

# IMAGE ANNOTATION

By Team CODELL

Members:

Chetan Pediredla

Harshit Singh

Khushi Sahoo

Saswat Samal

Saumya Panda

# The problem

## Aimed towards

Visually impaired people do not have the privilege to view images.

Describing the content of the images in words will help them to understand and appreciate the images.

## Context

Describing an image, also known as image annotation is a processing converting an image to text.

## Problem statement

We are expecting the team to build a model that can analyze an image, understand its contents and translate it into a human readable text format.

# Challenges deep-dive

## Challenge 1

### **Object classification**

Take simple images to start which contain objects like ball, hat, human etc.

## Challenge 2

### **Caption generation**

Should be extensible to translate complex image.

## Challenge 3

### **Providing free end-to-end solution**

Product creation

# Solution

## WEB APPLICATION

The solution involves creation of a functional, deployable web app that automatically generates captions for uploaded images.

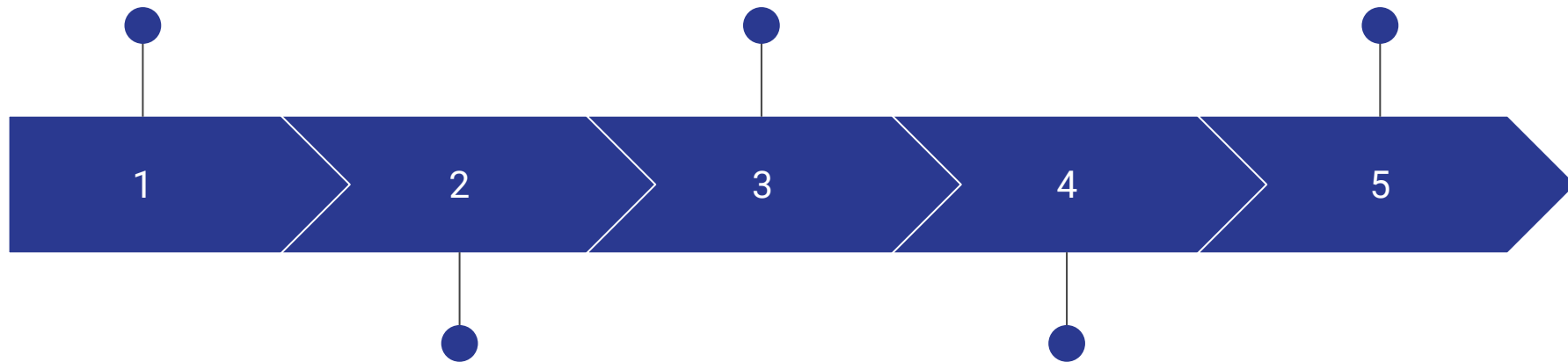
---

# Our Evaluation

OBJECT DETECTION  
MODEL

Inception V3 + LSTM

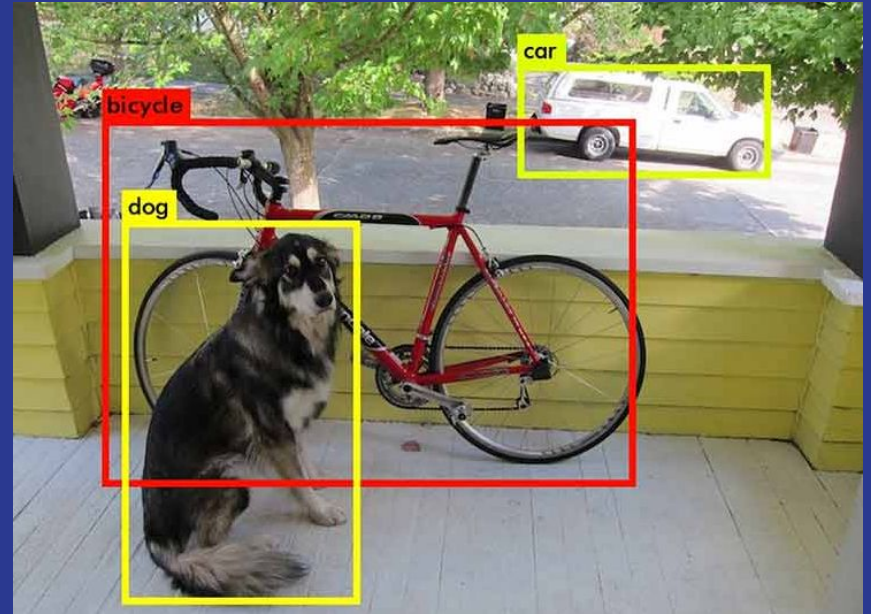
FLASK INTEGRATION



VGG-16 + LSTM

PROPOSED SOLUTION  
USING RESNET-50 &  
LSTM

# OBJECT DETECTION



# DATASET COLLECTION

There are many open source datasets:

- Flickr 8K
  - Flickr 30K
  - MS COCO with 180K images
- And many more

The Flickr 8K dataset contains 8000 images and each image contains 5 different captions.



69189650\_6687da7280.jpg, A brown dog is running through a brown field .  
69189650\_6687da7280.jpg, A brown dog is running through the field .  
69189650\_6687da7280.jpg, A brown dog with a collar runs in the dead  
grass with his tongue hanging out to the side .  
69189650\_6687da7280.jpg, a brown dog with his tongue wagging as he  
runs through a field  
69189650\_6687da7280.jpg, A dog running in the grass .

---

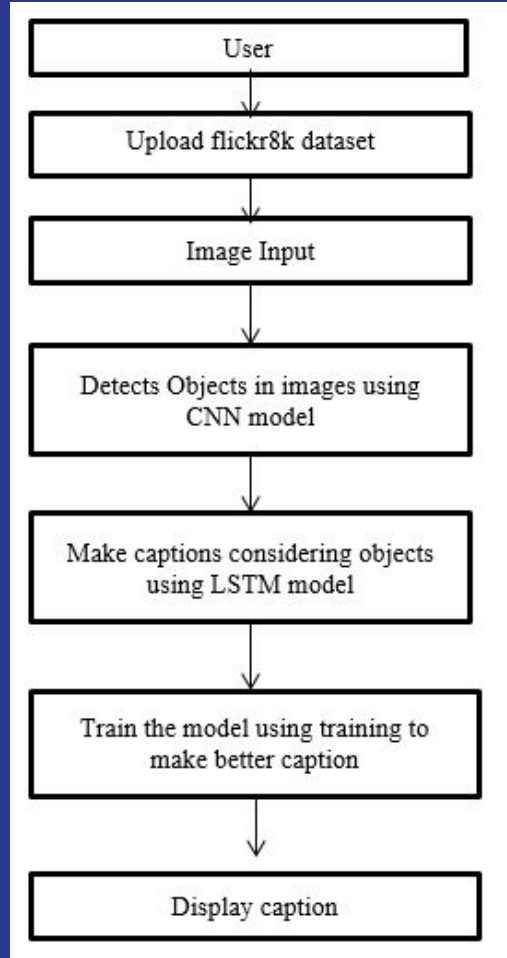


# FLOW CHART

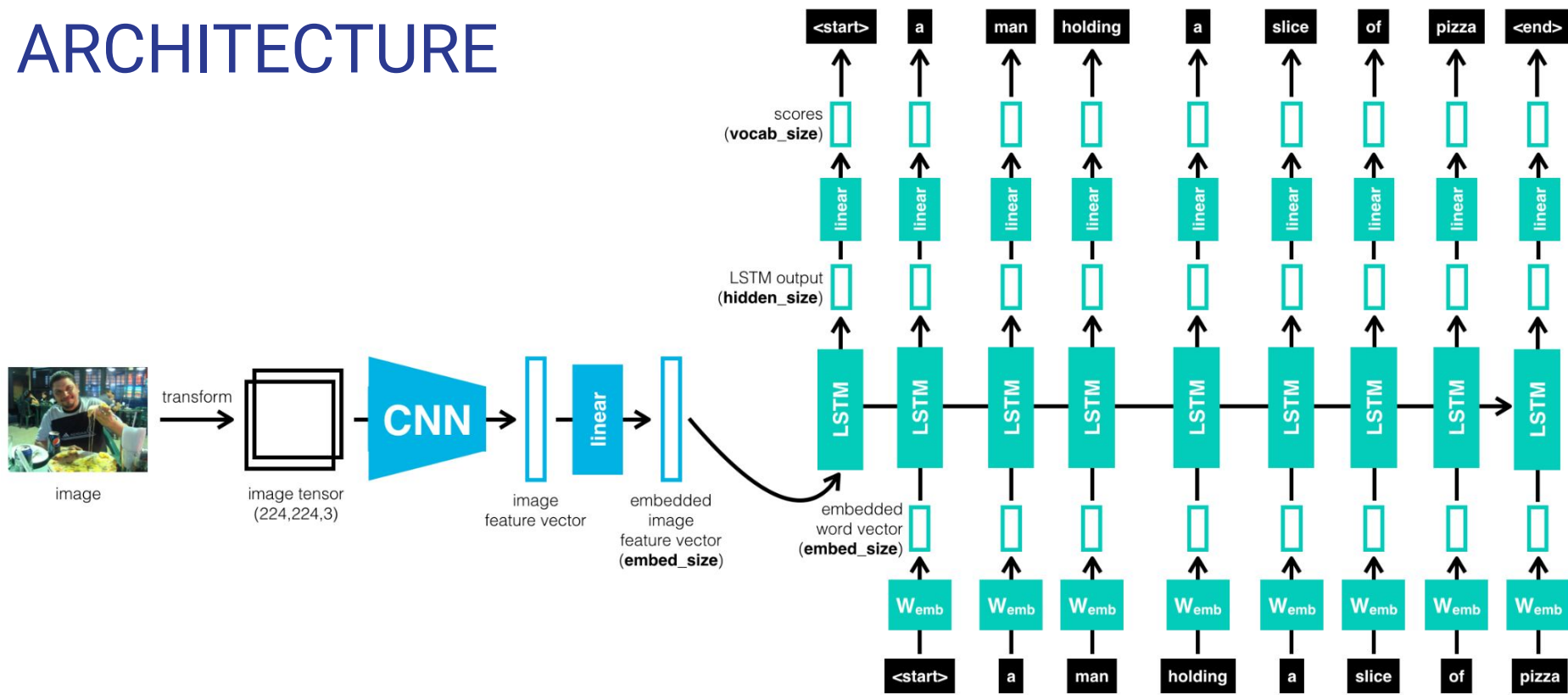


**Image**

**Caption**

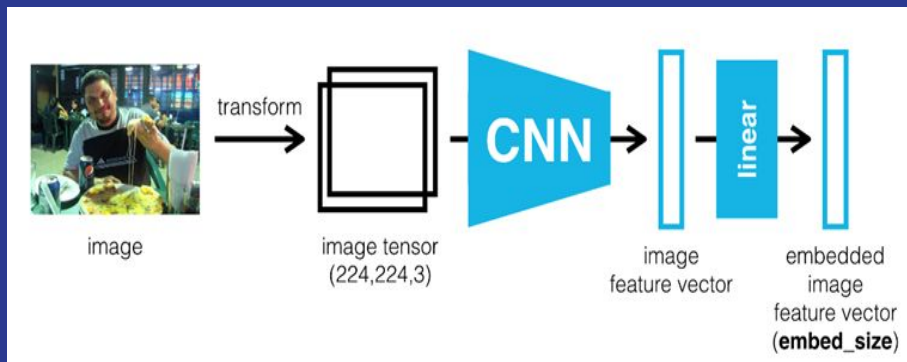


# PROPOSED ARCHITECTURE



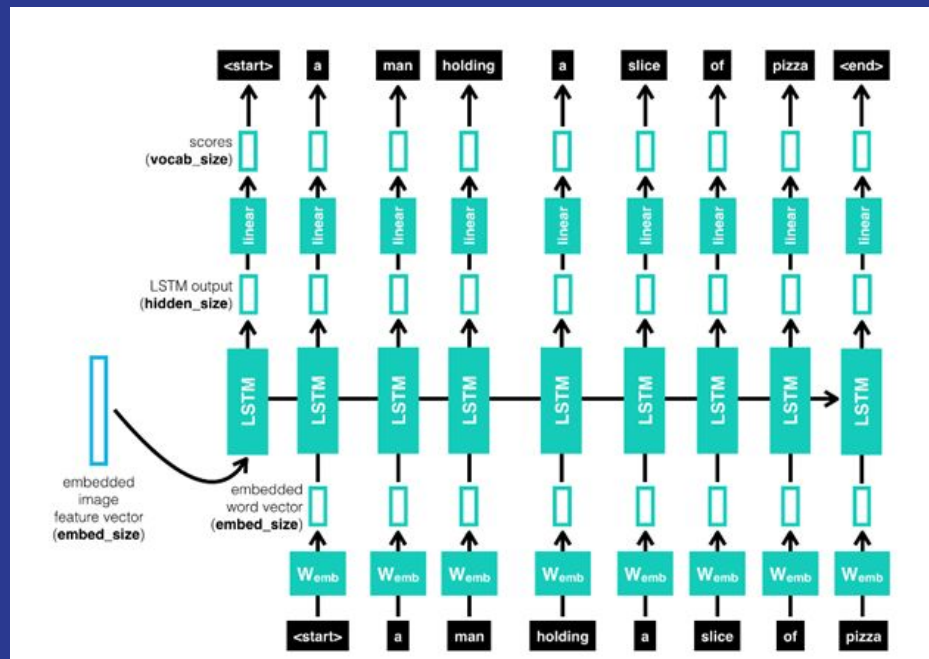
# Encoder:

- The encoder is based on a Convolutional Neural Network that encodes an image into a featurized compact representation (in the form of an embedding).



# Decoder:

- The feature vector is fed into the "Decoder RNN". Each word appearing as output at the top is fed back to the network as input (at the bottom) in a subsequent time step, until the entire caption is generated.
- The arrow pointing to the right that connects the LSTM boxes together represents hidden state information, which represents the network's "memory", also fed back to the LSTM at each time step.



# Training:

The output from the last hidden state of the CNN(Encoder) is given to the first time step of the decoder. We set  $x_1 = \langle \text{START} \rangle$  vector and the desired label  $y_1 = \text{first word in the sequence}$ . Analogously, we set  $x_2 = \text{word vector of the first word}$  and expect the network to predict the *second word*. Finally, on the last step,  $x_T = \text{last word}$ , the target label  $y_T = \langle \text{END} \rangle$  token.

During training, the correct input is given to the decoder at every time-step, even if the decoder made a mistake before.

# Testing:

**$\langle \text{START} \rangle$** A dog running in the grass  **$\langle \text{END} \rangle$**

The image representation is provided to the first time step of the decoder. Set  $x_1 = \langle \text{START} \rangle$  vector and compute the distribution over the first word  $y_1$ . We sample a word from the distribution (or pick the argmax), set its embedding vector as  $x_2$ , and repeat this process until the  $\langle \text{END} \rangle$  token is generated. During Testing, the output of the decoder at time  $t$  is fed back and becomes the input of the decoder at time  $t+1$

# Creating Dictionaries:

**Image\_features={}**

Key : image name

Value : output form model(ResNet/Vgg  
16/Inception)

```
images_features = {}
count = 0
for i in images:
    img = cv2.imread(i)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (224,224))

    img = img.reshape(1,224,224,3)
    pred = modele.predict(img).reshape(2048,)

    img_name = i.split('/')[0]

    images_features[img_name] = pred

    count += 1

    if count > 7999:
        break

    elif count % 50 == 0:
        print(count)
```

**Captions\_dict={}**

Key : image name

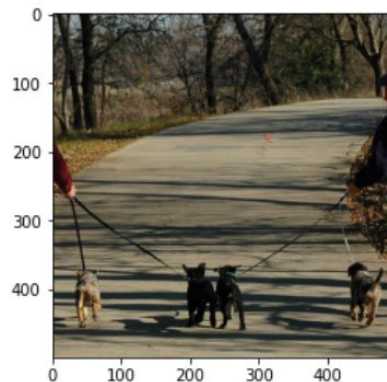
Value : captions

```
captions_dict = {}
for i in captions:
    try:
        img_name = i.split(',')[0]
        caption = i.split(',')[1]
        if img_name in images_features:
            if img_name not in captions_dict:
                captions_dict[img_name] = [caption]

            else:
                captions_dict[img_name].append(caption)

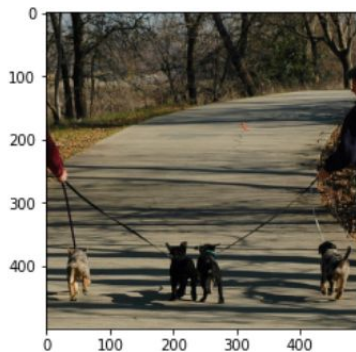
    except:
        pass
```

**Create Vocabulary**



['Four dogs are being walked by two owners .', 'Four dogs on leashes walk down a sidewalk .', 'Four little dogs on leashes take a walk on a wide path .', 'Four small dogs on leashes walking in a park .', 'Someone is walking their dogs .']

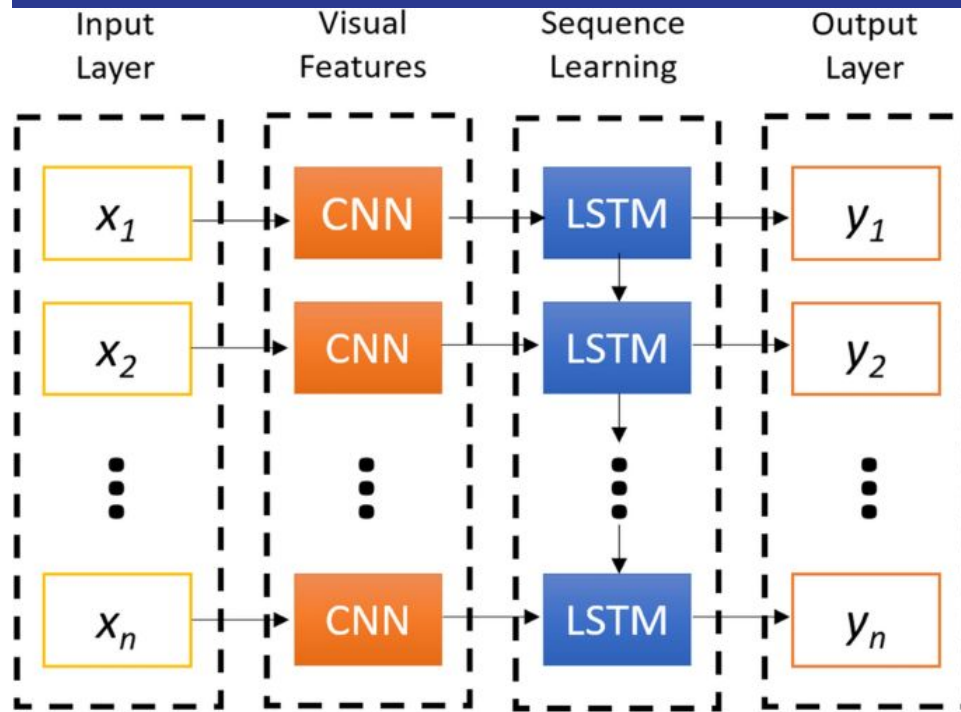
# Word Embedding



[[1, 367, 84, 131, 644, 2760, 298, 2, 2761, 14, 15], [1, 367, 84, 50, 1791, 256, 101, 6, 471, 14, 15], [1, 367, 137, 84, 50, 1791, 826, 6, 256, 50, 6, 1453, 108, 14, 15], [1, 367, 169, 84, 50, 1791, 174, 39, 6, 86, 14, 15], [1, 1204, 31, 174, 116, 84, 14, 15]]

# VGG-16

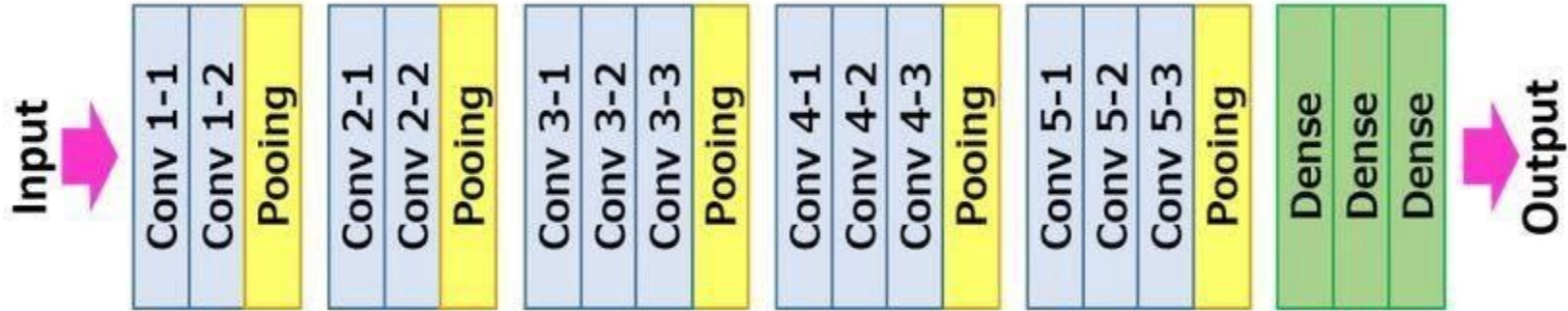
with  
LSTM





# Architecture

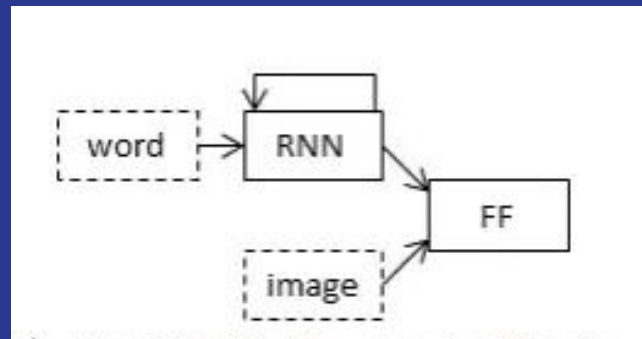
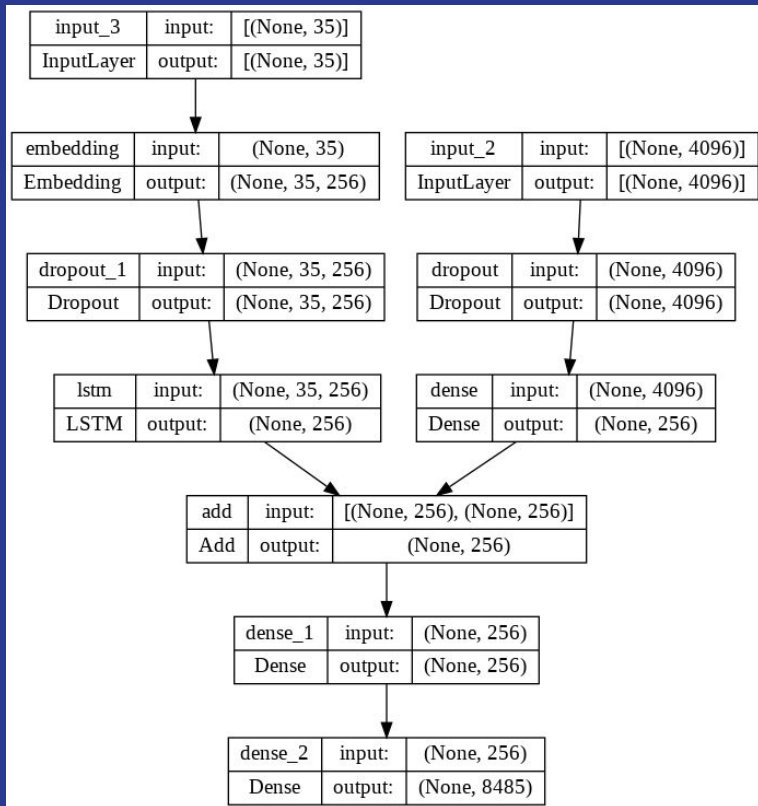
## VGG-16



# Steps

1. Extract Image Features
2. Load, split and append the captions data with the image
3. Preprocess Text Data
4. Train Test Split
5. Model Creation
6. Train the model

# Model



# Results

	Epochs	Batch Size	Image Data Taken	Accuracy
VGG16	25	32	90%	43%
VGG16	50	32	70%	57%

```
epochs = 50
batch_size = 32
# batch_size = 64
steps = len(train) // batch_size

for i in range(epochs):
    # create data generator
    generator = data_generator(train, mapping, features, tokenizer, max_length, vocab_size, batch_size)
    # fit for one epoch
    model.fit(generator, epochs=1, steps_per_epoch=steps, verbose=1)
```

```
176/176 [=====] - ETA: 0s - loss: 1.5898 - accuracy: 0.5568WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 44s 251ms/step - loss: 1.5898 - accuracy: 0.5568
176/176 [=====] - ETA: 0s - loss: 1.5823 - accuracy: 0.5575WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 47s 268ms/step - loss: 1.5823 - accuracy: 0.5575
176/176 [=====] - ETA: 0s - loss: 1.5716 - accuracy: 0.5604WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 45s 256ms/step - loss: 1.5716 - accuracy: 0.5604
176/176 [=====] - ETA: 0s - loss: 1.5634 - accuracy: 0.5625WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 44s 252ms/step - loss: 1.5634 - accuracy: 0.5625
176/176 [=====] - ETA: 0s - loss: 1.5560 - accuracy: 0.5639WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 44s 248ms/step - loss: 1.5560 - accuracy: 0.5639
176/176 [=====] - ETA: 0s - loss: 1.5472 - accuracy: 0.5658WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 45s 254ms/step - loss: 1.5472 - accuracy: 0.5658
176/176 [=====] - ETA: 0s - loss: 1.5401 - accuracy: 0.5677WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 47s 268ms/step - loss: 1.5401 - accuracy: 0.5677
176/176 [=====] - ETA: 0s - loss: 1.5342 - accuracy: 0.5685WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 47s 267ms/step - loss: 1.5342 - accuracy: 0.5685
176/176 [=====] - ETA: 0s - loss: 1.5262 - accuracy: 0.5696WARNING:tensorflow:Early stopping conditioned on metric `val_loss` which is not available. Available
176/176 [=====] - 55s 314ms/step - loss: 1.5262 - accuracy: 0.5696
```

# Results

	Epochs	Batch Size	Image Data Taken	Accuracy
VGG19	40	128	70%	41%
VGG19	50	128	50%	46%
VGG19	40	64	50%	49%
VGG19	64	32	50%	65%

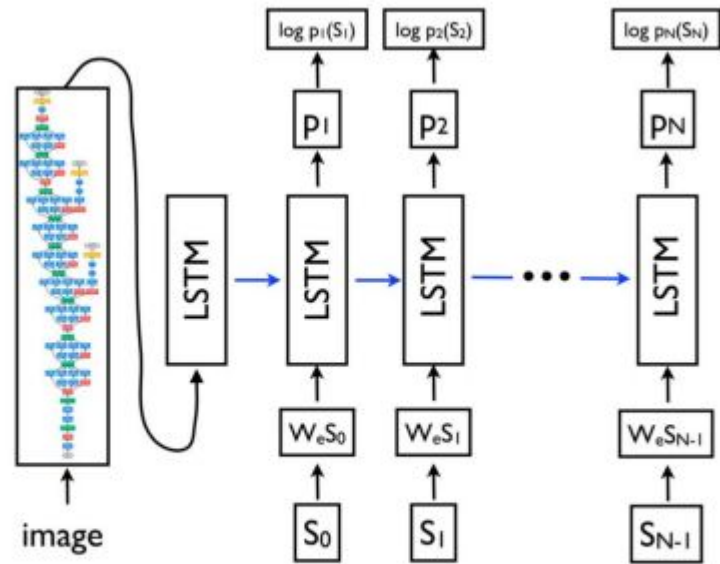
```
# train the model
epochs = 70
batch_size = 32
# batch_size = 64
# batch_size = 128
steps = len(train) // batch_size

for i in range(epochs):
    # create data generator
    generator_train = data_generator(train, mapping, features, tokenizer, max_length, vocab_size, batch_size)
    # generator_test = data_generator(test, mapping, features, tokenizer, max_length, vocab_size, batch_size)
    # fit for one epoch
    # model.fit(generator_train, validation_data=generator_test, epochs=1, steps_per_epoch=steps, callbacks=[earlystopping], verbose=1)
    model.fit(generator_train, epochs=1, steps_per_epoch=steps, verbose=1)
```

1/75 [.....] - ETA: 24s - loss: 1.0948 - accuracy: 0.6535

# INCEPTION-V3

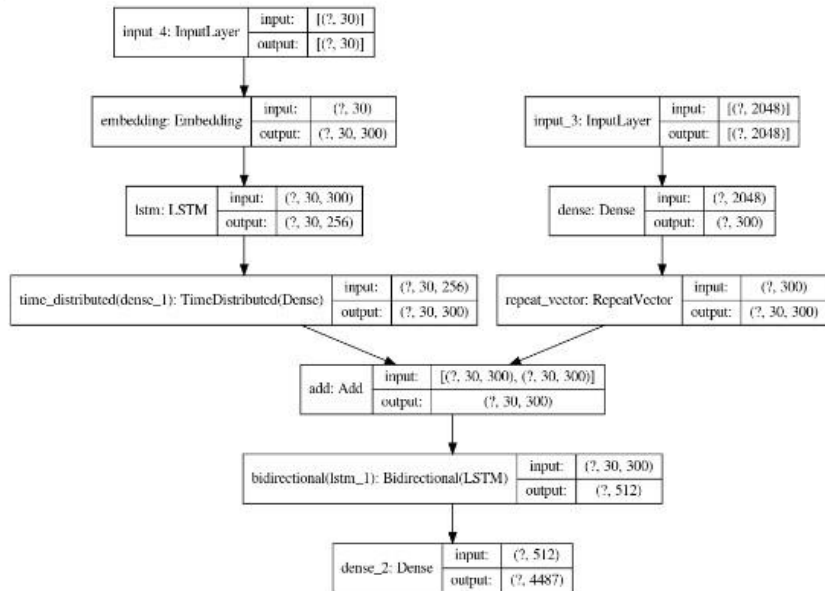
with  
LSTM



# Model Generation

```
In [28]: plot_model(model, to_file='model.png', show_shapes=True)
```

```
Out[28]:
```



```
# fit model
BATCH_SIZE = 16 * strategy.num_replicas_in_sync

EPOCHS = 20

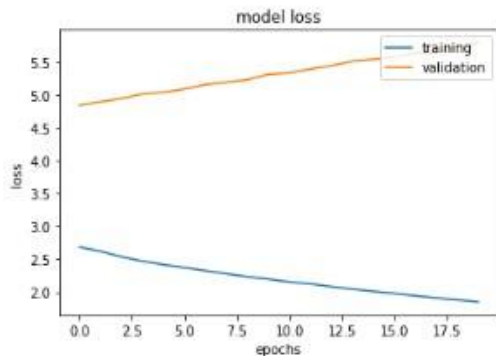
history = model.fit([train_x1, train_x2],
                    train_y,
                    verbose = 1,
                    epochs = EPOCHS,
                    batch_size = BATCH_SIZE,
                    callbacks = callbacks,
                    validation_data=(validate_x1, validate_x2, validate_y))
```

```
Epoch 9/20
566/566 [=====] - 12s 21ms/step - loss: 2.2340 - val_loss: 5.2293
Epoch 10/20
566/566 [=====] - 12s 21ms/step - loss: 2.1978 - val_loss: 5.3119
Epoch 11/20
566/566 [=====] - 12s 21ms/step - loss: 2.1555 - val_loss: 5.3329
Epoch 12/20
566/566 [=====] - 12s 21ms/step - loss: 2.1251 - val_loss: 5.3923
Epoch 13/20
566/566 [=====] - 12s 21ms/step - loss: 2.0832 - val_loss: 5.4425
Epoch 14/20
566/566 [=====] - 12s 21ms/step - loss: 2.0461 - val_loss: 5.5103
Epoch 15/20
566/566 [=====] - 13s 23ms/step - loss: 2.0113 - val_loss: 5.5405
Epoch 16/20
566/566 [=====] - 12s 22ms/step - loss: 1.9801 - val_loss: 5.5701
Epoch 17/20
566/566 [=====] - 12s 21ms/step - loss: 1.9445 - val_loss: 5.6335
Epoch 18/20
566/566 [=====] - 12s 21ms/step - loss: 1.9118 - val_loss: 5.6400
Epoch 19/20
566/566 [=====] - 12s 21ms/step - loss: 1.8824 - val_loss: 5.7397
Epoch 20/20
566/566 [=====] - 12s 21ms/step - loss: 1.8524 - val_loss: 5.7930
```

# Evaluation & Measure

```
In [44]: # plot training loss and validation loss
import matplotlib.pyplot as plt

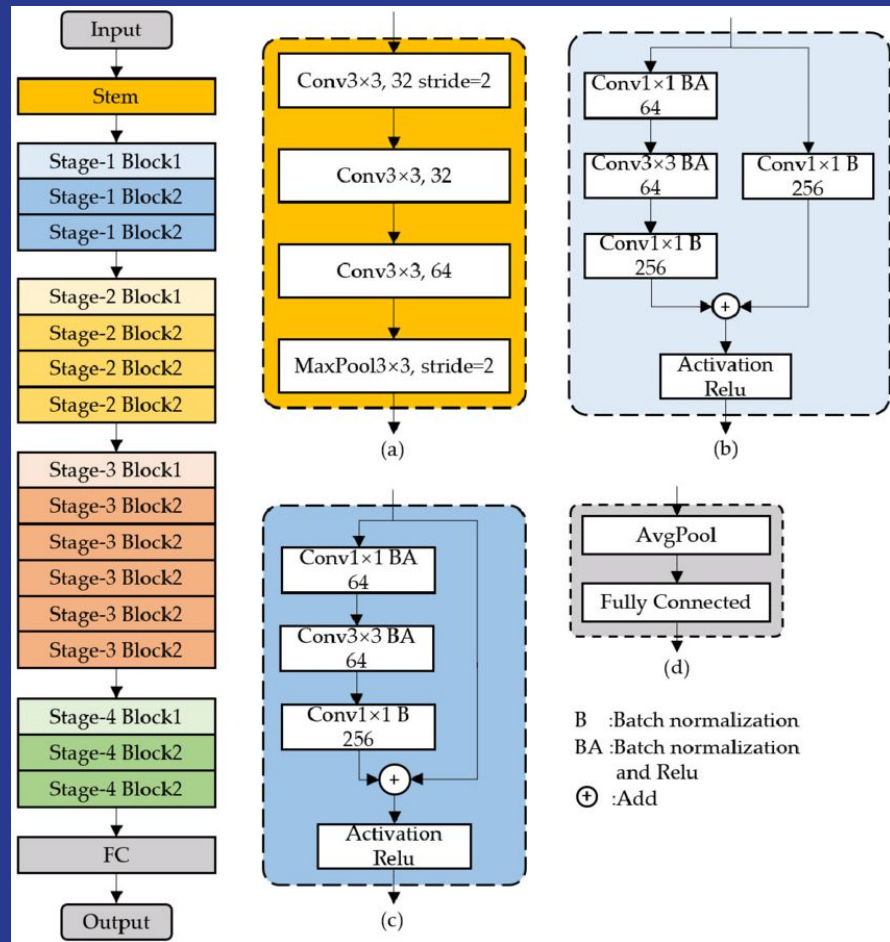
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epochs')
plt.legend(['training', 'validation'], loc='upper right')
plt.show()
```





# FINAL PROPOSED MODEL

using  
RESNET-50 & LSTM



# Brief Overview

ON FINAL PROPOSED SOLUTION

# Cons of Previous Encoders over ResNet-50

## VGG-16

Accuracy of 57% with 50 epochs and Batch size of 32.

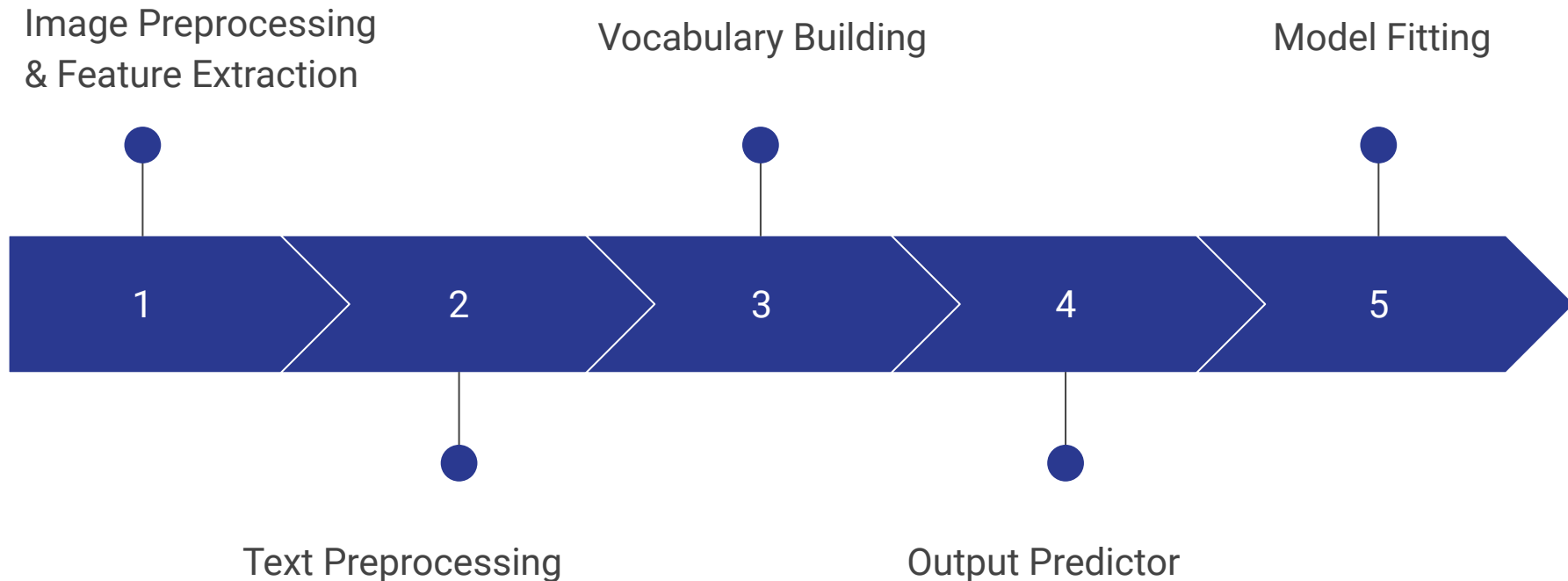
## VGG-19

Accuracy of 46% with 50 epochs and Batch size of 128.

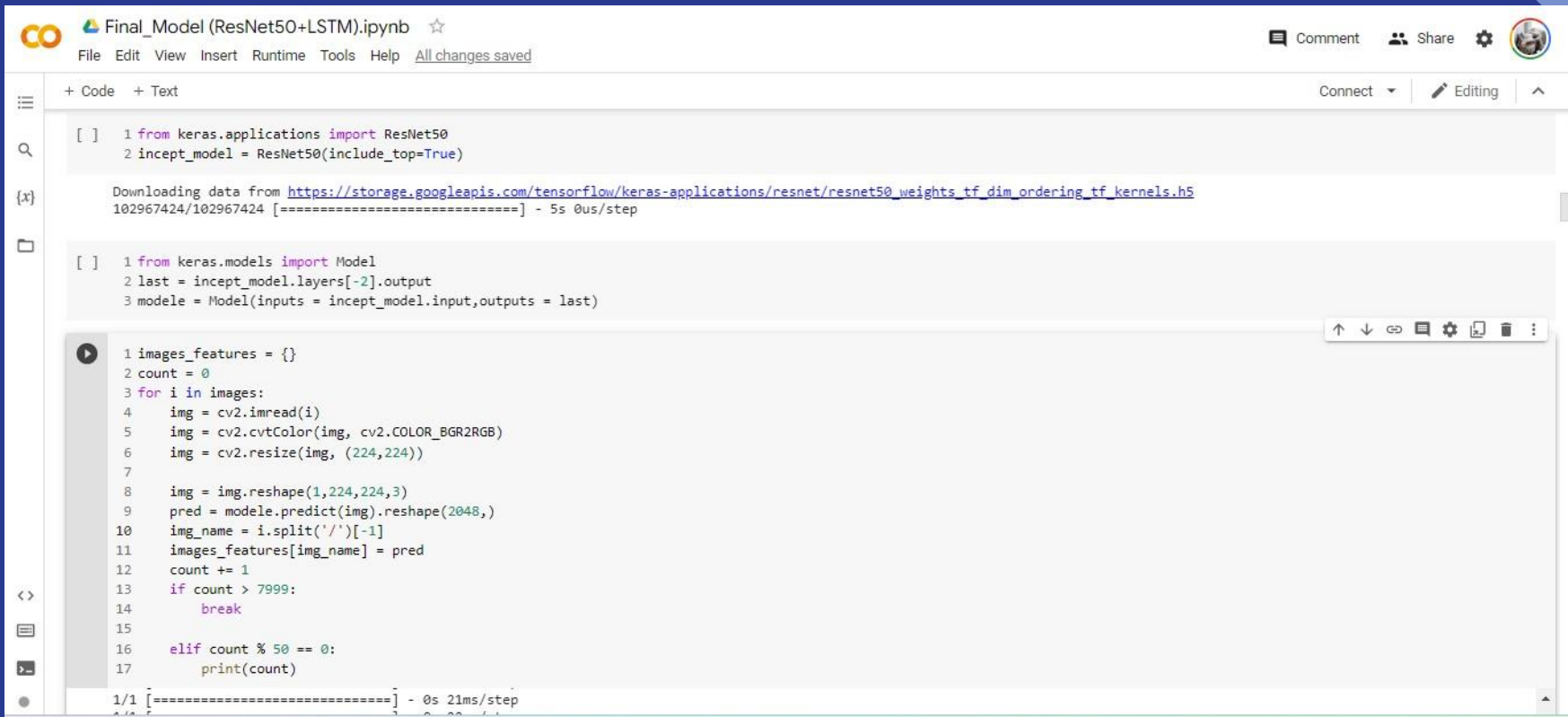
## Inception-V3

Overfitting at an earlier stage.

# Model Generation



# Image Preprocessing & Feature Extraction



The screenshot displays a Jupyter Notebook titled "Final\_Model (ResNet50+LSTM).ipynb". The interface includes a top bar with the Jupyter logo, a menu (File, Edit, View, Insert, Runtime, Tools, Help), and a status bar indicating "All changes saved". On the right, there are buttons for "Comment", "Share", and a settings icon. Below the top bar, the notebook is in "Editing" mode, as indicated by the "Editing" button and the "Connect" dropdown. The code is organized into cells, with the first cell containing two lines of Python code: 

```
[ ] 1 from keras.applications import ResNet50
2 incept_model = ResNet50(include_top=True)
```

 The second cell shows the output of a download operation: 

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels.h5
102967424/102967424 [=====] - 5s 0us/step
```

 The third cell contains three lines of Python code: 

```
[ ] 1 from keras.models import Model
2 last = incept_model.layers[-2].output
3 modele = Model(inputs = incept_model.input, outputs = last)
```

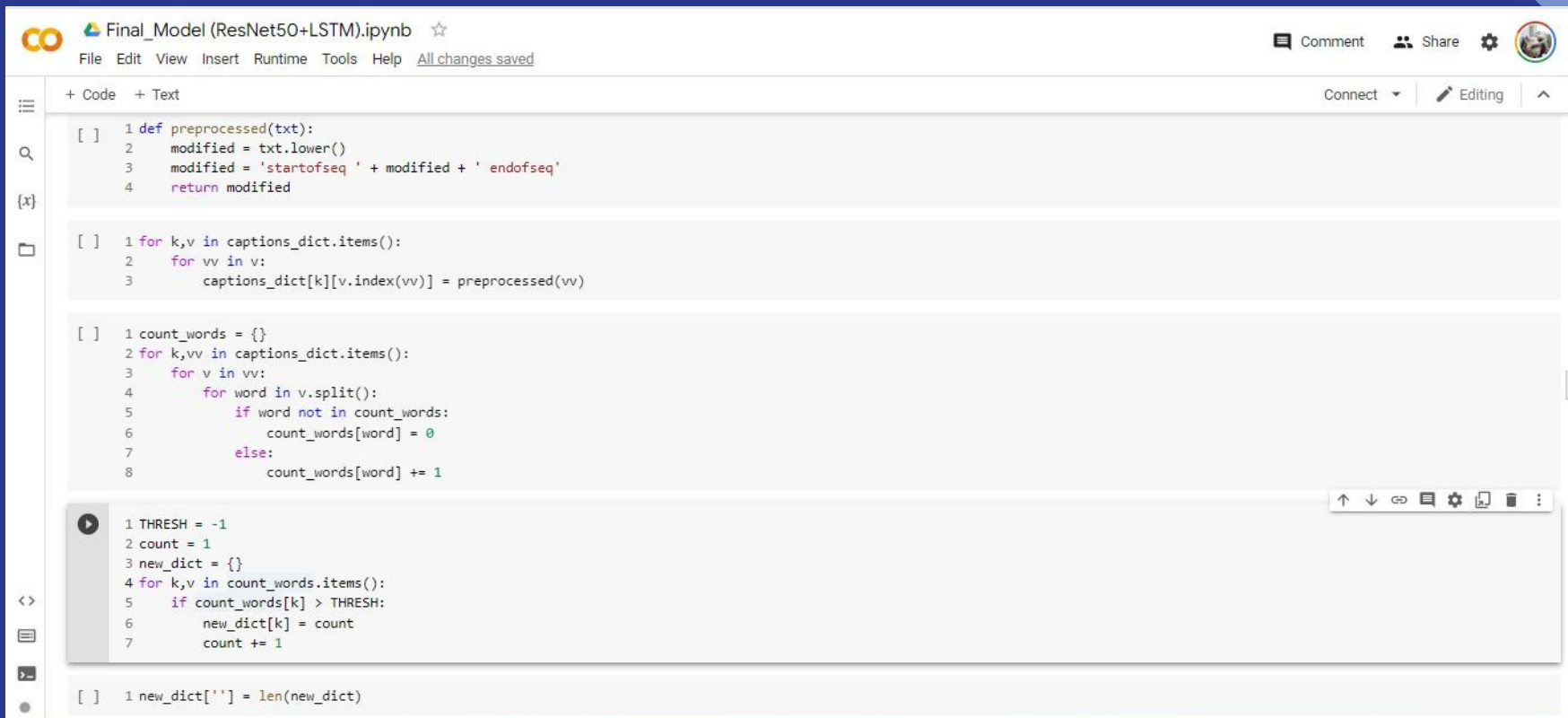
 The fourth cell is a code cell with a play button icon on the left, containing a loop of Python code for image preprocessing: 

```
1 images_features = {}
2 count = 0
3 for i in images:
4     img = cv2.imread(i)
5     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
6     img = cv2.resize(img, (224,224))
7
8     img = img.reshape(1,224,224,3)
9     pred = modele.predict(img).reshape(2048,)
10    img_name = i.split('/')[-1]
11    images_features[img_name] = pred
12    count += 1
13    if count > 7999:
14        break
15
16    elif count % 50 == 0:
17        print(count)
```

 The bottom of the notebook shows the output of the loop: 

```
1/1 [=====] - 0s 21ms/step
```

# Text Preprocessing



The screenshot shows a Jupyter Notebook titled "Final\_Model (ResNet50+LSTM).ipynb". The interface includes a top bar with the Jupyter logo, file editing tools, and user options. The left sidebar contains navigation icons. The main area displays three code cells. The first cell defines a `preprocessed(txt)` function. The second cell iterates over `captions_dict` to apply the preprocessing function. The third cell iterates over `count_words` to filter words based on a threshold. A fourth cell is partially visible at the bottom.

```
[ ] 1 def preprocessed(txt):
2     modified = txt.lower()
3     modified = 'startofseq ' + modified + ' endofseq'
4     return modified

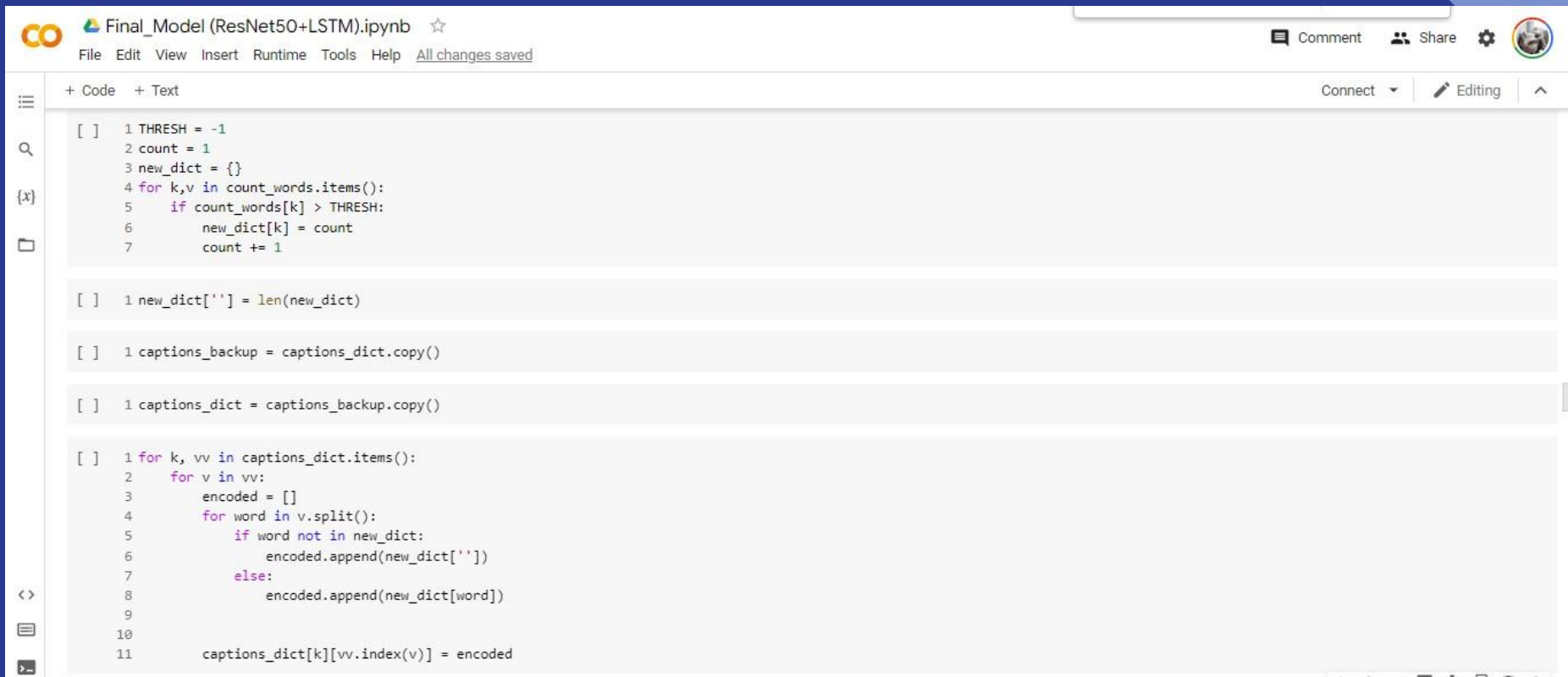
[ ] 1 for k,v in captions_dict.items():
2     for vv in v:
3         captions_dict[k][v.index(vv)] = preprocessed(vv)

[ ] 1 count_words = {}
2 for k,vv in captions_dict.items():
3     for v in vv:
4         for word in v.split():
5             if word not in count_words:
6                 count_words[word] = 0
7             else:
8                 count_words[word] += 1

[ ] 1 THRESH = -1
2 count = 1
3 new_dict = {}
4 for k,v in count_words.items():
5     if count_words[k] > THRESH:
6         new_dict[k] = count
7         count += 1

[ ] 1 new_dict[''] = len(new_dict)
```

# Vocabulary Building



The screenshot displays a Jupyter Notebook titled "Final\_Model (ResNet50+LSTM).ipynb". The interface includes a top bar with the Colab logo, a menu (File, Edit, View, Insert, Runtime, Tools, Help), and a status indicator "All changes saved". On the right, there are buttons for "Comment", "Share", and a settings gear, along with a user profile icon. The left sidebar contains icons for a table of contents, search, variable explorer, and file explorer. The main area shows a code editor with the following Python code:

```
[ ] 1 THRESH = -1
    2 count = 1
    3 new_dict = {}
    4 for k,v in count_words.items():
    5     if count_words[k] > THRESH:
    6         new_dict[k] = count
    7         count += 1

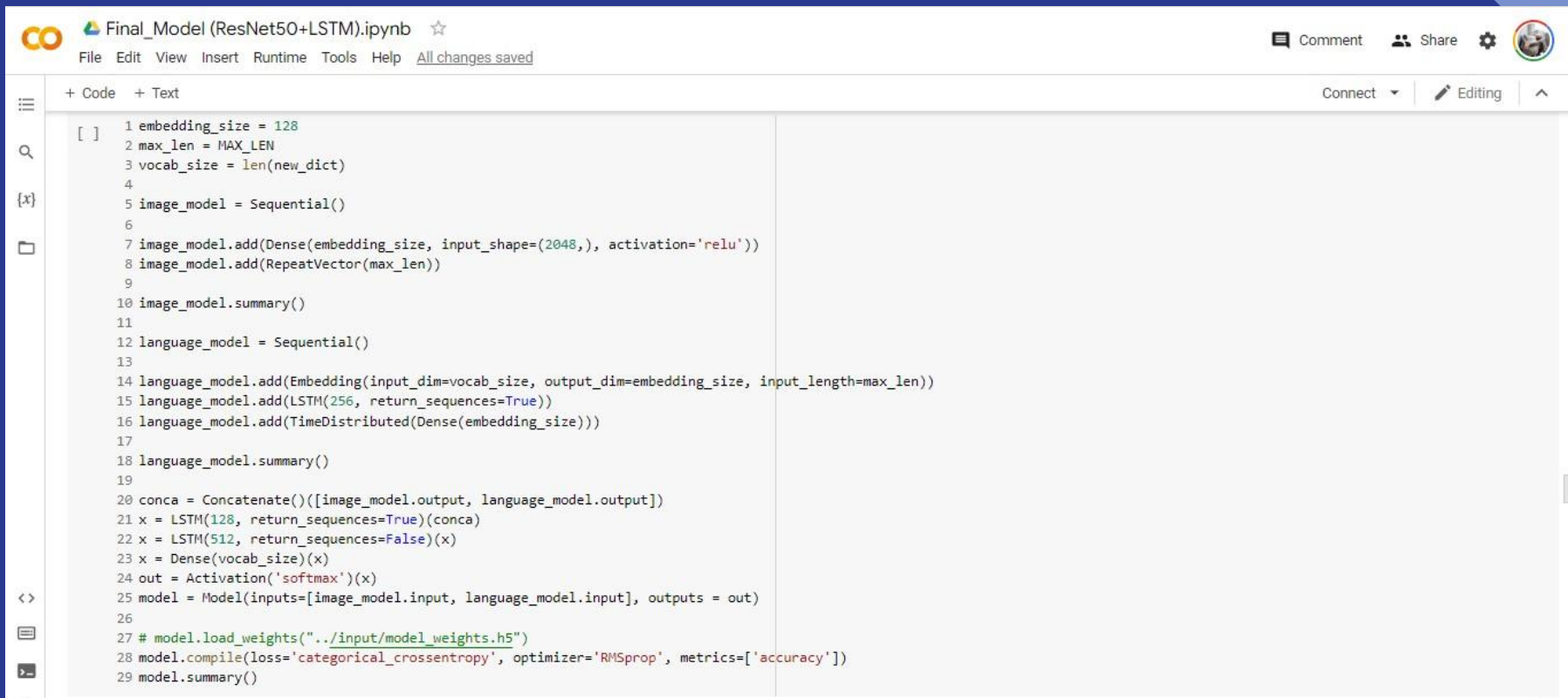
[ ] 1 new_dict[''] = len(new_dict)

[ ] 1 captions_backup = captions_dict.copy()

[ ] 1 captions_dict = captions_backup.copy()

[ ] 1 for k, vv in captions_dict.items():
    2     for v in vv:
    3         encoded = []
    4         for word in v.split():
    5             if word not in new_dict:
    6                 encoded.append(new_dict[''])
    7             else:
    8                 encoded.append(new_dict[word])
    9
   10
   11 captions_dict[k][vv.index(v)] = encoded
```

# Output Predictor



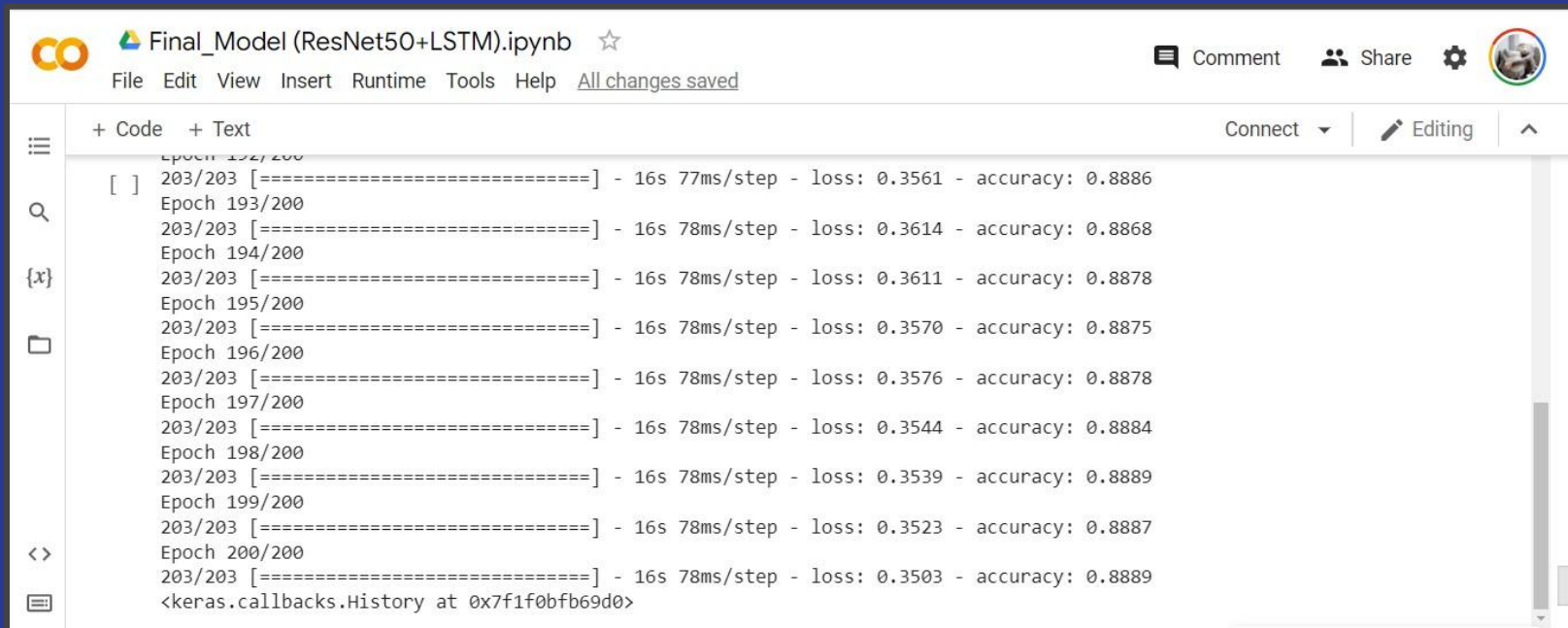
The screenshot shows a Jupyter Notebook titled "Final\_Model (ResNet50+LSTM).ipynb". The notebook contains a Python script that builds a Keras model. The model consists of an image model (ResNet50) and a language model (LSTM). The image model takes an input of shape (2048,) and outputs a vector of size 128. The language model takes an input of size 128 and outputs a vector of size 128. The two outputs are concatenated and passed through another LSTM layer, followed by a dense layer and a softmax activation to produce the final output.

```
[ ] 1 embedding_size = 128
    2 max_len = MAX_LEN
    3 vocab_size = len(new_dict)
    4
    5 image_model = Sequential()
    6
    7 image_model.add(Dense(embedding_size, input_shape=(2048,), activation='relu'))
    8 image_model.add(RepeatVector(max_len))
    9
   10 image_model.summary()
   11
   12 language_model = Sequential()
   13
   14 language_model.add(Embedding(input_dim=vocab_size, output_dim=embedding_size, input_length=max_len))
   15 language_model.add(LSTM(256, return_sequences=True))
   16 language_model.add(TimeDistributed(Dense(embedding_size)))
   17
   18 language_model.summary()
   19
   20 conca = Concatenate()([image_model.output, language_model.output])
   21 x = LSTM(128, return_sequences=True)(conca)
   22 x = LSTM(512, return_sequences=False)(x)
   23 x = Dense(vocab_size)(x)
   24 out = Activation('softmax')(x)
   25 model = Model(inputs=[image_model.input, language_model.input], outputs = out)
   26
   27 # model.load_weights("../input/model_weights.h5")
   28 model.compile(loss='categorical_crossentropy', optimizer='RMSprop', metrics=['accuracy'])
   29 model.summary()
```



# Fitting the Model

```
[ ] 1 model.fit([X, y_in], y_out, batch_size=512, epochs=200)
```



The screenshot shows a Jupyter Notebook titled "Final\_Model (ResNet50+LSTM).ipynb". The interface includes a top bar with the Colab logo, file management icons, and a menu (File, Edit, View, Insert, Runtime, Tools, Help). On the right, there are buttons for Comment, Share, and a settings gear. Below the menu is a toolbar with "+ Code" and "+ Text" buttons, a "Connect" dropdown, and an "Editing" mode indicator. The main area displays the output of a model fit over 200 epochs. Each epoch's output shows the time taken (16s), steps per second (77ms/step), loss, and accuracy. The loss decreases from 0.3561 to 0.3503, while accuracy remains stable around 0.8886-0.8889. The notebook ends with a call to `<keras.callbacks.History at 0x7f1f0bf69d0>`.

Final\_Model (ResNet50+LSTM).ipynb ☆

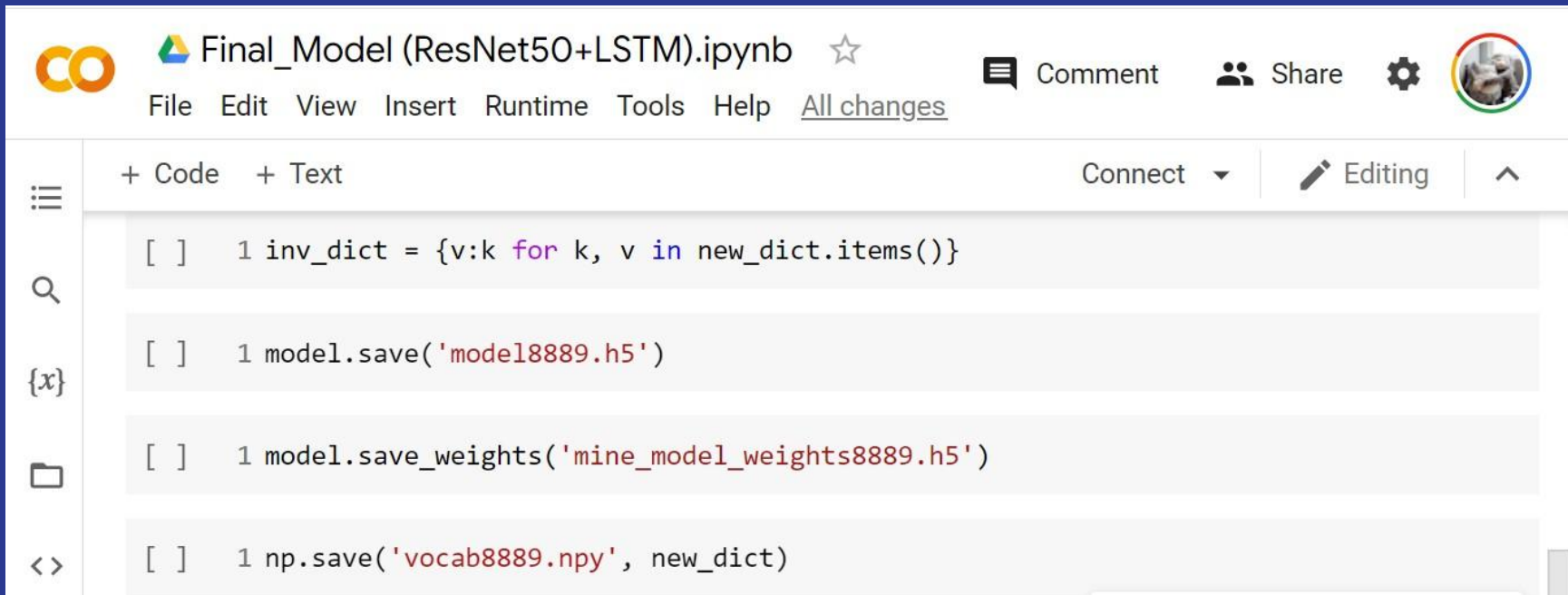
File Edit View Insert Runtime Tools Help [All changes saved](#)

Comment Share Settings

+ Code + Text Connect Editing

```
Epoch 192/200
[ ] 203/203 [=====] - 16s 77ms/step - loss: 0.3561 - accuracy: 0.8886
Epoch 193/200
203/203 [=====] - 16s 78ms/step - loss: 0.3614 - accuracy: 0.8868
Epoch 194/200
203/203 [=====] - 16s 78ms/step - loss: 0.3611 - accuracy: 0.8878
Epoch 195/200
203/203 [=====] - 16s 78ms/step - loss: 0.3570 - accuracy: 0.8875
Epoch 196/200
203/203 [=====] - 16s 78ms/step - loss: 0.3576 - accuracy: 0.8878
Epoch 197/200
203/203 [=====] - 16s 78ms/step - loss: 0.3544 - accuracy: 0.8884
Epoch 198/200
203/203 [=====] - 16s 78ms/step - loss: 0.3539 - accuracy: 0.8889
Epoch 199/200
203/203 [=====] - 16s 78ms/step - loss: 0.3523 - accuracy: 0.8887
Epoch 200/200
203/203 [=====] - 16s 78ms/step - loss: 0.3503 - accuracy: 0.8889
<keras.callbacks.History at 0x7f1f0bf69d0>
```

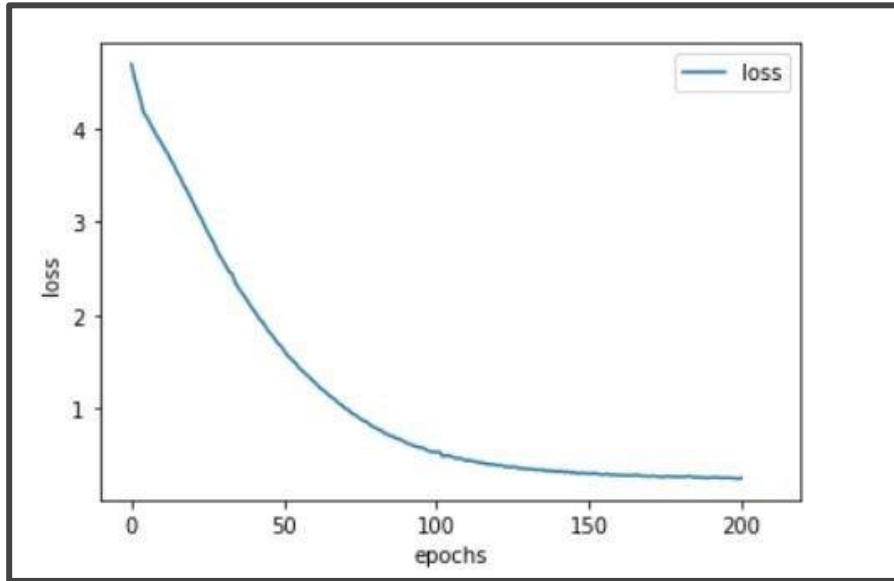
# Save Model & Vocabulary



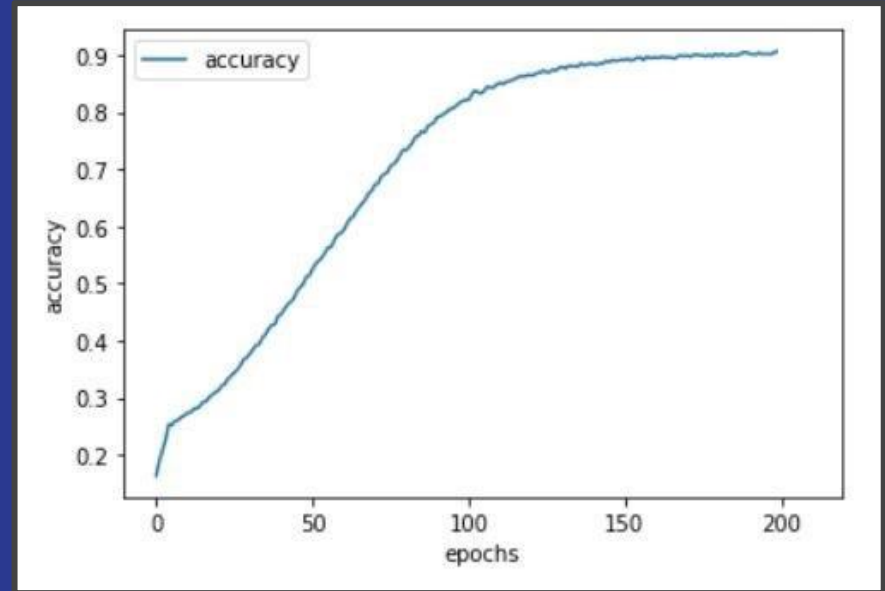
The screenshot shows a Jupyter Notebook interface with the title "Final\_Model (ResNet50+LSTM).ipynb". The interface includes a top bar with navigation links (File, Edit, View, Insert, Runtime, Tools, Help) and a "All changes" link. On the right, there are buttons for "Comment", "Share", and a settings gear, along with a user profile picture. The left sidebar contains icons for a menu, search, a variable "{x}", a folder, and a code editor icon "<>". The main area displays four lines of Python code, each preceded by a "[ ]" prompt:

```
[ ] 1 inv_dict = {v:k for k, v in new_dict.items()}  
  
[ ] 1 model.save('model8889.h5')  
  
[ ] 1 model.save_weights('mine_model_weights8889.h5')  
  
[ ] 1 np.save('vocab8889.npy', new_dict)
```

# Performance



# Measure

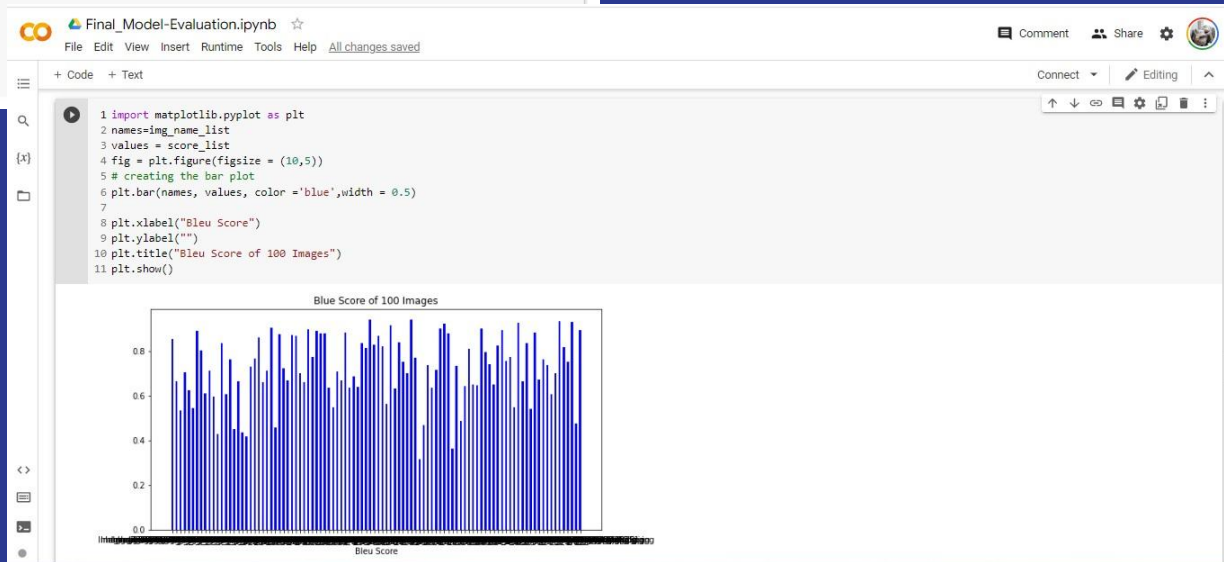


# BLEU Score (on 100 images)

```
Final_Model-Evaluation.ipynb ☆
File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

1 import matplotlib.pyplot as plt
2 names=img_name_list
3 values = score_list
4 fig = plt.figure(figsize = (10,5))
5 # creating the bar plot
6 plt.bar(names, values, color='blue',width = 0.5)
7
8 plt.xlabel("Bleu Score")
9 plt.ylabel("")
10 plt.title("Bleu Score of 100 Images")
11 plt.show()
```



# Average BLEU Score = 0.7238



Final\_Model-Evaluation.ipynb ☆

File Edit View Insert Runtime Tools Help [All changes saved](#)



Comment



Share



+ Code + Text

Connect ▾

Editing



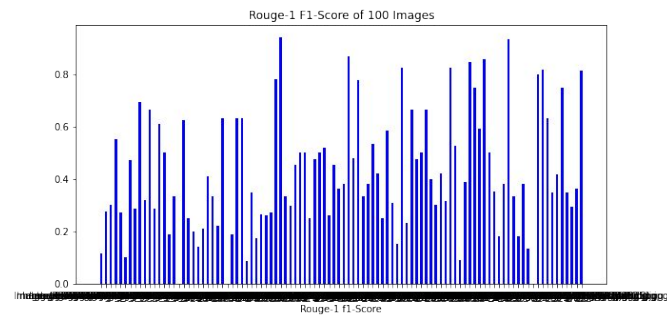
```
[ ] 1 img_name_list=[]
    2 score=0
    3 score_list=[]
    4 for i in range(100):
    5     img_name = images[i]
    6     reference_captions=captions_dict[img_name.split('/')[0]]
    7     bleu_score = sentence_bleu(reference_captions, cap_list[i])
    8     score_list.append(bleu_score)
    9     img_name_list.append(img_name)
   10     score=score+bleu_score
```

```
[ ] 1 avg_bleu_score=score/100
    2 print("Average Bleu Score : "+str(avg_bleu_score))
```

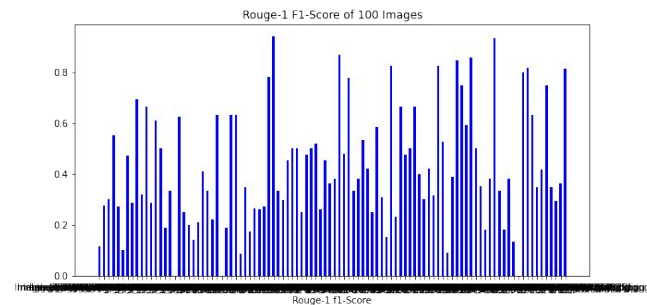
Average Bleu Score : 0.7237759639289125

# ROUGE Score

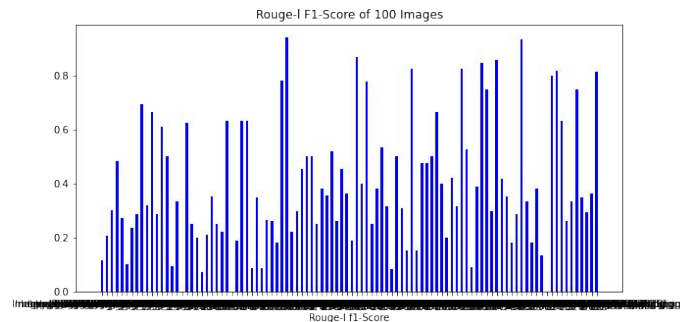
Rouge-1 F1-Score



Rouge-2 F1-Score




Rouge-l F1-Score



# Average ROUGE Score

 Final\_Model-Evaluation.ipynb ☆

File Edit View Insert Runtime Tools Help [All changes saved](#)

Comment Share ⚙️ 

+ Code + Text

Connect Editing ^

```
[ ] 1 rogue_1_s=0
    2 for i in range(100):
    3     rogue_1_s=rogue_1_s+rouge_1_f[i]
    4 print(rogue_1_s)
```

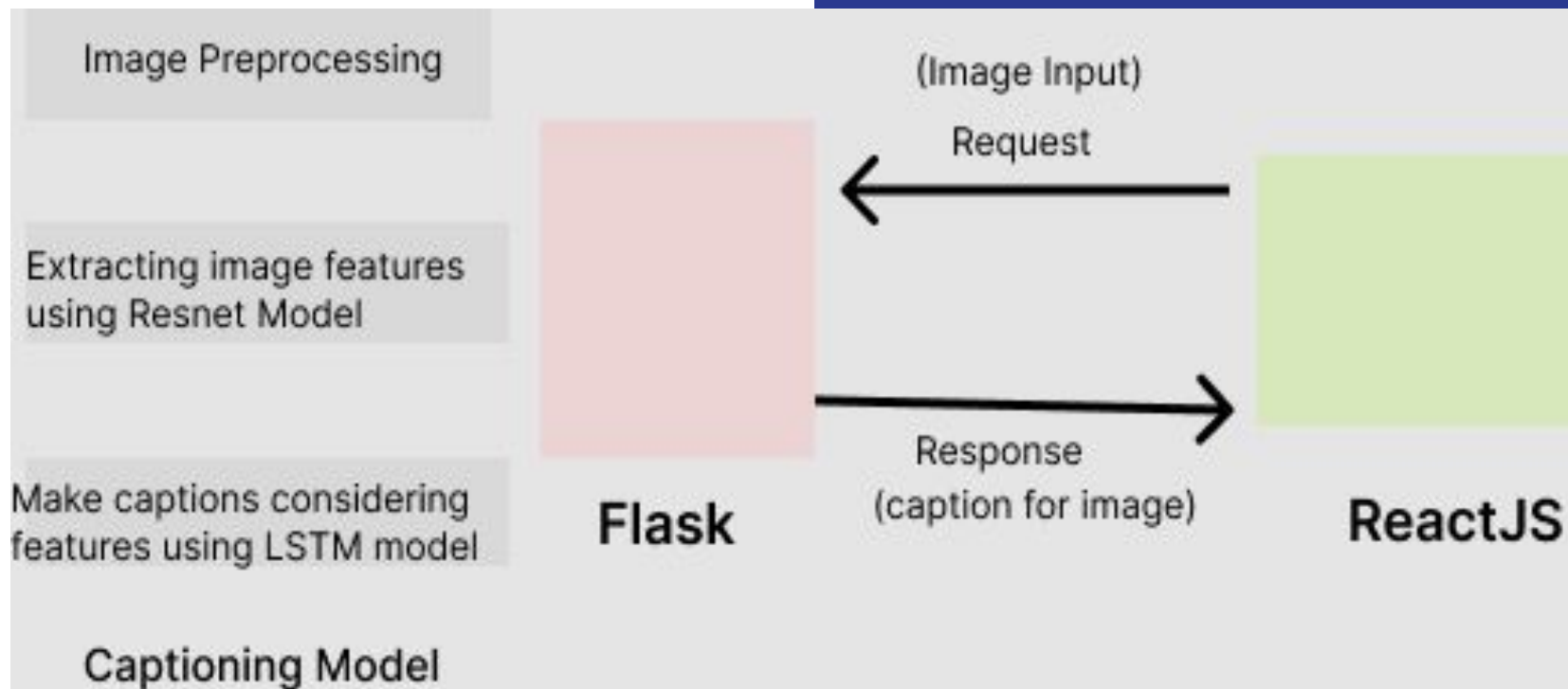
42.605771931299316

```
[ ] 1 avg_rouge_1=sum(rouge_1_f)/100
    2 avg_rouge_2=sum(rouge_2_f)/100
    3 avg_rouge_l=sum(rouge_l_f)/100
```

```
[ ] 1 print("Average Rouge-1 F1-score : " + str(avg_rouge_1))
    2 print("Average Rouge-2 F1-score : " + str(avg_rouge_2))
    3 print("Average Rouge-l F1-score : " + str(avg_rouge_l))
```

Average Rouge-1 F1-score : 0.42605771931299313  
Average Rouge-2 F1-score : 0.21421628278383217  
Average Rouge-l F1-score : 0.3967502175885946

# WEB APPLICATION





# CONCLUSION & FUTURE

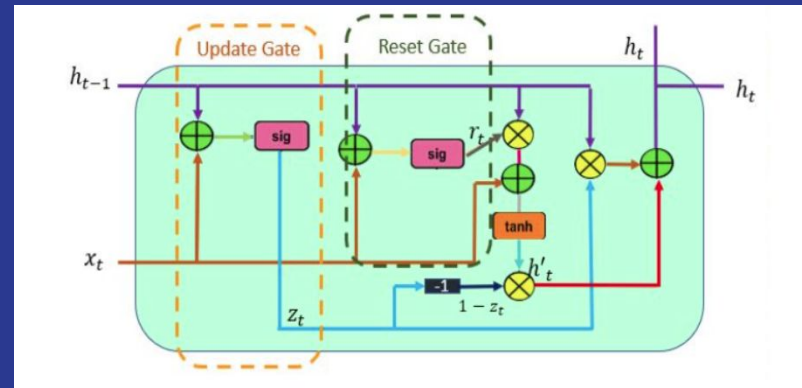
Estimated time for processing  
and result

Can we still improve it?

---

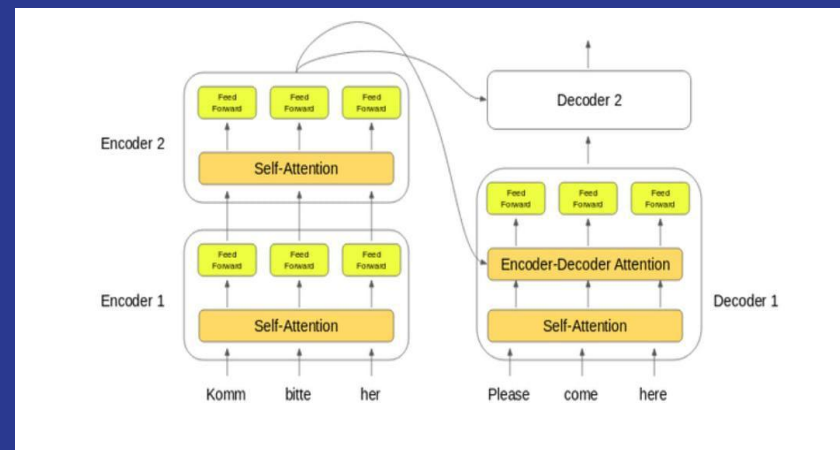
## GRU(Gated Recurrent Units) for captioning

- More efficient computation wise compared to LSTM
- uses less training parameter and therefore uses less memory and executes faster (29.3% faster) than LSTM



## Transformers

- solve sequence-to-sequence tasks while handling long-range dependencies
- relies entirely on self-attention mechanism
- Parallel processing
- Way more accurate and faster training time



# GITHUB REPOSITORY:

<https://github.com/saumyapanda17/image-captioning-dell>

# Thank You!

---