CNN and RNN

Machine Learning

Introduction to Deep Learning

- Supervised Learning with Deep Neural Networks

Input(x)	Output(y)	Application	Model
Home features	Price	Real Estate	Fully- Connected Neural Nets
Ad, user info	Click on ad?(0/1)	Online Advertising	Fully- Connected Neural Nets
Image	Object (1,…,1000)	Photo tagging	Convolutional Neural Nets
Audio	Text transcript	Speech recognition	Recurrent Neural Nets
Korean	English	Machine translation	Recurrent Neural Nets

https://cs230.stanford.edu/

Introduction to Deep Learning

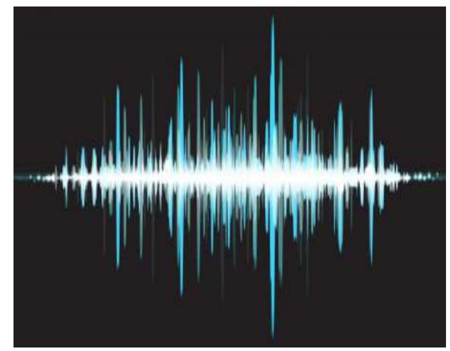
- Supervised Learning with Deep Neural Networks

Structured Data

Size	#bedrooms	…	Price(1000$)
2104	3	…	400
1600	3	…	330
2400	3	…	369
…	…	…	…
3000	4	…	540

Unstructured Data

Image



Audio

This shirt is very flattering to all due to …

Text

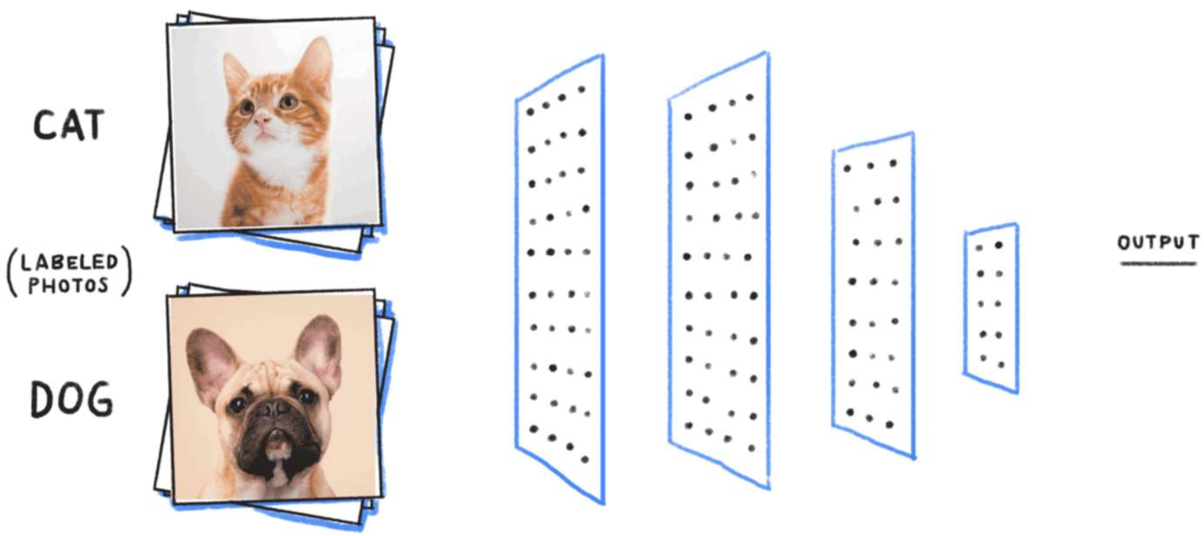
Video

Contents

- Convolutional Neural Networks
- Recurrent Neural Networks
- Google Co-lab link to given code
  - https://colab.research.google.com/drive/1Vc1gP1w1uxdD7GjUmbLbqRYz1jPPX5ap?usp=sharing

Convolutional Neural Networks

- Convolutional Neural Networks : for Analyzing Visual Data
  - Smaller number of connection
  - Weight sharing
  - Detect features at different positions in a image



https://becominghuman.ai/building-an-image-classifier-using-deep-learning-in-python-totally-from-a-beginners-perspective-be8dbaf22dd8

Convolutional Neural Networks

- Image Classification with CNNs
  - Load Fashion-MNIST Dataset and Preprocessing

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train),(x_test, y_test) = fashion_mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0
num_classes = 10

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_train.shape
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_test.shape
y_train = y_train.reshape(y_train.shape[0], 1)
y_train.shape
y_test = y_test.reshape(y_test.shape[0], 1)
y_test.shape

y_train
```

```
(60000, 28, 28, 1)
(10000, 28, 28, 1)
(60000, 1)
(10000, 1)
```

```
array([[9],
       [0],
       [0],
       ...,
       [3],
       [0],
       [5]], dtype=uint8)
```

Convolutional Neural Networks

- Image Classification with CNNs
  - Build the CNN Model

```
base_model = tf.keras.models.Sequential([
     layers.Conv2D(32, kernel_size=(3, 3), padding = "same",
                            input_shape = (28, 28, 1),
                            activation="relu"),
     layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
     layers.MaxPool2D(pool_size=2),

     layers.Flatten(),
     layers.Dense(128, activation="relu"),
     layers.Dense(10, activation="softmax")
])
```

```
base_model.compile(loss='categorical_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

Convolutional Neural Networks

- Image Classification with CNNs
  - Train the model

```
base_history = base_model.fit(x_train, y_train, epochs=10,
validation_data=(X_test, y_test))
```

```
Epoch 1/10
60000/60000 [==============================] - 9s 151us/step - loss: 0.3607 - acc: 0.8718
Epoch 2/10
60000/60000 [==============================] - 9s 142us/step - loss: 0.2224 - acc: 0.9172
Epoch 3/10
60000/60000 [==============================] - 9s 143us/step - loss: 0.1684 - acc: 0.9377
Epoch 4/10
60000/60000 [==============================] - 8s 138us/step - loss: 0.1264 - acc: 0.9533
Epoch 5/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.0928 - acc: 0.9657
Epoch 6/10
60000/60000 [==============================] - 8s 140us/step - loss: 0.0667 - acc: 0.9756
Epoch 7/10
60000/60000 [==============================] - 8s 138us/step - loss: 0.0494 - acc: 0.9822
Epoch 8/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.0361 - acc: 0.9869
Epoch 9/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.0288 - acc: 0.9896
Epoch 10/10
60000/60000 [==============================] - 8s 138us/step - loss: 0.0255 - acc: 0.9908

<tensorflow.python.keras.callbacks.History at 0x279dbfdacf8>
```

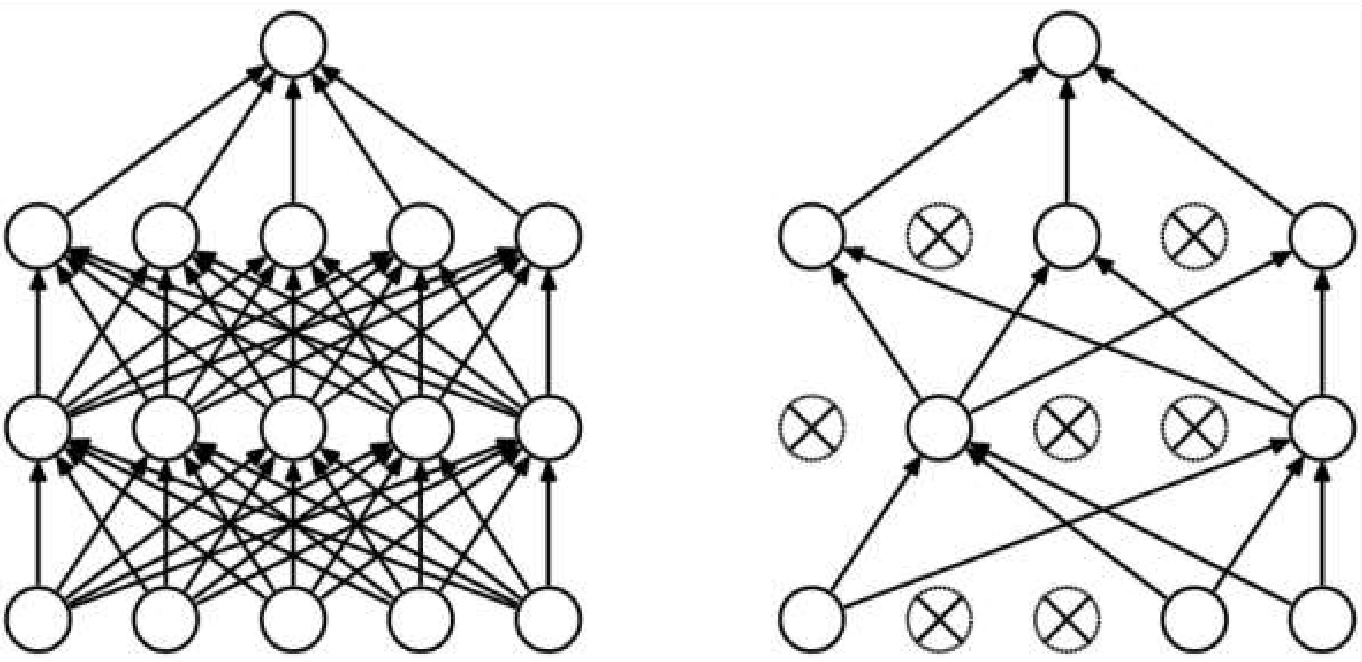
```
base_model.evaluate(x_train, y_train)
```
→ `[0.019858361043675422, 0.9931833333333333]`

```
base_model.evaluate(x_test, y_test)
```
→ `[0.42943426142530516, 0.9202]`

loss *accuracy*

Convolutional Neural Networks

- Avoid the Overfitting, Dropout
  - In each forward pass, randomly set some neurons to zero
  - Probability of dropping is a hyperparameter; 0.5 is common

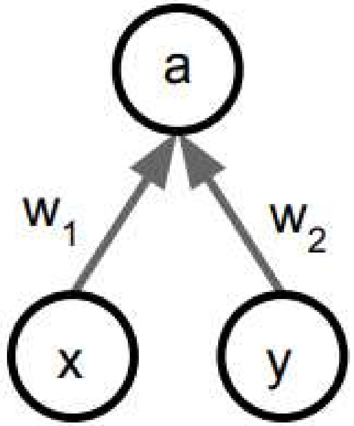


(a) Standard Neural Net (b) After applying dropout.

http://cs231n.stanford.edu/

Convolutional Neural Networks

- Avoid the Overfitting, Dropout
  - Consider a single neuron



  - Training phase
    - Dropout applying probability p (ex_ 0.5)

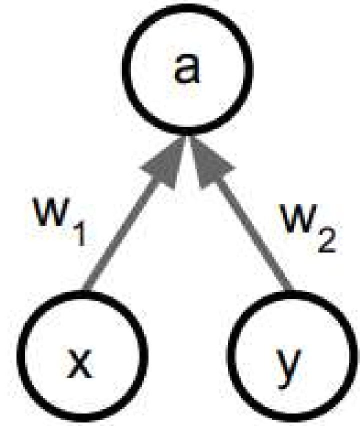
$$a = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + w_2y) + \frac{1}{4}(0x + 0y)$$
$$= \frac{1}{2}(w_1x + w_2y)$$

  - Test phase
    - No dropout

$$a = w_1x + w_2y$$

Convolutional Neural Networks

- Avoid the Overfitting, Dropout
  - Consider a single neuron



  - Training phase
    - Dropout applying probability p (ex_ 0.5)

$$a = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + w_2y) + \frac{1}{4}(0x + 0y)$$
$$= \frac{1}{2}(w_1x + w_2y)$$

  - Test phase
    - No dropout

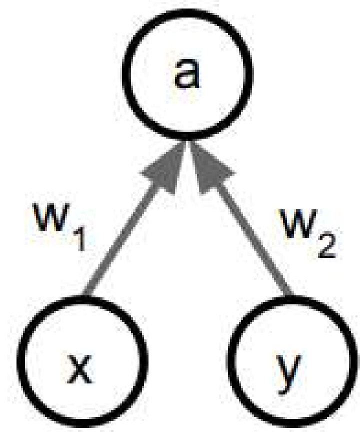
$$a = w_1x + w_2y$$

$\times p$

  - So, we need to multiply p at test time for approximation!

Convolutional Neural Networks

- Avoid the Overfitting, Dropout
  - Consider a single neuron



  - Training phase
    - Dropout applying probability p (ex_ 0.5)

$$a = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + w_2y) + \frac{1}{4}(0x +$$

$$= \frac{1}{2}(w_1x + w_2y)$$

  - Test phase
    - No dropout

$$a = w_1x + w_2y$$

$\times\, 1/p$

  - But commonly, we multiply $1/p$ at training time for approximation!
  - This is called "Inverted Dropout"

Convolutional Neural Networks

- Image Classification with CNNs
  - The model with dropout regularization

```
dropout_model = tf.keras.models.Sequential([
     layers.Conv2D(32, kernel_size=(3, 3), padding = "same",
                            input_shape = (28, 28, 1),
                            activation="relu"),
     layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
     layers.MaxPool2D(pool_size=2),

     layers.Flatten(),
     layers.Dense(128, activation="relu"),
     layers.Dropout(0.5),
     layers.Dense(10, activation="softmax")
])

dropout_model.compile(loss='sparse_categorical_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

Convolutional Neural Networks

- Image Classification with CNNs
  - The model with dropout regularization

```
drop_history = dropout_model.fit(x_train, y_train, epochs=10,
validation_data=(X_test, y_test)))
```

```
Epoch 1/10
60000/60000 [==============================] - 9s 147us/step - loss: 0.4540 - acc: 0.8393
Epoch 2/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.2944 - acc: 0.8937
Epoch 3/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.2476 - acc: 0.9098
Epoch 4/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.2151 - acc: 0.9218
Epoch 5/10
60000/60000 [==============================] - 8s 138us/step - loss: 0.1936 - acc: 0.9286
Epoch 6/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.1668 - acc: 0.9381
Epoch 7/10
60000/60000 [==============================] - 8s 138us/step - loss: 0.1505 - acc: 0.9434
Epoch 8/10
60000/60000 [==============================] - 8s 139us/step - loss: 0.1363 - acc: 0.9485
Epoch 9/10
60000/60000 [==============================] - 8s 138us/step - loss: 0.1235 - acc: 0.9527
Epoch 10/10
60000/60000 [==============================] - 8s 140us/step - loss: 0.1123 - acc: 0.9571

<tensorflow.python.keras.callbacks.History at 0x2770408a7f0>
```

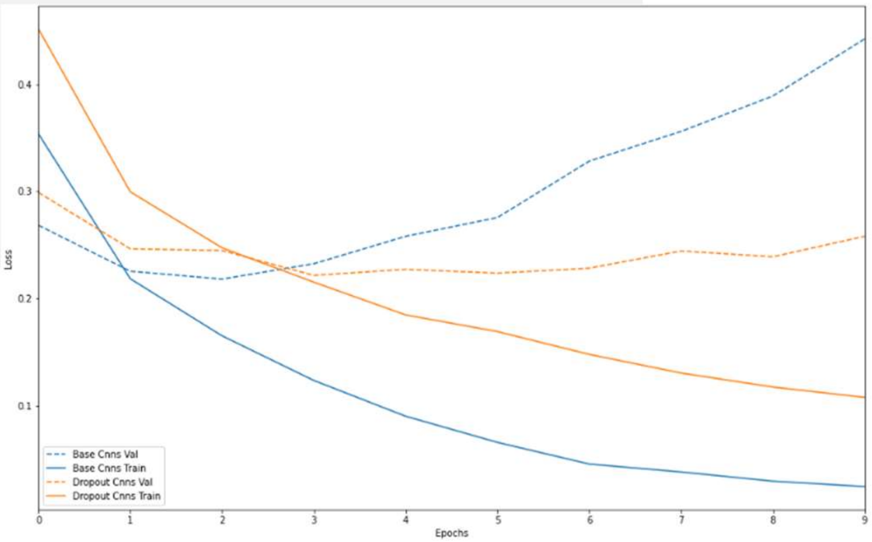
```
dropout_model.evaluate(x_train, y_train)  →  [0.05887163372437159, 0.9790333333333333]

dropout_model.evaluate(x_test, y_test)  →  [0.2695904264975339, 0.9225]
```

loss *accuracy*

Convolutional Neural Networks

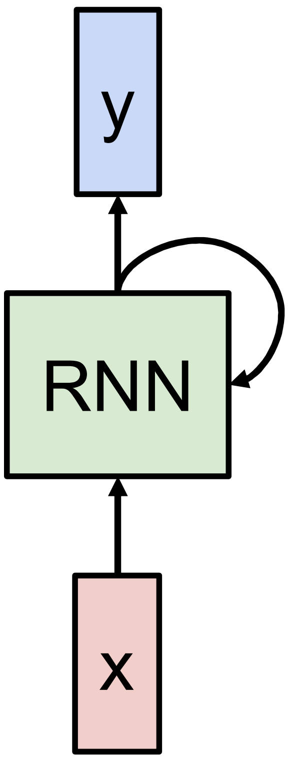
- Image Classification with CNNs
  - Plotting the learning curve

```
def plot_history(histories, key='loss'):
      plt.figure(figsize=(16,10))
      for name, history in histories:
            val = plt.plot(history.epoch, history.history['val_'+key],
            '--', label=name.title()+' Val')
            plt.plot(history.epoch, history.history[key],
            color=val[0].get_color(),
            label=name.title()+' Train')

      plt.xlabel('Epochs')
      plt.ylabel(key.replace('_',' ').title())
      plt.legend()
      plt.xlim([0,max(history.epoch)])

plot_history([('Base CNNs', base_history),
('Dropout CNNs', drop_history)])
```



Recurrent Neural Networks

- Recurrent Neural Networks : for analyzing sequential Data
  - Sequential data : language, video, stock price, weather, …
  - We can process a sequence of vectors **x** (x_1, x_2, x_3, …)
    by applying a **recurrence formula** at every time step



$$y_t = W_{hy} h_t$$

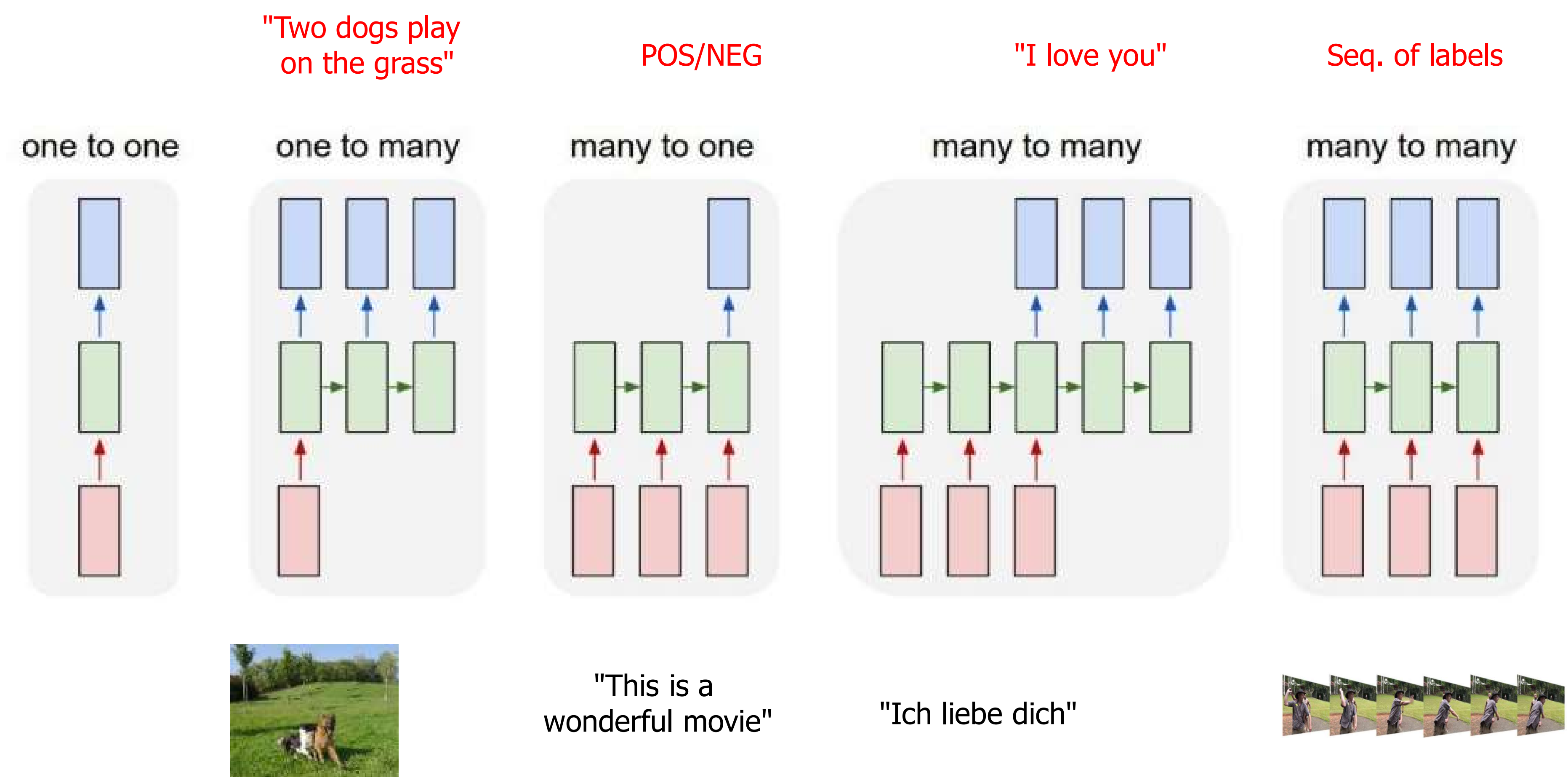
$$h_t = f_W(h_{t-1}, x_t)$$
$$= \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$x_t$$

http://cs231n.stanford.edu

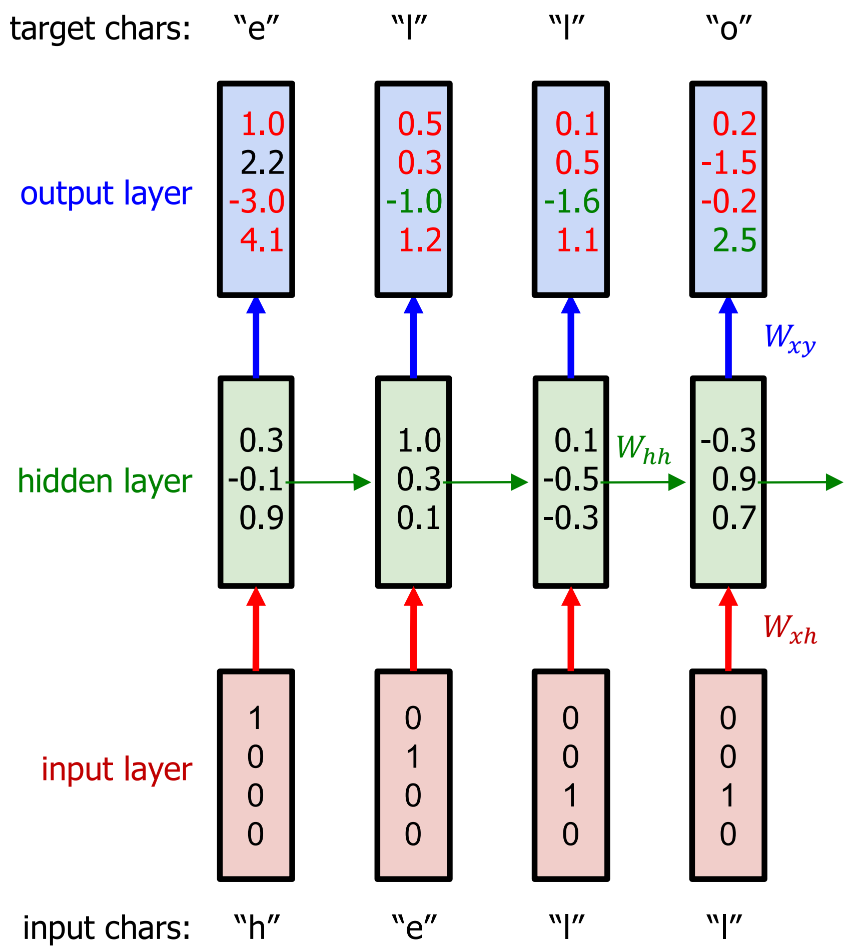
Recurrent Neural Networks

- Various types of input-output relations



Text Generation using RNN

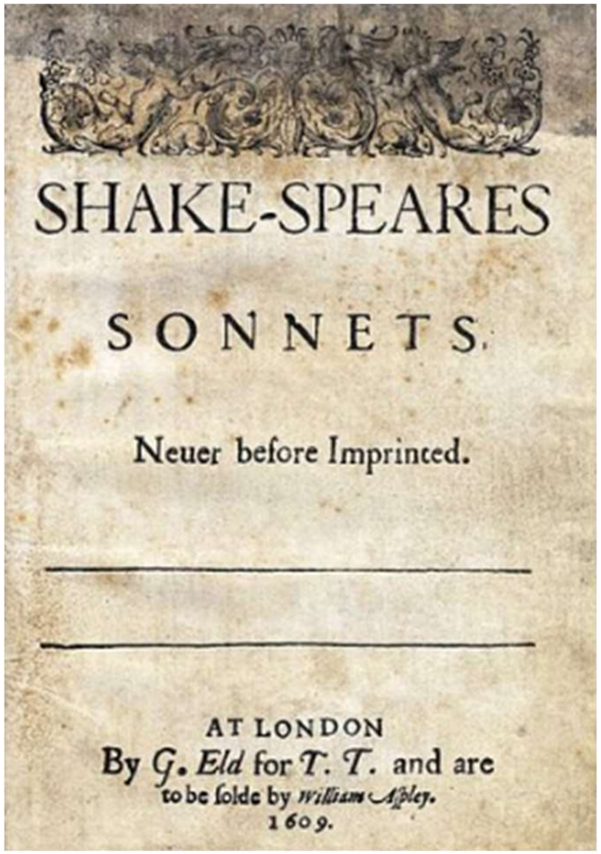
- Example :
  Learning Character-level
  Language Model
  - Vocabulary : [h, e, l, o]
  - Example training sequence : "hello"
  - $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
  - $\hat{y}_t = \mathrm{Softmax}(W_{hy}h_t)$



http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

Recurrent Neural Networks

- Implementing simple RNN with TensorFlow
  - Dataset : Shakespeare's sonnets (Shakespeare.txt)




```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Load and preprocess the Shakespeare dataset

```
import tensorflow as tf
import numpy as np
import os
import time

# 1. Download the Shakespeare's Sonnet dataset
path_to_file = tf.keras.utils.get_file('shakespeare.txt',
'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')

# Load whole text file as a string, then decode.
text = open(path_to_file, 'rb').read().decode(encoding='utf-8')

# length of text is the number of characters in it
print ('Length of text: {} characters'.format(len(text)))
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt
1122304/1115394 [==============================] - 0s 0us/step
Length of text: 1115394 characters
```

```
# We'll use the subset
text = text[:14592]
len(text)
```

```
14592
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Load and preprocess the Shakespeare dataset

```
# Take a look first 250 characters
print(text[:250])
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.
```

```
# The unique characters in the file
vocab = sorted(set(text))
print ('{} unique characters'.format(len(vocab)))
```

```
59 unique characters
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Load and preprocess the Shakespeare dataset

```
# 2. Vectorize the text
# Creating a mapping from unique characters to indices, and vice versa
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

# Convert the characters to the indices
text_as_int = np.array([char2idx[c] for c in text])

# Show how the first 13 characters from the text are mapped to integers
print ('{} ---- characters mapped to int ---- > {}'.format(repr(text[:13]),
text_as_int[:13]))
```

```
'First Citizen' ---- characters mapped to int ---- > [18 47 56 57 58  1 15 47 58 47 64 43 52]
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Creating training examples and targets (tensorflow dataset)

```
# example of TensorFlow dataset
dataset_ex = tf.data.Dataset.from_tensor_slices([10, 20, 30])
for item in dataset_ex:
    print(item)
```

```
tf.Tensor(10, shape=(), dtype=int32)
tf.Tensor(20, shape=(), dtype=int32)
tf.Tensor(30, shape=(), dtype=int32)
```

```
# example of dataset to numpy iterator
it = dataset_ex.as_numpy_iterator()
for element in it:
    print(element)
```

```
10
20
30
```

```
np.array(list(dataset_ex.as_numpy_iterator())).shape
```

```
(3,)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Creating training examples and targets (tensorflow dataset)

```
# Create dataset from text_as_int
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

# check first 5 elements
for item in char_dataset.take(5):
    print(item.numpy())

# Check the shape : char_dataset
np.array(list(char_dataset.as_numpy_iterator())).shape
```

```
15
41
50
51
52

(50000,)
```

```
# example of making batches
dataset_ex = tf.data.Dataset.from_tensor_slices([1, 2, 3, 4, 5, 6, 7])
sequences_ex = dataset_ex.batch(3, drop_remainder=True)

for item in sequences_ex:
    print(item.numpy())
```

```
[1 2 3]
[4 5 6]
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Load and preprocess the Shakespeare dataset

```
# 'batch' method convert these individual characters to sequences of the desired size
sequences = char_dataset.batch(seq_length+1, drop_remainder=True)

for item in sequences.take(5):
     print(repr(''.join(idx2char[item.numpy()])))
```

```
'First Citizen:₩nBefore we proceed any further, hear me speak.₩n₩nAll:₩nSpeak, speak.₩n₩nFirst Citizen
'are all resolved rather to die than to famish?₩n₩nAll:₩nResolved. resolved.₩n₩nFirst Citizen:₩nFirst,
"now Caius Marcius is chief enemy to the people.₩n₩nAll:₩nWe know't, we know't.₩n₩nFirst Citizen:₩nLet
"ll him, and we'll have corn at our own price.₩nIs't a verdict?₩n₩nAll:₩nNo more talking on't; let it I
'one: away, away!₩n₩nSecond Citizen:₩nOne word, good citizens.₩n₩nFirst Citizen:₩nWe are accounted poo
```

```
# Check the shape : sequences
np.array(list(sequences.as_numpy_iterator())).shape
```

```
(495, 101)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Load and preprocess the Shakespeare dataset

```
# map_func
def split_input_target(sequence):
     # input text is shifted to form the target text
     input_text = sequence[:-1]
     target_text = sequence[1:]
     return input_text, target_text

# 'map' method lets us easily apply a simple function to each batch
dataset = sequences.map(split_input_target)

# Print the examples
for input_ex, target_ex in dataset.take(1):
     print ('Input : ', repr(''.join(idx2char[input_ex.numpy()])))
     print ('Target :', repr(''.join(idx2char[target_ex.numpy()])))
```

```
Input :  'First Citizen:₩nBefore we proceed any further, hear me speak.₩n₩nAll:₩nSpeak, speak.₩n₩nFirs
Target : 'irst Citizen:₩nBefore we proceed any further, hear me speak.₩n₩nAll:₩nSpeak, speak.₩n₩nFirst
```

```
# Check the shape : dataset - (1)
np.array(list(dataset.as_numpy_iterator())).shape
```

```
(495, 2, 100)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Load and preprocess the Shakespeare dataset

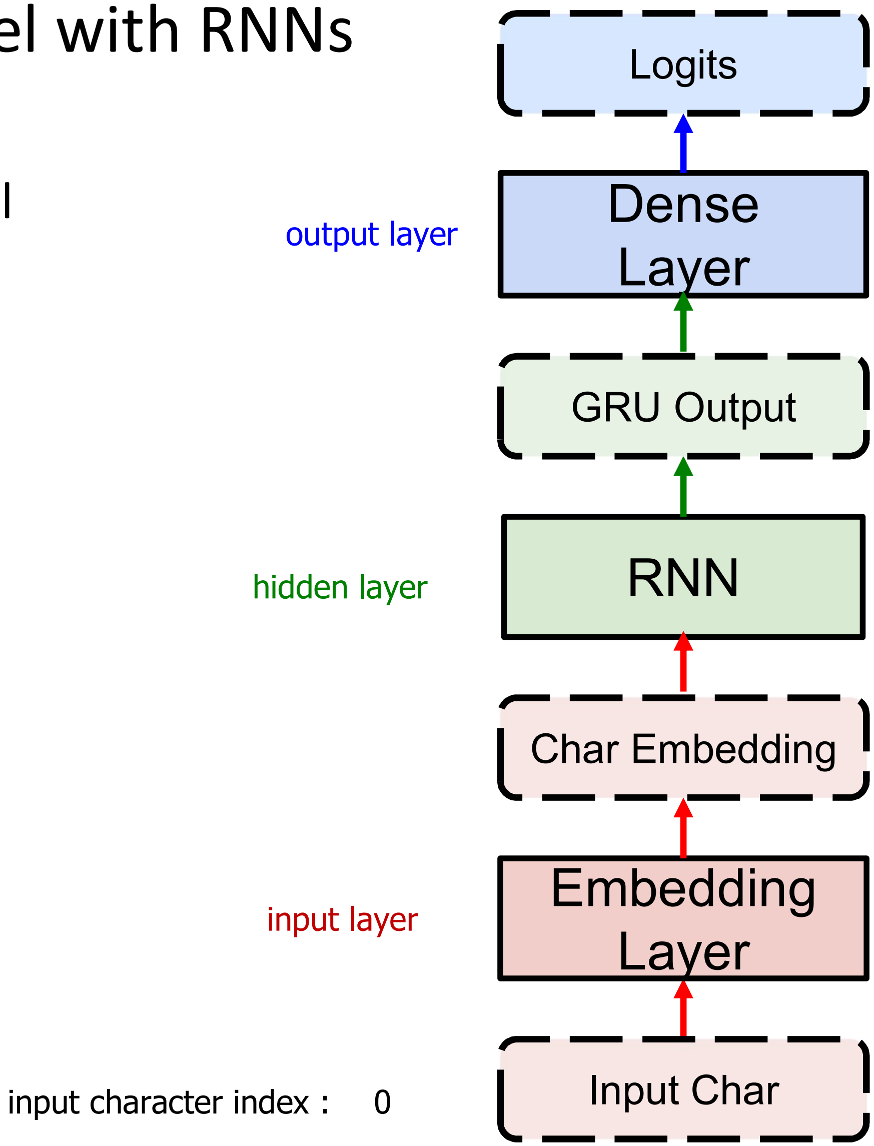
```
# 4. Create training batches
# Batch size
BATCH_SIZE = 16

# Buffer size to shuffle the dataset
BUFFER_SIZE = 100

dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
```

```
# Check the shape : dataset - (2)
np.array(list(dataset.as_numpy_iterator())).shape
```

```
(9, 2, 16, 100)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Build the model
  - 3 layers are used to define this model
    1. tf.keras.layers.Embedding
    2. tf.keras.layers.RNN
    3. tf.keras.layers.Dense



Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Build the model

```
# Length of the vocabulary in chars
vocab_size = len(vocab)
# The embedding dimension
embedding_dim = 128
# Number of RNN units
rnn_units = 256
```

```
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
     layers = tf.keras.layers
     model = tf.keras.Sequential([
          layers.Embedding(input_dim=vocab_size,
                           output_dim=embedding_dim,
                           batch_input_shape=[batch_size, None]
                           ),
          layers.SimpleRNN(rnn_units,
                           return_sequences=True,
                           stateful=True
                           ),
          layers.Dense(vocab_size)
     ])
     return model
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Build the model

```
# Build the model
model = build_model(
     vocab_size = vocab_size,
     embedding_dim=embedding_dim,
     rnn_units=rnn_units,
     batch_size=BATCH_SIZE)

# Check the model architecture
# Model can be run on inputs of any length
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (16, None, 128)           7424
_________________________________________________________________
simple_rnn (SimpleRNN)       (16, None, 256)           98560
_________________________________________________________________
dense_10 (Dense)             (16, None, 58)            14906
=================================================================
Total params: 120,890
Trainable params: 120,890
Non-trainable params: 0
_________________________________________________________________
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Try the model

```
# Run the model to see that it behaves as expected
for input_example_batch, target_example_batch in dataset.take(1):
    example_batch_predictions = model(input_example_batch)

# Check the shape of the input, output
print(input_example_batch.shape)
print(input_example_batch)
print(target_example_batch.shape)
print(target_example_batch)
```

```
(16, 100)
tf.Tensor(
[[ 2  0  0 ... 35 43  4]
 [38 47 50 ... 37  1 57]
 [30 24 21 ... 53 51  1]
 ...
 [40 37 45 ...  7  0 15]
 [46  1 36 ... 33 50 37]
 [ 1 48 33 ... 44 37 46]], shape=(16, 100), dtype=int32)
(16, 100)
tf.Tensor(
[[ 0  0 30 ... 43  4  1]
 [47 50 37 ...  1 57 47]
 [24 21 29 ... 51  1 36]
 ...
 [37 45  1 ...  0 15 41]
 [ 1 36 47 ... 50 37 55]
 [48 33 50 ... 37 46 52]], shape=(16, 100), dtype=int32)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Try the model

```
# Check the shape of the prediction
print(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")
example_batch_predictions[0]
```

```
(16, 100, 59) # (batch_size, sequence_length, vocab_size)

<tf.Tensor: shape=(100, 59), dtype=float32, numpy=
array([[-0.01268253,  0.0158019 , -0.00668244, ...,  0.02808128,
         0.02813094, -0.01326987],
       [ 0.01567873, -0.0699064 , -0.00254133, ...,  0.00900359,
         0.03720927,  0.00952545],
       [-0.01228126, -0.0309497 , -0.02090799, ...,  0.02750907,
         0.00664292, -0.02523949],
       ...,
       [-0.15138564, -0.12352969, -0.18299891, ...,  0.20869936,
         0.01410972,  0.29109895],
       [-0.22140183, -0.01176666,  0.00486418, ...,  0.15129526,
         0.01754857,  0.27657592],
       [-0.07483387, -0.26550916,  0.1061826 , ..., -0.13492906,
        -0.1753861 ,  0.0569937 ]], dtype=float32)>
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Try the model

```
# We need to sample from the output distribution, not to take the argmax of the
distribution
sampled_indices = tf.random.categorical(example_batch_predictions[0],
num_samples=1)
sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()
sampled_indices
```

```
array([29, 25, 20, 20, 45, 46, 22, 40, 40, 36, 25, 20, 51, 57, 10, 48, 11,
       50, 22,  5, 11, 10, 16,  9, 61, 36, 36, 63, 51, 22, 54, 63, 14, 25,
       64, 16, 59, 28, 53, 47, 15, 17, 35, 32, 16, 50, 58, 51,  0,  6,  4,
       28, 23, 27, 52, 59, 12,  9, 45, 53, 63, 13, 28, 52,  3, 19, 30, 64,
       38,  2, 44, 57, 48, 58, 38, 15, 42, 40, 35, 56,  0, 40, 10, 42, 19,
       43, 27, 26,  0, 31, 39,  6, 17, 24, 27, 25, 26, 21, 52, 56])
```

```
# Decode the predictions, the model shows poor performance
print("Input: \n", repr("".join(idx2char[input_example_batch[0]])))
print()
print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices ])))
```

```
Input:
 ':₩nNow, by Saint Paul, this news is bad indeed.₩nO, he hath kept an evil diet long,₩nAnd overmuch consu'

Next Char Predictions:
 "QMHHghJbbXMHms:j;lJ';:D3wXXymJpyBMzDuPoiCE₩TDltm₩n,&PKOnu?3goyAPn$GRzZ!fsjtZCdb₩r₩nb:dGeON₩nSa,ELOMNlnr"
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Train the model

```
# Use the standard tf.keras.losses.sparse_categorical_crossentropy loss
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)

# Use tf.keras.optimizers.Adam optimizer with clipnorm=5.0
optimizer = tf.keras.optimizers.Adam(clipnorm=5.0)

# Configure the training procedure
model.compile(optimizer=optimizer, loss=loss)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Train the model

```
# 2. Configure the checkpoints
# `tf.keras.callbacks.ModelCheckpoint` : The callback function to save the model
checkpoint

# Directory where the model weights will be saved
ckpt_dir = './training_rnns_ckpts'

# Checkpoint name
ckpt_prefix = os.path.join(ckpt_dir, "ckpt_rnns_{epoch}")

# Callback function to save the model weights
ckpt_callback=tf.keras.callbacks.ModelCheckpoint(
filepath=ckpt_prefix,
save_weights_only=True)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Train the model

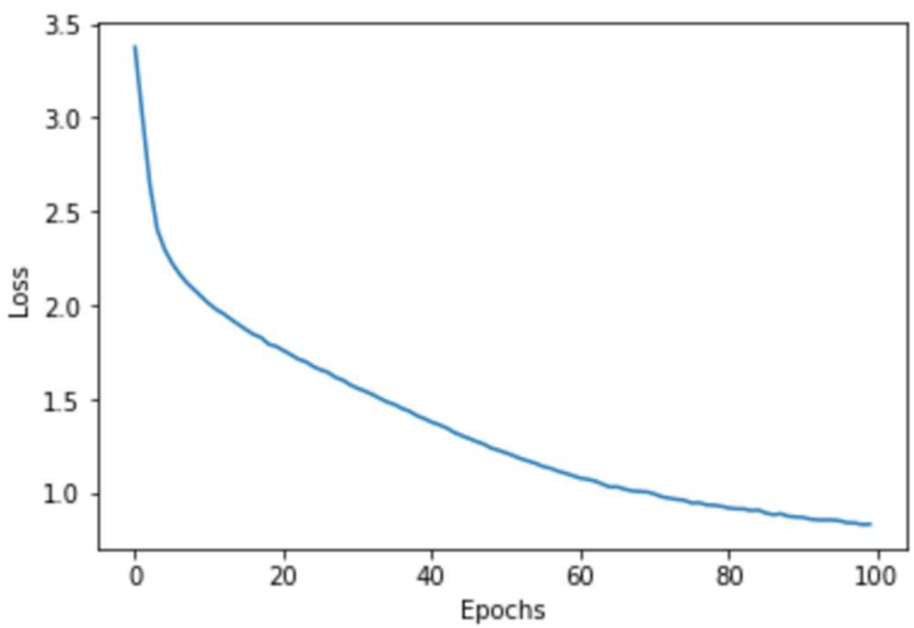
```
# 3. Execute the training
EPOCHS=10
rnn_history = model.fit(dataset, epochs=EPOCHS, callbacks=[ckpt_callback])
```

```
Epoch 1/10
172/172 [==============================] - 20s 114ms/step - loss: 2.6591
Epoch 2/10
172/172 [==============================] - 20s 114ms/step - loss: 2.0140
Epoch 3/10
172/172 [==============================] - 20s 114ms/step - loss: 1.7984
Epoch 4/10
172/172 [==============================] - 20s 114ms/step - loss: 1.6711
Epoch 5/10
172/172 [==============================] - 20s 115ms/step - loss: 1.5867
Epoch 6/10
172/172 [==============================] - 20s 114ms/step - loss: 1.5296
Epoch 7/10
172/172 [==============================] - 20s 114ms/step - loss: 1.4854
Epoch 8/10
172/172 [==============================] - 20s 114ms/step - loss: 1.4529
Epoch 9/10
172/172 [==============================] - 20s 114ms/step - loss: 1.4287
Epoch 10/10
172/172 [==============================] - 20s 114ms/step - loss: 1.4027
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Plot the loss

```
# Plot the loss
plt.plot(rnn_history.epoch, rnn_history.history['loss'])
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Generate text

```
# Check the latest checkpoint
tf.train.latest_checkpoint(ckpt_dir)
```

```
'./training_rnns_ckpts/ckpt_rnns_10'
```

```
# To run the model with one sample(not with batch_size of samples),
# We rebuild the model, and load the weights from the saved checkpoint.
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)
model.load_weights(tf.train.latest_checkpoint(ckpt_dir))
```

```
# Check the model summary
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (1, None, 128)            7552

 simple_rnn_3 (SimpleRNN)    (1, None, 256)            98560

 dense_7 (Dense)             (1, None, 59)             15163

=================================================================
Total params: 121,275
Trainable params: 121,275
Non-trainable params: 0
_________________________________________________________________
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Generate text – (1)

```
# The prediction loop
def generate_text(model, start_string, n_generate, display):

    # Converting start_strings to index (vectorizing)
    input_eval = [char2idx[s] for s in start_string]
    input_eval = tf.expand_dims(input_eval, 0)

    # Making the empty list to store results
    text_generated = []

    # Here batch size == 1
    model.reset_states()
    for i in range(n_generate):
        if(display): print(input_eval.numpy())

        predictions = model(input_eval)

        # remove the batch dimension
        predictions = tf.squeeze(predictions, 0)
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Generate text – (2)

```
    # using a categorical distribution to predict the character
    predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()
    if(display): print(predicted_id)

    # Passing the predicted character as the next input to the model along with the previous hidden state
    input_eval = tf.expand_dims([predicted_id], 0)

    text_generated.append(idx2char[predicted_id])

return (start_string + ''.join(text_generated))
```

Recurrent Neural Networks

- Character-level Language Model with RNNs
  - Generate text

```
# Generate text from start string "All: "
# Test 10 generation with display of input/prediction
print(generate_text(model, start_string="All: ", n_generate=10, display=1))
```

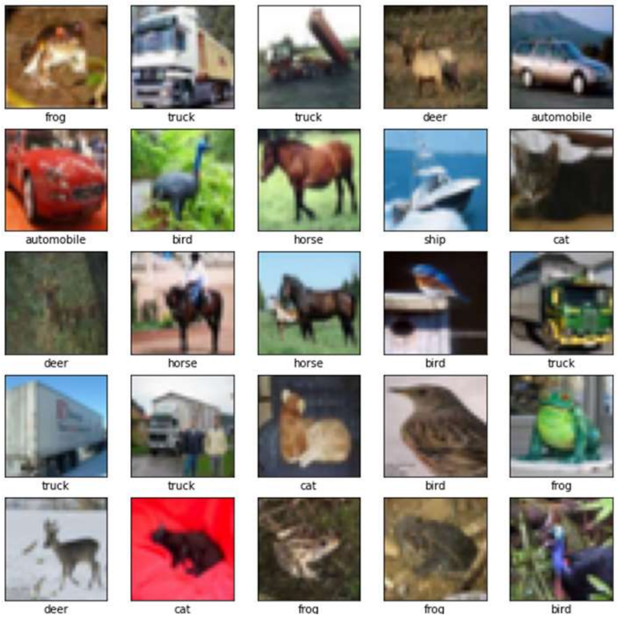
```
# Generate text from start string "All: "
# 1000 generation without display
print(generate_text(model, start_string="All: ", n_generate=1000, display=0))
```

```
All: Marcius.

AUFIDIUS:
Is it good falem whil the they was and aboot well.

BRUTUS:
Before as in the nobs'd would, shald fell st as.

MENENIUS:
I was a fould faclers, porcimess, Marcius, it menabye deiter ere confer oun nofting he
caves a guld with norest;
Well! you grached and corict,
Let umo wor un hy prater liget if
for whether thous
shil. He,
Cond, madar,, by the hather?
```

Quiz 1 : Image Classification Model on the CIFAR-10

- Build the Convolutional Neural Networks
  - Build the model following the bellow model summary
  - Apply the dropout regularization to the model and compare the result
- Compare the performance of the model built last week

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 128)               401536
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 422,218
Trainable params: 422,218
Non-trainable params: 0
```



Quiz 2 : Character-level Language Model

- Build the Character-level Language Model with LSTM
- Compare the generated text to the one generated by RNNs