',
 '-',
 '.',
 ':',
 '?',
 'Abend',
 'Abend!',
 'Abend?',
 'Abendbrot',
```

Figure 7.39: Screenshot for input text and target words

```
input_token_index
```

```
{'"Look,"': 0,
 '"aah."': 1,
 '$': 2,
 '%': 3,
 ',': 4,
 '-': 5,
 '.': 6,
 '...': 7,
 ':': 8,
 '?': 9,
 'A': 10,
 'A.': 11,
 'ATM?': 12,
 'AWOL.': 13,
 'Abandon': 14,
 'About': 15,
 'Act': 16,
 'Add': 17,
 'Admission': 18,
 'After': 19,
```

```
target_token_index
```

```
{'"Schau!"': 0,
 '$.': 1,
 '%': 2,
 "'ne": 3,
 ',': 4,
 '-': 5,
 '.': 6,
 ':': 7,
 '?': 8,
 'Abend': 9,
 'Abend!': 10,
 'Abend?': 11,
 'Abendbrot': 12,
```

Figure 7.40: Screenshot for output of integer index for each token

```
encoder_input_data
```

```
array([[ 283.,    0.,    0., ...,    0.,    0.,    0.],
       [ 283.,    0.,    0., ...,    0.,    0.,    0.],
       [ 505.,    0.,    0., ...,    0.,    0.,    0.],
       ...,
       [ 696., 3001., 4502., ...,    0.,    0.,    0.],
       [ 696., 3001., 4682., ...,    0.,    0.,    0.],
       [ 696., 3004., 3008., ...,    0.,    0.,    0.]], dtyp
e=float32)
```

```
decoder_input_data
```

```
array([[ 175., 1172., 3665., ...,    0.,    0.,    0.],
       [ 175., 1140., 1113., ...,    0.,    0.,    0.],
       [ 175., 1706., 3665., ...,    0.,    0.,    0.],
       ...,
       [ 175., 3405., 8432., ...,    0.,    0.,    0.],
       [ 175., 3405., 6239., ...,    0.,    0.,    0.],
       [ 175., 3405., 6239., ...,    0.,    0.,    0.]], dtyp
e=float32)
```

```
decoder_target_data
```

```
array([[[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]],

       [[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
```

Figure 7.41: Screenshot of matrix population

```
_____________________________________
Layer (type)                    Output Shape         Param #
Connected to
==============================================================
=====================================
input_1 (InputLayer)            (None, None)         0
______________________________________________________________
_____________________________________
input_2 (InputLayer)            (None, None)         0
______________________________________________________________
_____________________________________
embedding_1 (Embedding)         (None, None, 50)     286200
input_1[0][0]
______________________________________________________________
_____________________________________
embedding_2 (Embedding)         (None, None, 50)     456300
input_2[0][0]
______________________________________________________________
_____________________________________
lstm_1 (LSTM)                   [(None, 50), (None,  20200
embedding_1[0][0]
______________________________________________________________
_____________________________________
lstm_2 (LSTM)                   [(None, None, 50), ( 20200
embedding_2[0][0]

lstm_1[0][1]

lstm_1[0][2]
______________________________________________________________
_____________________________________
dense_1 (Dense)                 (None, None, 9126)   465426
lstm_2[0][0]
==============================================================
=====================================
Total params: 1,248,326
Trainable params: 1,248,326
Non-trainable params: 0
```

Figure 7.42: Screenshot of model summary

```
Train on 19000 samples, validate on 1000 samples
Epoch 1/20
19000/19000 [==============================] - 310s 16ms/step
- loss: 1.6492 - acc: 0.0787 - val_loss: 1.8068 - val_acc: 0.
0674
Epoch 2/20
19000/19000 [==============================] - 303s 16ms/step
- loss: 1.5174 - acc: 0.0908 - val_loss: 1.6923 - val_acc: 0.
0822
Epoch 3/20
19000/19000 [==============================] - 304s 16ms/step
- loss: 1.4060 - acc: 0.1040 - val_loss: 1.6107 - val_acc: 0.
1065
Epoch 4/20
19000/19000 [==============================] - 292s 15ms/step
- loss: 1.3343 - acc: 0.1157 - val_loss: 1.5683 - val_acc: 0.
1100
Epoch 5/20
19000/19000 [==============================] - 292s 15ms/step
- loss: 1.2860 - acc: 0.1212 - val_loss: 1.5299 - val_acc: 0.
1197
Epoch 6/20
19000/19000 [==============================] - 291s 15ms/step
- loss: 1.2510 - acc: 0.1241 - val_loss: 1.5037 - val_acc: 0.
1145
Epoch 7/20
19000/19000 [==============================] - 291s 15ms/step
```

Figure 7.43: Screenshot of model fitting with 20 epochs

```
reverse_input_word_index
```

```
 3: '%',
 4: ',',
 5: '-',
 6: '.',
 7: '...',
 8: ':',
 9: '?',
 10: 'A',
 11: 'A.',
 12: 'ATM?',
 13: 'AWOL.',
 14: 'Abandon',
 15: 'About',
 16: 'Act',
 17: 'Add',
 18: 'Admission',
 19: 'After',
 20: 'Aim.',
 21: "Ain't",
 22: 'Air',
```

```
reverse_target_word_index
```

```
{0: '"Schau!"',
 1: '$.',
 2: '%',
 3: "'ne",
 4: ',',
 5: '-',
 6: '.',
 7: ':',
 8: '?',
 9: 'Abend',
 10: 'Abend!',
 11: 'Abend?',
 12: 'Abendbrot',
```

Figure 7.44: Screenshot of dictionary values

```
In [122]: text_to_translate = "Where is my car?"
```

```
In [123]: encoder_input_to_translate = np.zeros(
              (1, max_input_seq_length),
              dtype='float32')

          for t, word in enumerate(text_to_translate.split()):
              encoder_input_to_translate[0, t] = input_token_index[word]
```

```
In [124]: decode_sequence(encoder_input_to_translate)

Out[124]: ' Wo ist mein Auto? _END'
```

Figure 7.45: Screenshot of English-to-German translator

```
' Get a lot. _END'
```

Figure 7.46: Output for French to English translator model

Lesson 08: State-of-the-Art Natural Language Processing

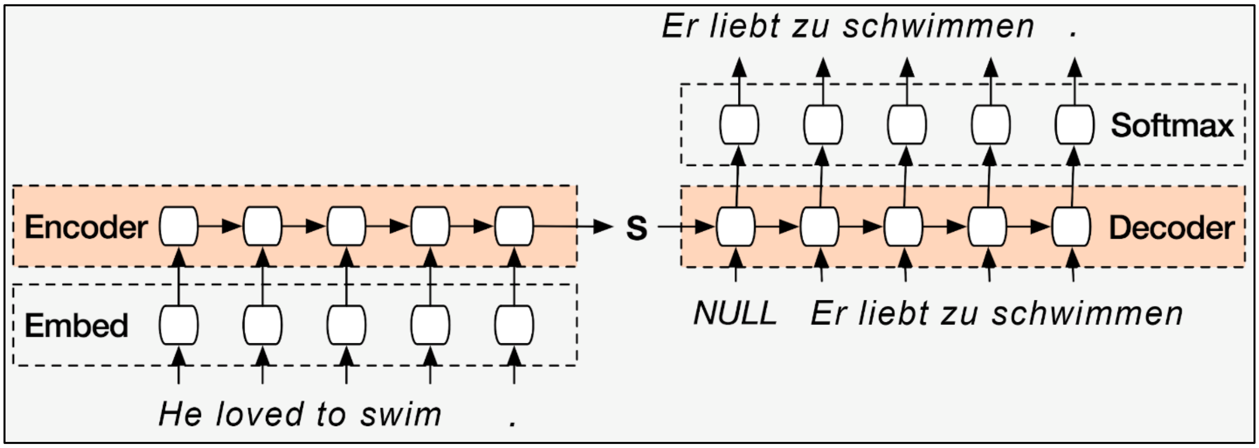

Figure 8.1: Neural language translation model

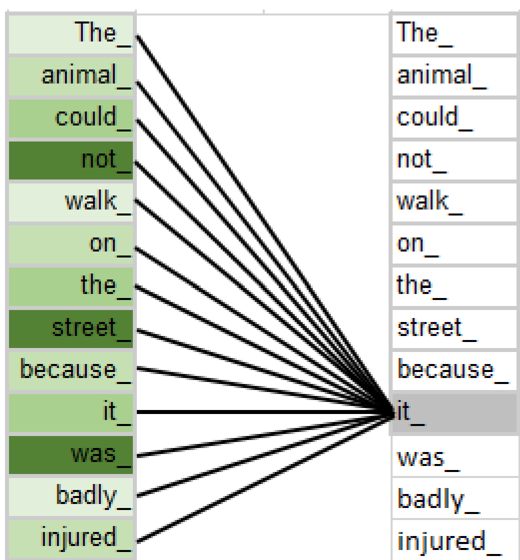

Figure 8.2: An example of an attention mechanism

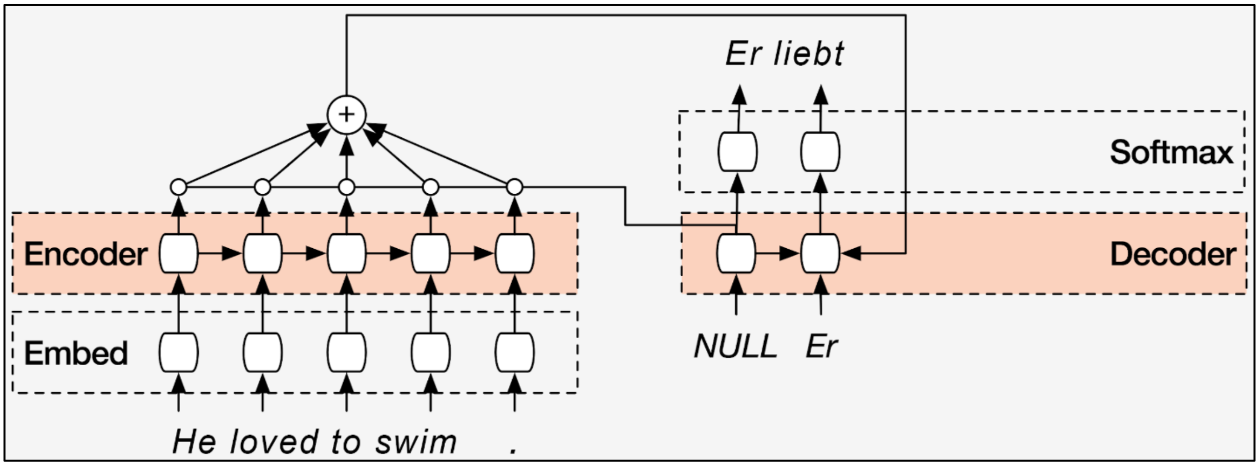

Figure 8.3: An attention mechanism model

User Input	Normalized Date
3-May-79	5/3/1979
5-Apr-09	5/5/2009
21th of August 2016	8/21/2016
Tue 10 Jul 2007	7/10/2007

Figure 8.4: Table for date normalization

```
context[t] = dot(H, alpha[t])
```

Figure 8.5: Expression for the context vector

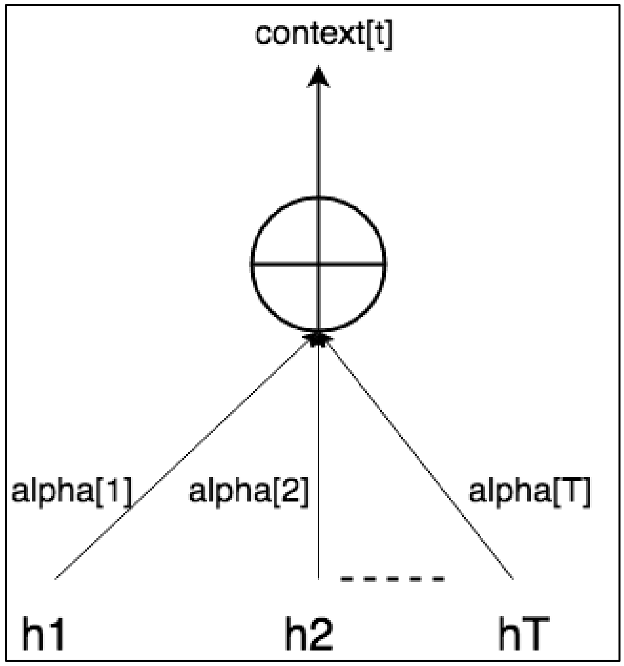

Figure 8.6: Determination of attention to inputs

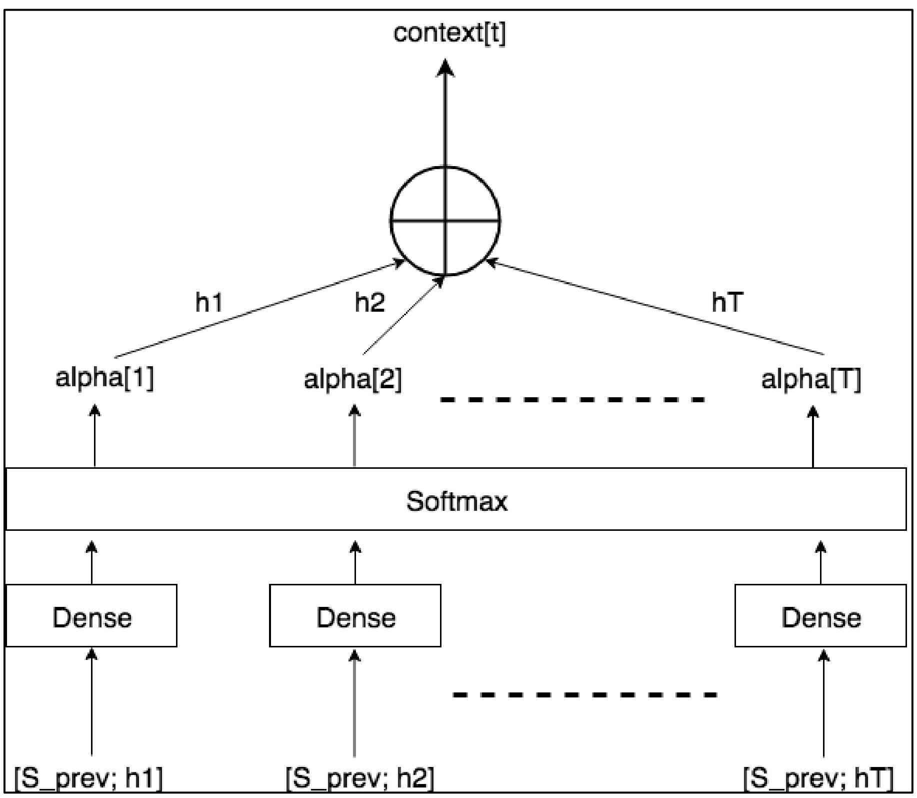

Figure 8.7: The calculation of alpha

```
m = 10000
dataset, human_vocab, machine_vocab, inv_machine_vocab = load_dataset(m)
```

```
100%|██████████| 10000/10000 [00:00<00:00, 23983.69it/s]
```

```
dataset
```

```
[('9 may 1998', '1998-05-09'),
 ('10.09.70', '1970-09-10'),
 ('4/28/90', '1990-04-28'),
 ('thursday january 26 1995', '1995-01-26'),
 ('monday march 7 1983', '1983-03-07'),
```

Figure 8.8: Screenshot displaying variable values

```
human_vocab
```

```
 '9': 12,
 'a': 13,
 'b': 14,
 'c': 15,
 'd': 16,
 'e': 17,
 'f': 18,
```

Figure 8.9: Screenshot for human_vocab dictionary

```
machine_vocab
```

```
{'-': 0,
 '0': 1,
 '1': 2,
 '2': 3,
 '3': 4,
 '4': 5,
 '5': 6,
 '6': 7,
 '7': 8,
 '8': 9,
 '9': 10}
```

Figure 8.10: Screenshot for the machine_vocab dictionary

```
inv_machine_vocab
```

```
{0: '-',
 1: '0',
 2: '1',
 3: '2',
 4: '3',
 5: '4',
 6: '5',
 7: '6',
 8: '7',
 9: '8',
 10: '9'}
```

Figure 8.11: Screenshot for the inv_machine_vocab dictionary

```
X.shape: (10000, 30)
Y.shape: (10000, 10)
Xoh.shape: (10000, 30, 37)
Yoh.shape: (10000, 10, 11)
```

Figure 8.12: Screenshot for the shape of matrices

```
index = 0
print("Source date:", dataset[index][0])
print("Target date:", dataset[index][1])
print()
print("Source after preprocessing (indices):", X[index].shape)
print("Target after preprocessing (indices):", Y[index].shape)
print()
print("Source after preprocessing (one-hot):", Xoh[index].shape)
print("Target after preprocessing (one-hot):", Yoh[index].shape)
```

```
Source date: 9 may 1998
Target date: 1998-05-09

Source after preprocessing (indices): (30,)
Target after preprocessing (indices): (10,)

Source after preprocessing (one-hot): (30, 37)
Target after preprocessing (one-hot): (10, 11)
```

Figure 8.13: Screenshot for the shape of matrices after processing

```
model.summary()
```

```
______________________________________________________________________________________________
______
dense_3 (Dense)                 (None, 11)           715         lstm_1[0][0]
                                                                 lstm_1[1][0]
                                                                 lstm_1[2][0]
                                                                 lstm_1[3][0]
                                                                 lstm_1[4][0]
                                                                 lstm_1[5][0]
                                                                 lstm_1[6][0]
                                                                 lstm_1[7][0]
                                                                 lstm_1[8][0]
                                                                 lstm_1[9][0]
==============================================================================================
======
Total params: 52,960
Trainable params: 52,960
Non-trainable params: 0
______________________________________________________________________________________________
```

Figure 8.14: Screenshot for model summary

```
Epoch 1/1
10000/10000 [==============================] - 15s 1ms/step - loss: 17.0066 - dense_3_loss:
2.5402 - dense_3_acc: 0.4576 - dense_3_acc_1: 0.7088 - dense_3_acc_2: 0.3134 - dense_3_acc_3:
0.0748 - dense_3_acc_4: 0.8606 - dense_3_acc_5: 0.3337 - dense_3_acc_6: 0.0510 - dense_3_acc_
7: 0.8976 - dense_3_acc_8: 0.2671 - dense_3_acc_9: 0.1082
```

Figure 8.15: Screenshot for epoch training

```
source: 3 May 1979
output: 1979-05-03
source: 5 April 09
output: 2009-05-05
source: 21th of August 2016
output: 2016-08-21
source: Tue 10 Jul 2007
output: 2007-07-10
source: Saturday May 9 2018
output: 2018-05-09
source: March 3 2001
output: 2001-03-03
source: March 3rd 2001
output: 2001-03-03
source: 1 March 2001
output: 2001-03-01
```

Figure 8.16: Screenshot for normalized date output

```
source: Last night a meteorite was seen flying near the earth's moon.
output: aaaaa            <pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
```

Figure 8.17: Output for text summarization

Lesson 09: A Practical NLP Project Workflow in an Organization

Figure 9.1: General workflow for the development of a machine learning product

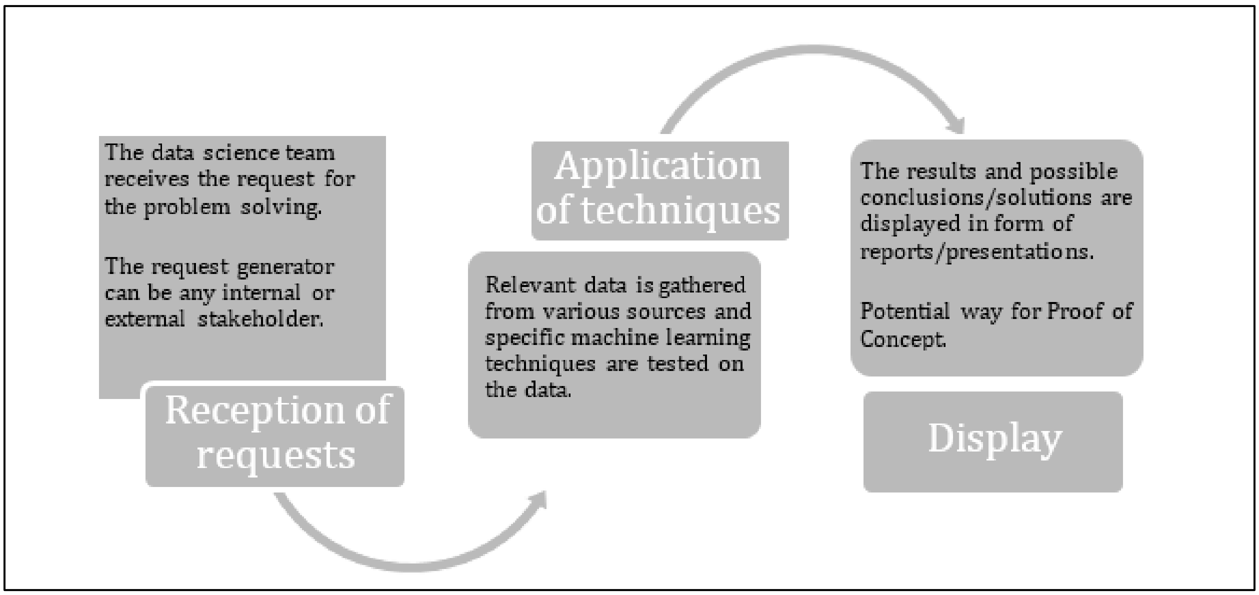

Figure 9.2: General presentation workflow

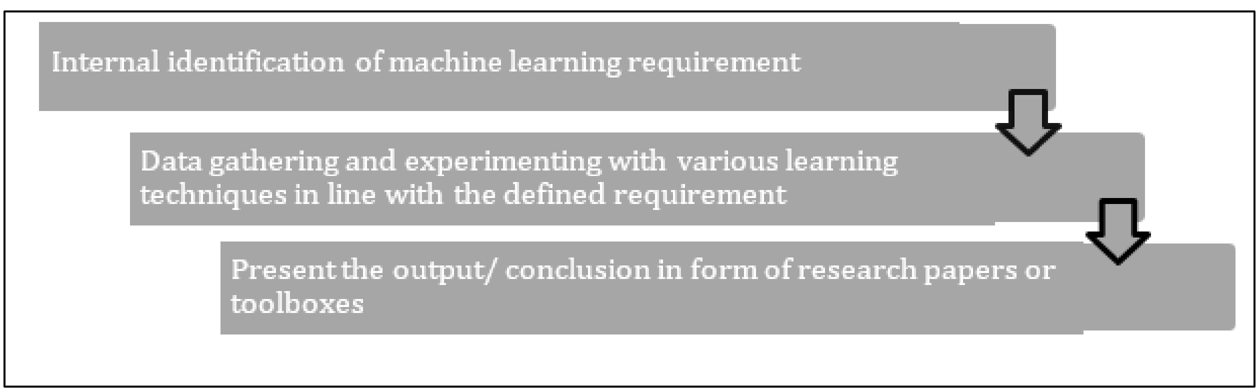

Figure 9.3: Research workflow

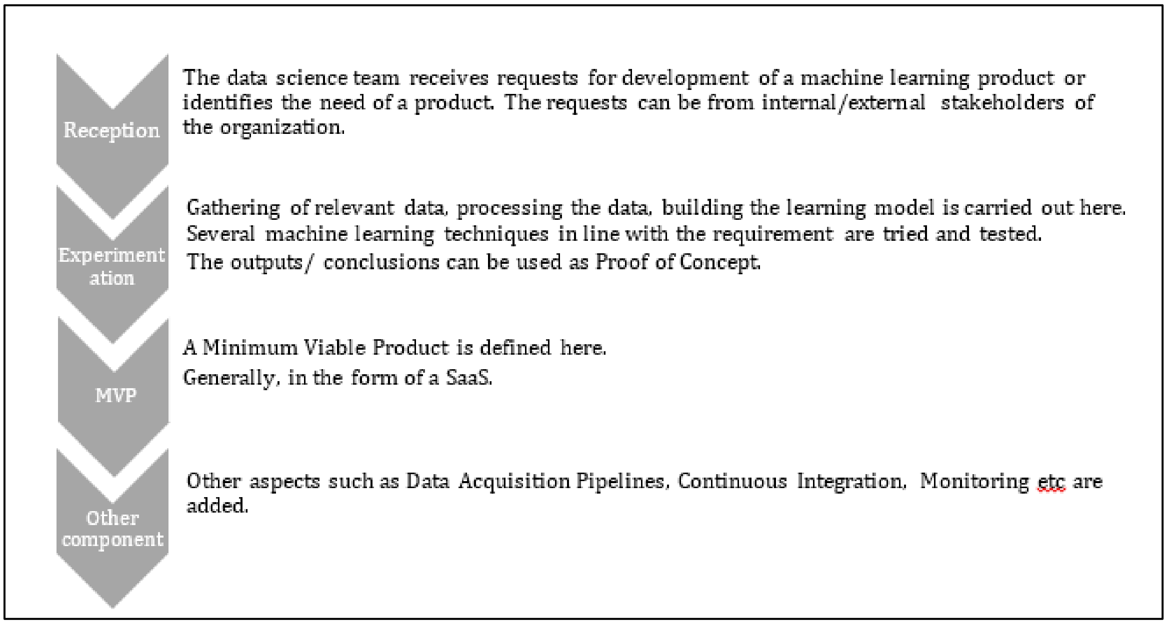

Figure 9.4: Production-oriented workflow

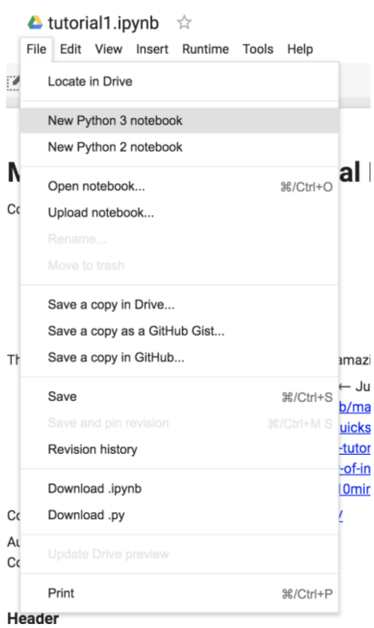

Figure 9.5: A new Python notebook on Google Colab

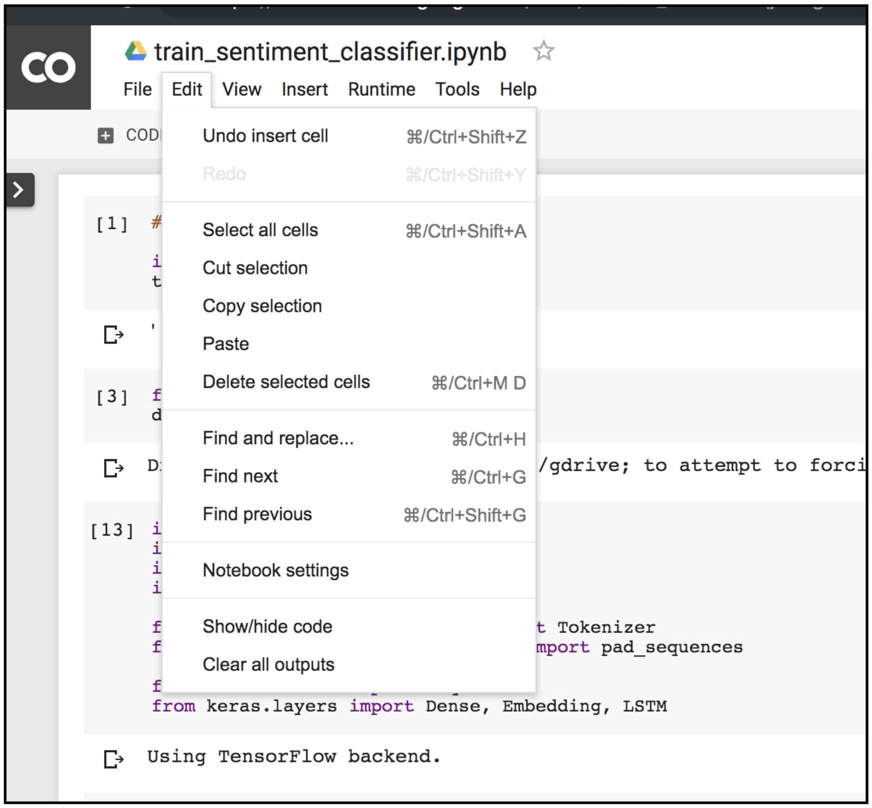

Figure 9.6: Edit dropdown in Google Colab

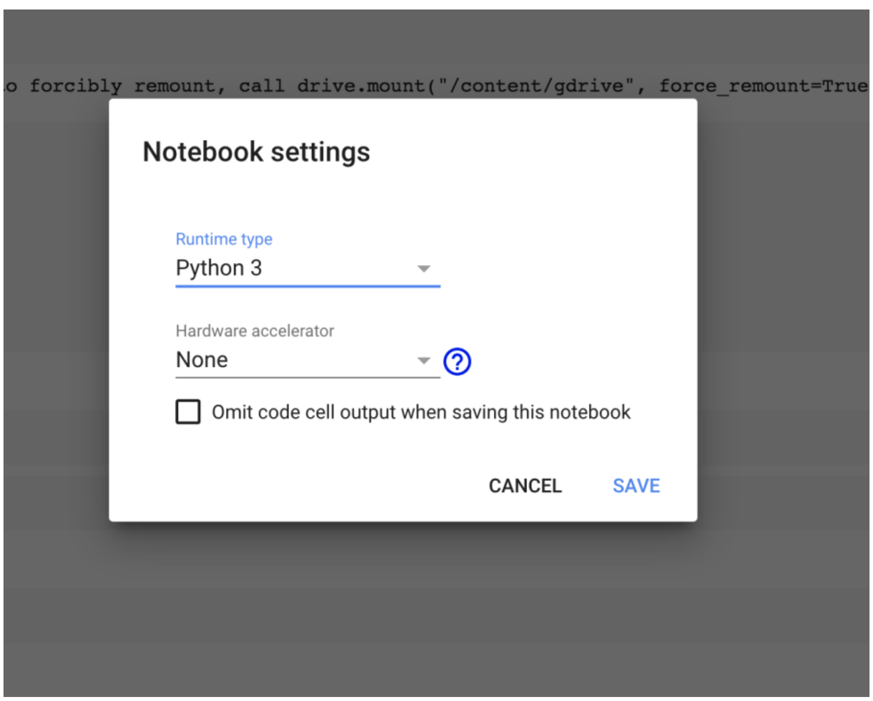

Figure 9.7: Notebook settings for Google Colab

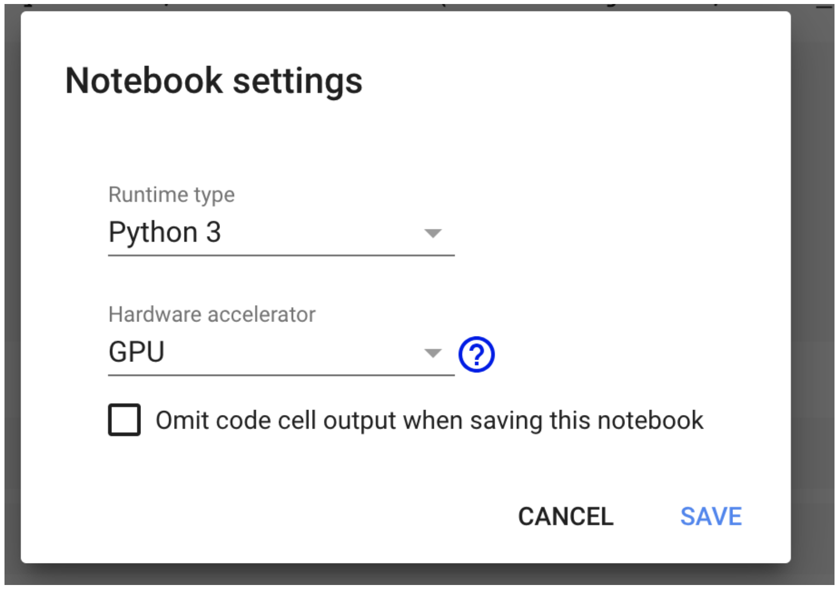

Figure 9.8: GPU hardware accelerator

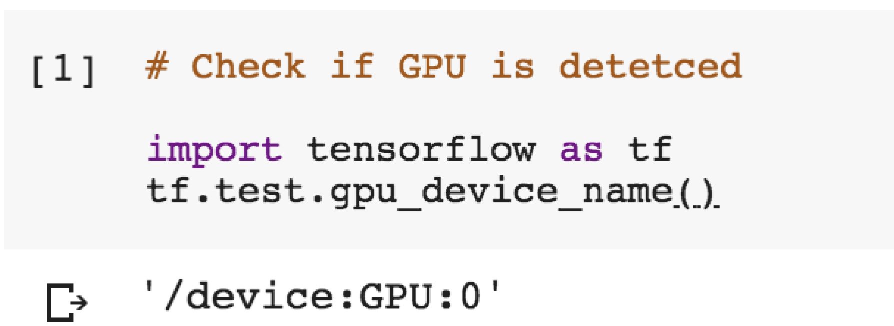

Figure 9.9: Screenshot for GPU device name

Figure 9.10: Screenshot for importing data from Google Drive

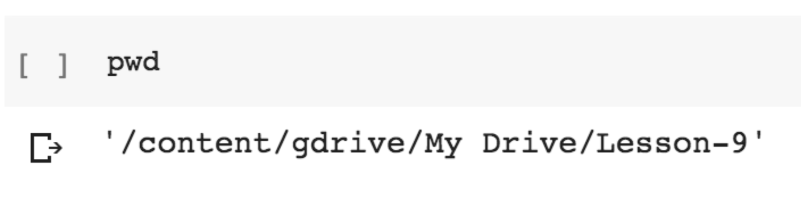

Figure 9.11: Data imported on the Colab notebook from Google Drive

```
[ ]  !unzip data.csv.zip

     Archive:  data.csv.zip
       inflating: data.csv
        creating: __MACOSX/
       inflating: __MACOSX/._data.csv
```

Figure 9.12: Unzipping a data file on a Colab notebook

```
df.head()
```

	rating	title	review
0	3	more like funchuck	gave this to my dad for a gag gift after direc...
1	5	Inspiring	i hope a lot of people hear this cd we need mo...
2	5	The best soundtrack ever to anything.	im reading a lot of reviews saying that this i...
3	4	Chrono Cross OST	the music of yasunori misuda is without questi...
4	5	Too good to be true	probably the greatest soundtrack in history us...

Figure 9.13: Screenshot of dataframe contents

```
X

array([[   0,    0,    0, ...,   40,    7,    6],
       [   0,    0,    0, ...,   23, 1694,    2],
       [   0,    0,    0, ...,   24,  171,  170],
       ...,
       [   0,    0,    0, ...,   42,  712, 1358],
       [   0,    0,    0, ...,  580,  290, 1722],
       [   0,    0,    0, ...,    1,   38, 1840]], dtype=int32)
```

Figure 9.14: Screenshot of the X variable array

```
y_train

array([[0, 0, 1, 0, 0],
       [0, 0, 0, 0, 1],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [1, 0, 0, 0, 0]], dtype=uint8)
```

Figure 9.15: y_train output

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 250, 128)          256000
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               91600
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 505
=================================================================
Total params: 348,105
Trainable params: 348,105
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 9.16: Screenshot of the model summary

```
# fit the model
model.fit(X[:100000, :], y_train[:100000, :], batch_size = 128, epochs=15, validation_split=0.2)

Train on 80000 samples, validate on 20000 samples
Epoch 1/15
80000/80000 [==============================] - 320s 4ms/step - loss: 1.1106 - acc: 0.5231 - val_loss: 1.1261 - val_acc: 0.5171
Epoch 2/15
80000/80000 [==============================] - 319s 4ms/step - loss: 1.0786 - acc: 0.5385 - val_loss: 1.1099 - val_acc: 0.5192
Epoch 3/15
80000/80000 [==============================] - 318s 4ms/step - loss: 1.0482 - acc: 0.5533 - val_loss: 1.1256 - val_acc: 0.5164
Epoch 4/15
80000/80000 [==============================] - 311s 4ms/step - loss: 1.0226 - acc: 0.5660 - val_loss: 1.1226 - val_acc: 0.5172
Epoch 5/15
80000/80000 [==============================] - 315s 4ms/step - loss: 1.0014 - acc: 0.5771 - val_loss: 1.1348 - val_acc: 0.5087
Epoch 6/15
80000/80000 [==============================] - 319s 4ms/step - loss: 0.9754 - acc: 0.5873 - val_loss: 1.1455 - val_acc: 0.5078
Epoch 7/15
80000/80000 [==============================] - 320s 4ms/step - loss: 0.9496 - acc: 0.6015 - val_loss: 1.1708 - val_acc: 0.5051
Epoch 8/15
80000/80000 [==============================] - 322s 4ms/step - loss: 0.9244 - acc: 0.6099 - val_loss: 1.1870 - val_acc: 0.5028
Epoch 9/15
80000/80000 [==============================] - 317s 4ms/step - loss: 0.8978 - acc: 0.6226 - val_loss: 1.2118 - val_acc: 0.5002
Epoch 10/15
80000/80000 [==============================] - 313s 4ms/step - loss: 0.8678 - acc: 0.6383 - val_loss: 1.2304 - val_acc: 0.4975
Epoch 11/15
80000/80000 [==============================] - 319s 4ms/step - loss: 0.8391 - acc: 0.6508 - val_loss: 1.2817 - val_acc: 0.4953
Epoch 12/15
80000/80000 [==============================] - 320s 4ms/step - loss: 0.8089 - acc: 0.6655 - val_loss: 1.3062 - val_acc: 0.4907
Epoch 13/15
80000/80000 [==============================] - 319s 4ms/step - loss: 0.7753 - acc: 0.6810 - val_loss: 1.3529 - val_acc: 0.4883
Epoch 14/15
80000/80000 [==============================] - 315s 4ms/step - loss: 0.7442 - acc: 0.6958 - val_loss: 1.3931 - val_acc: 0.4814
Epoch 15/15
80000/80000 [==============================] - 316s 4ms/step - loss: 0.7081 - acc: 0.7134 - val_loss: 1.4570 - val_acc: 0.4803
<keras.callbacks.History at 0x7fcba53a00f0>
```

Figure 9.17: Screenshot of the training session

```
Using TensorFlow backend.
2019-03-24 23:08:25.948604: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions
that this TensorFlow binary was not compiled to use: AVX2 FMA
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
Using TensorFlow backend.
2019-03-24 23:08:31.730337: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions
that this TensorFlow binary was not compiled to use: AVX2 FMA
 * Debugger is active!
 * Debugger PIN: 150-665-765
```

Figure 9.18: Output for Flask

```
Sending build context to Docker daemon  115.6MB
Step 1/9 : FROM python:3.6-slim
 ---> 5d4dd7f71a65
Step 2/9 : COPY ./app.py /deploy/
 ---> f71341666654
Step 3/9 : COPY ./requirements.txt /deploy/
 ---> 688538f2682c
Step 4/9 : COPY ./trained_model.h5 /deploy/
 ---> 89af21aa696e
Step 5/9 : COPY ./trained_tokenizer.pkl /deploy/
 ---> 9cba42121f49
Step 6/9 : WORKDIR /deploy/
 ---> Running in 204358b07798
Removing intermediate container 204358b07798
 ---> 33241b6c6015
Step 7/9 : RUN pip install -r requirements.txt
 ---> Running in d19156053f1d
Collecting Flask==1.0.2 (from -r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dacb4696386634607af824ea99
7202cd0edb4b/Flask-1.0.2-py2.py3-none-any.whl (91kB)
Collecting numpy==1.14.1 (from -r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/de/7d/348c5d8d44443656e76285aa97b828b6dbd9c10e5b9c0f7f
98eff0ff70e4/numpy-1.14.1-cp36-cp36m-manylinux1_x86_64.whl (12.2MB)
Collecting keras==2.2.4 (from -r requirements.txt (line 3))
  Downloading https://files.pythonhosted.org/packages/5e/10/aa32dad071ce52b5502266b5c659451cfd6ffcbf14e6c8c4
f16c0ff5aaab/Keras-2.2.4-py2.py3-none-any.whl (312kB)
Collecting tensorflow==1.10.0 (from -r requirements.txt (line 4))
  Downloading https://files.pythonhosted.org/packages/ee/e6/a6d371306c23c2b01cd2cb38909673d17ddd388d9e4b3c0f
6602bfd972c8/tensorflow-1.10.0-cp36-cp36m-manylinux1_x86_64.whl (58.4MB)
```

Figure 9.19: Output screenshot for docker build

```
(py36) ./deployment  docker run -p 80:80 app-packt
2019-04-28 21:57:24.697584: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructio
s that this TensorFlow binary was not compiled to use: AVX2 FMA
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
Using TensorFlow backend.
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Figure 9.20: Output screenshot for the docker run command

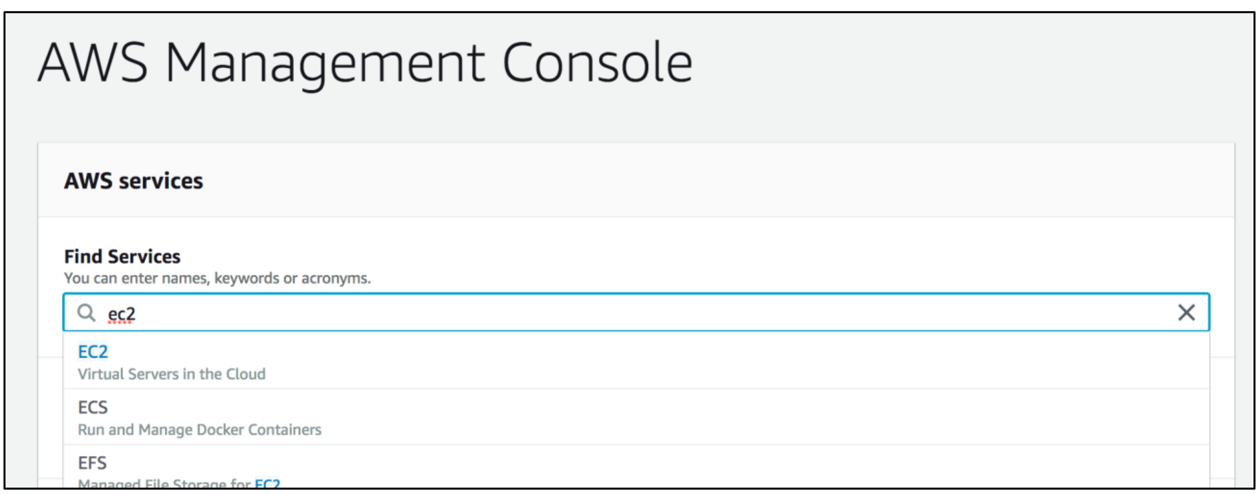

Figure 9.21: AWS services in the AWS Management Console

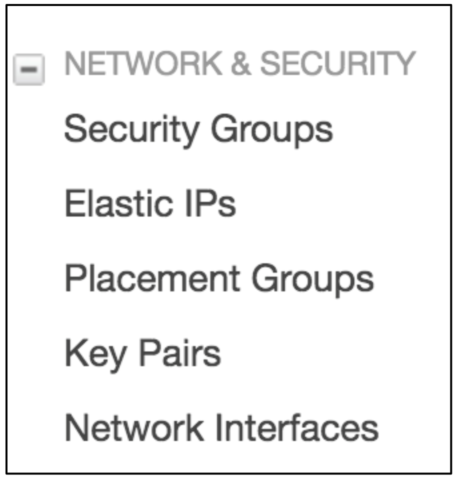

Figure 9.22: Network and security on the AWS console

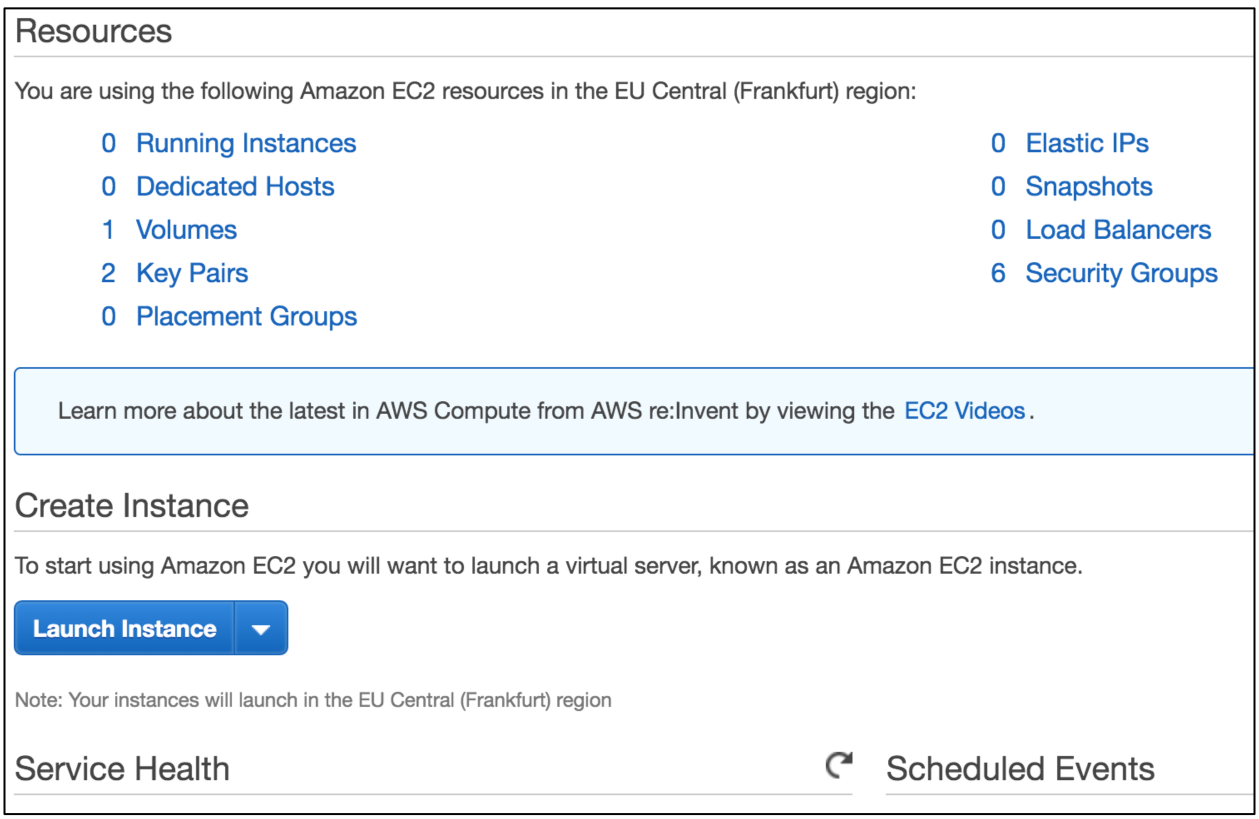

Figure 9.23: Resources on the AWS console

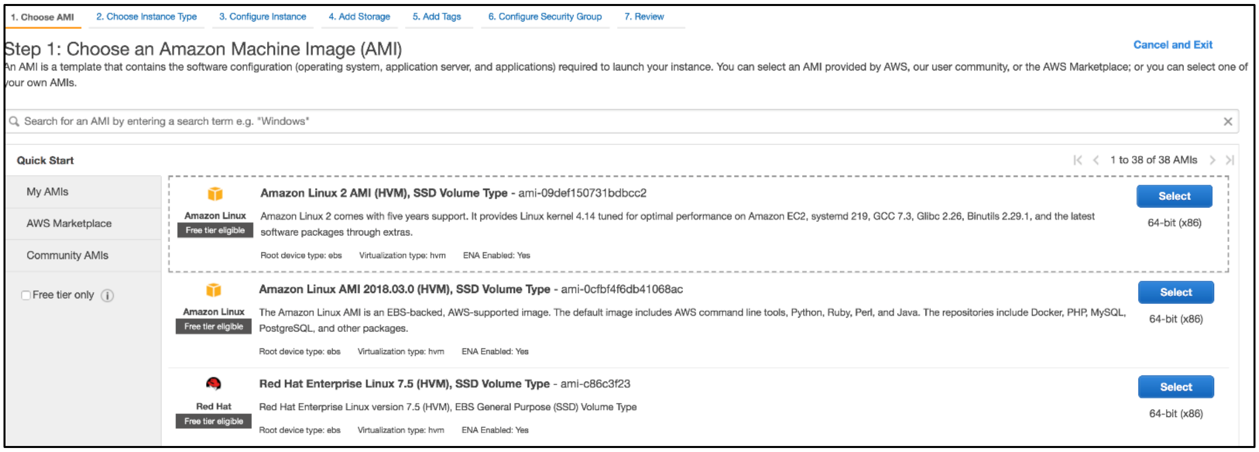

Figure 9.24: Amazon Machine Instance (AMI)

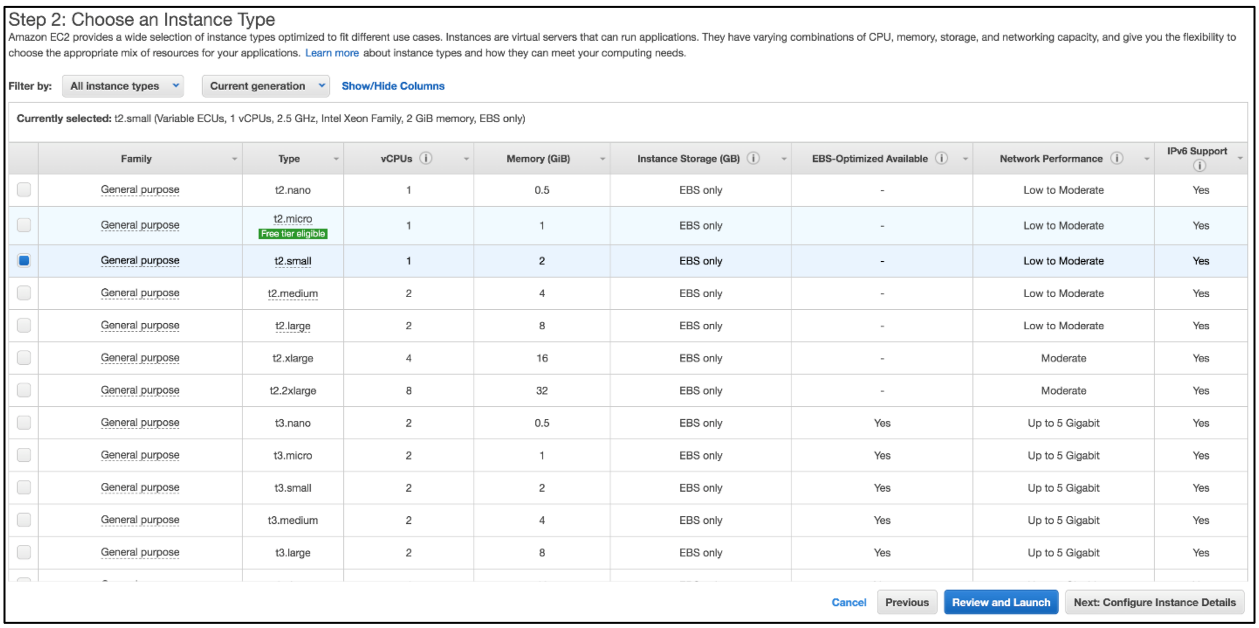

Figure 9.25: Choosing the instance type on AMI

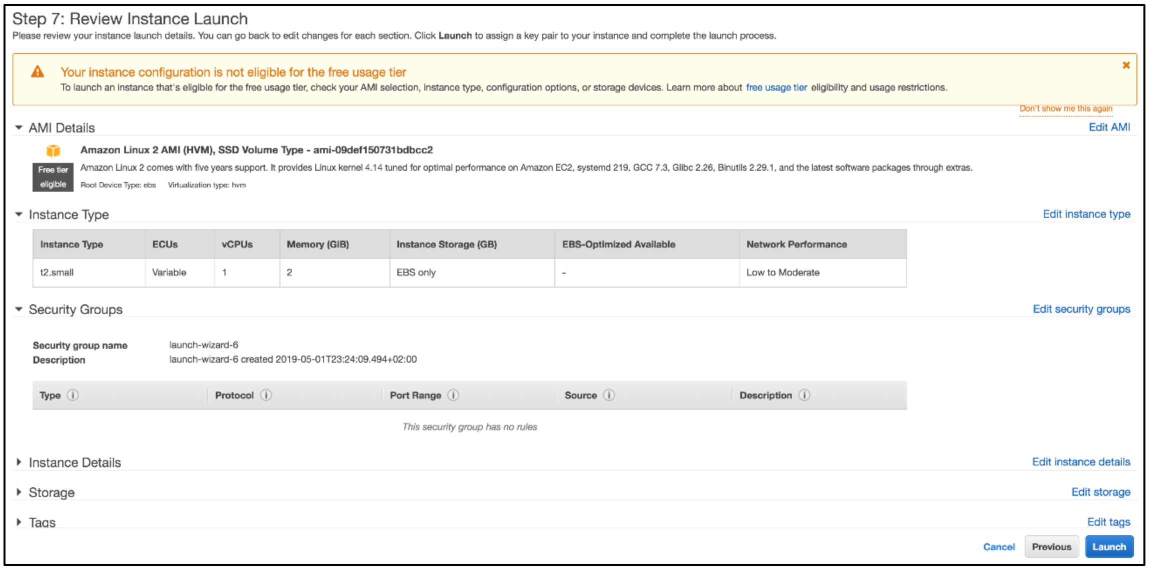

Figure 9.26: The review instance launch screen

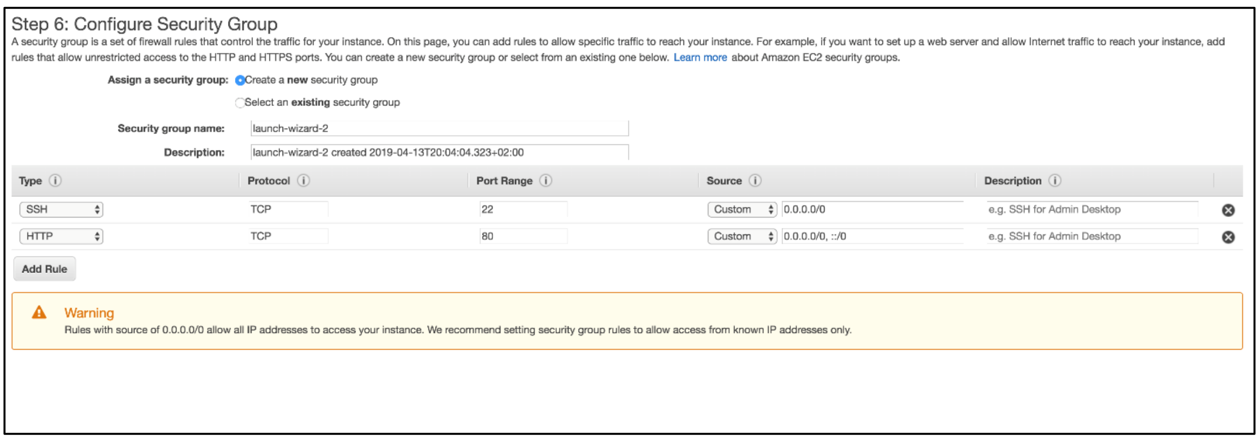

Figure 9.27: Configure the security group

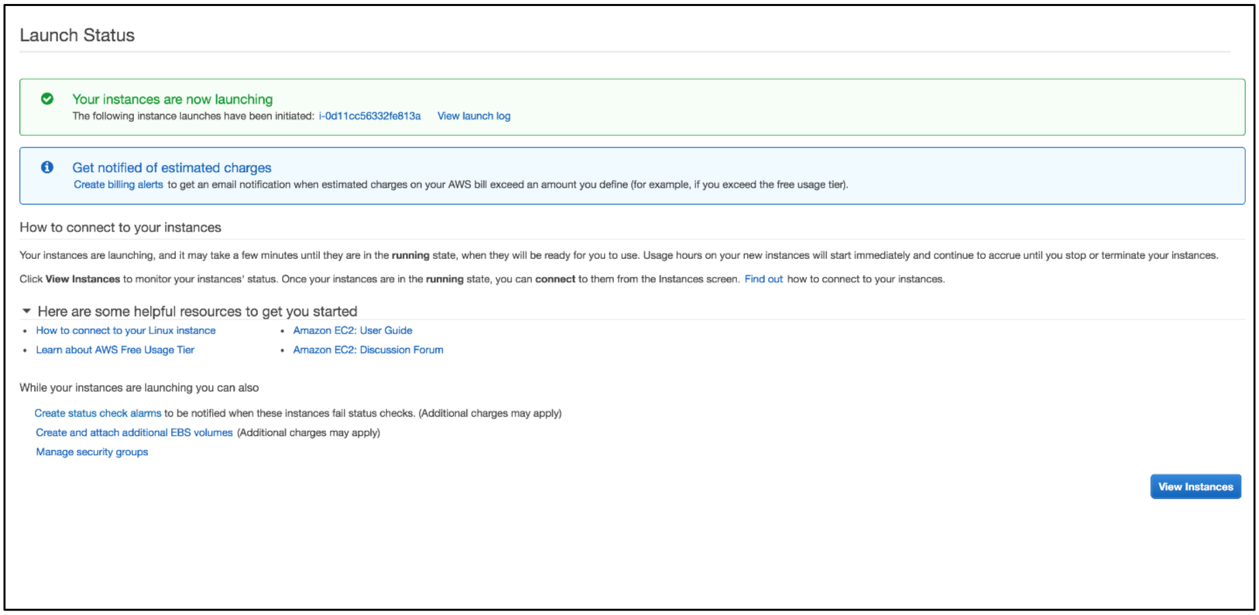

Figure 9.28: Launch status on the AWS instance

Figure 9.29: Screenshot for the home endpoint

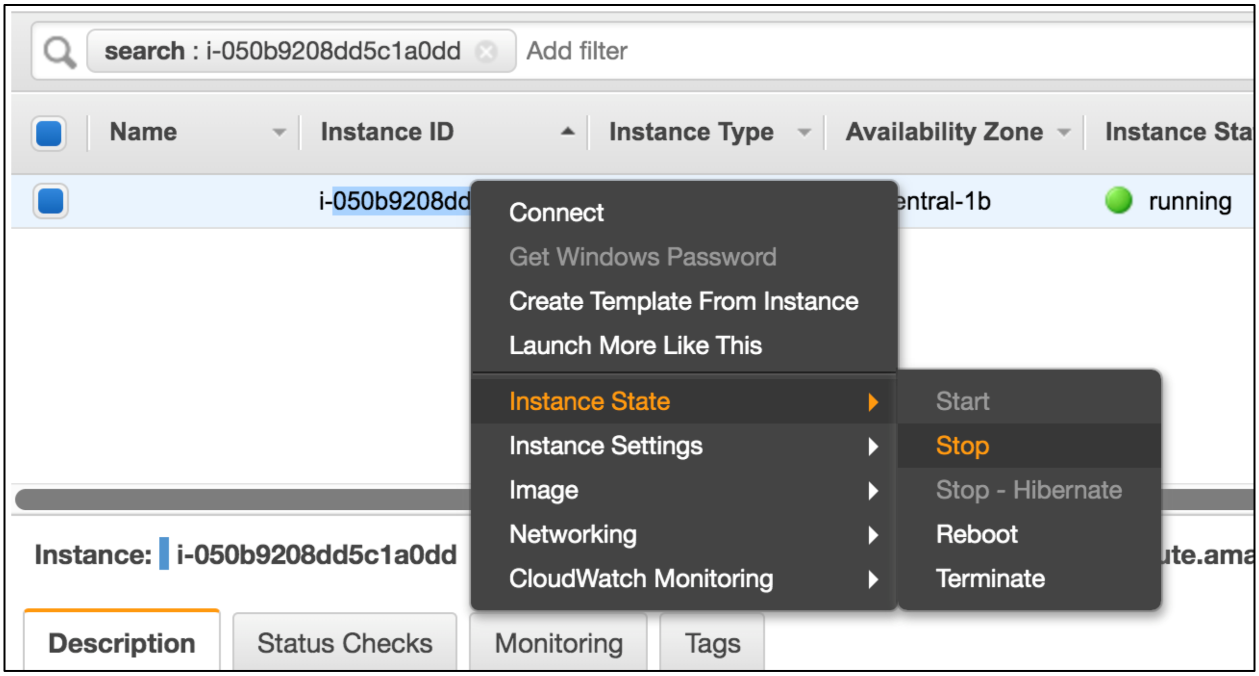

Figure 9.30: Stopping the AWS EC2 instance