SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

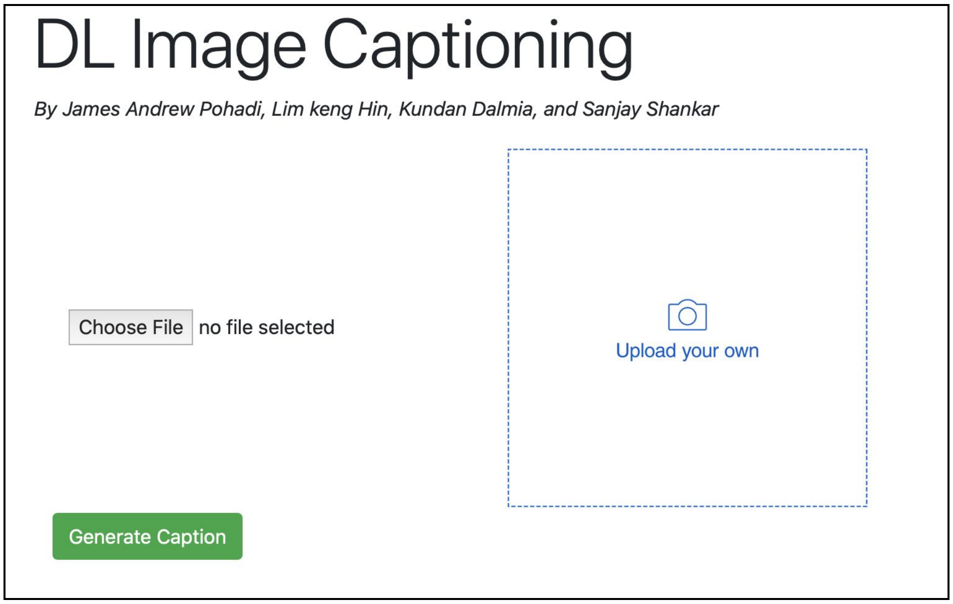
50.039: Theory and Practice of Deep Learning

Final Project Report: Image Captioning

Done by:
James Andrew Pohadi (1002899)
Kundan Dalmia (1002332)
Lim Kheng Hin (1002656)
Sanjay Shankar (1003044)

1. GUI

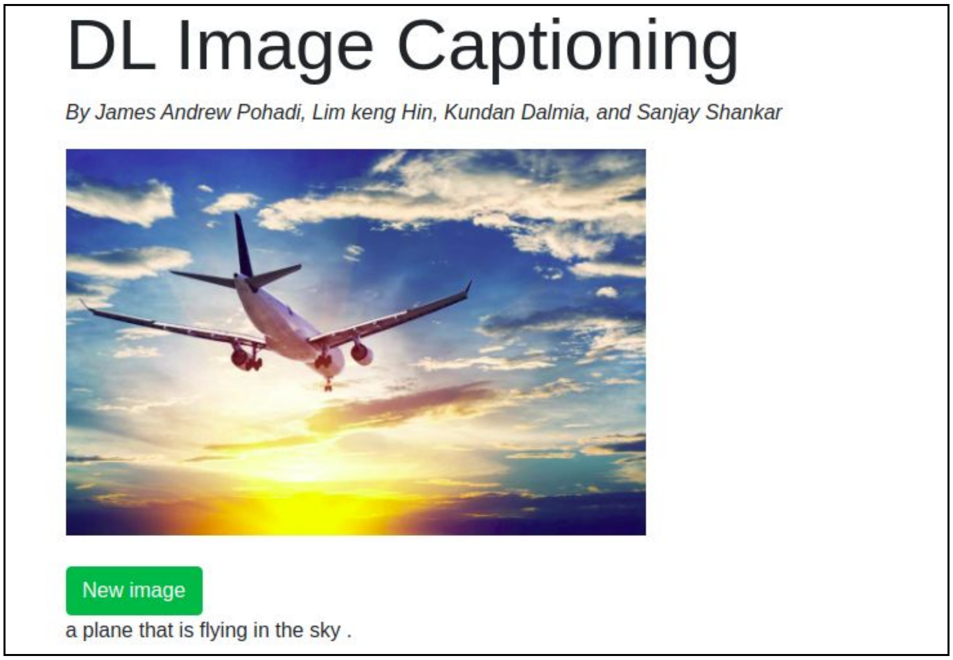
The team made use of Flask for the development of the front end to view the image captioning in action algorithm in action. A screenshot of the interface (without any image uploaded yet) is shown below:



DL Image Captioning

By James Andrew Pohadi, Lim keng Hin, Kundan Dalmia, and Sanjay Shankar

A screenshot of the interface (with image uploaded and relevant caption) is shown below:



DL Image Captioning

By James Andrew Pohadi, Lim keng Hin, Kundan Dalmia, and Sanjay Shankar

a plane that is flying in the sky .

2. Method

❖ Word Level modelling

- ➢ We chose to use word level modelling for our model, as we believe that word level modelling is more suitable in our case compared to character level modelling because in character level modelling, we need to learn how to create a word before we learn how to create a meaningful sentence which corresponds to an image. Therefore, character level modelling might require more parameters to store information on how to generate a word in addition to generate a sentence.

- ➢ Since we are experimenting on pre-trained weights from Glove6B - 200d, which uses words without **stemming**, we relied on 'using the words as they come' for our model to generate words in the caption. This makes the process of training and generating sequences easier.

❖ Dataset Used

- ➢ **Training**
  In order to train our model, we used the coco2014/train2014 dataset.

- ➢ **Validation**
  We used 1000 validation data sampled randomly from 40504 samples in coco2014/val2014 dataset. We use this strategy for evaluation because it is to long to evaluate our model with the full val dataset

- **Testing**

  Since there is no test dataset provided, we sampled 1000 data from 40504 samples in coco2014/val2014 dataset. We ensured that the test dataset doesn't overlap with the validation dataset.

Development

EfficientNet

- We decided to use **EfficientNet** for our Encoder class, as it superpasses state-of-the-art accuracy with up to 10x better efficiency (smaller and faster).

- After comparing EfficientNets with other existing CNN models on ImageNet, AI researchers at Google confirmed that their model achieved both higher accuracy and better efficiency over existing CNNs, by reducing parameter size and FLOPS by an order of magnitude. Hence, we decided to use it for our project as our main objective was to use a model which yields in maximum accuracy with minimal computation cost.

- The fact that we reduced the Encoder's size from 235 MB to 17.3 MB by using **EfficientNet**, without it having affected our training performance convinced us to go ahead with this as our encoder.

LSTM

- LSTM is a recurrent neural network that is often used in the sequence modelling task such as sequence generation and translation task. LSTM doesn't suffer from vanishing gradient descent unlike Vanila RNN.
- Thus, we used LSTM for our decoder class.

Connected cell and hidden

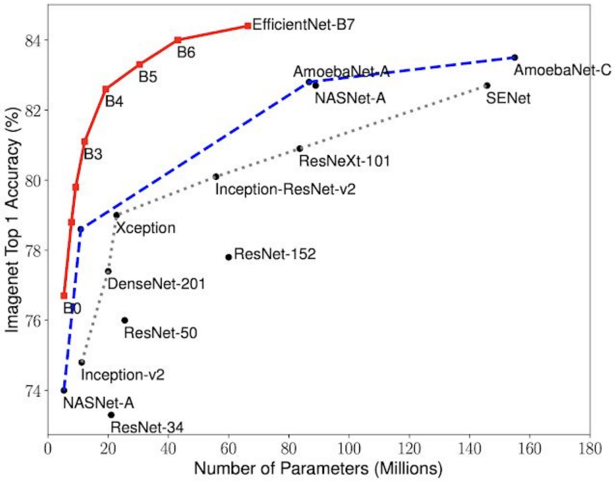
- We connected the output from the encoder to the hidden state and cell state of LSTM because we want to preserve the information of the output from the encoder throughout the sequence generation.

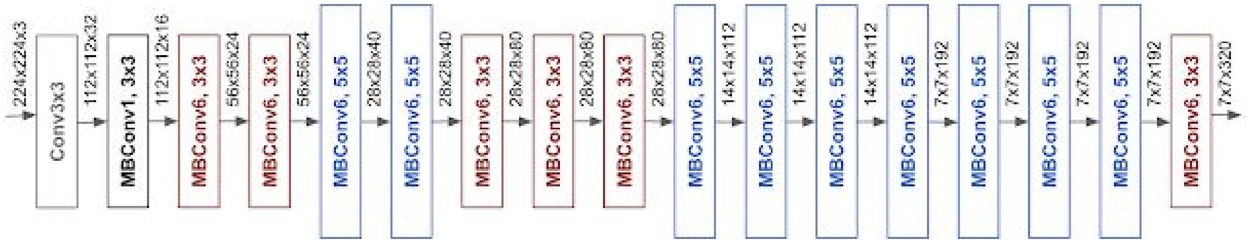
3. Experiments

The process of our model development was iterative. We started by using the code in https://github.com/yunjey/pytorch-tutorial/tree/master/tutorials/03-advanced/image_captioning. We saw that the model for the code above was quite large in size - 235 MB for encoder and 36.9 MB for decoder. Hence, in order to make the model more lightweight as well as ensure a high accuracy, we decided to perform experiments which will be shown in this subsection.

3.1. Experiments with EfficientNet

EfficientNet is a recent CNN model that was found by engineers from Google AI. It uses a new Compound coefficient based method to scale up CNNs in a structured manner. *This compound scaling method consistently improves model accuracy and efficiency for scaling up existing models such as MobileNet (+1.4% imagenet accuracy), and ResNet (+0.7%), compared to conventional scaling methods.*





The main reason we chose EfficientNet is because it has high accuracy and better efficiency than existing CNN models while reducing computation cost, battery usage and also training and inference speeds.

	Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	# Encoder Params	# Decoder Params
B	glove.200d	0.695	0.495	0.375	0.246	58,705,988	8,560,900
C	Efficientnetb0-glove.200d	0.684	0.478	0.331	0.232	4,306,614	8,560,900

As we can see from the table above, the performance of our model on BLEU score decreases a bit when we change the backbone to Efficientnet-b0. However, the number of parameters in the encoder decreases by more than 14x. Since we want to reduce number of parameters, we went forward with using efficientnet.

3.2. Experiments with Glove embedding

As we need to use word embedding for our task, we experiment with pre trained word embedding available online, Glove, by using the weights that is available in glove to

We used Glove embeddings with 6 Billion words with each word being a 200 dimensional vector (Glove6B - 200d). Initially, each word was a 256 dimensional vector, however we changed it to a 200 dimensional vector. Following is the result of our experiments

	Method	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
A	Default	0.709	0.501	0.353	0.253
B	glove.200d	0.695	0.495	0.375	0.246

As we can see, our experiment shows that there isn't much difference between using pre-trained word embedding or training embedding from scratch. Therefore, we decided not to use word embedding in our decoder.

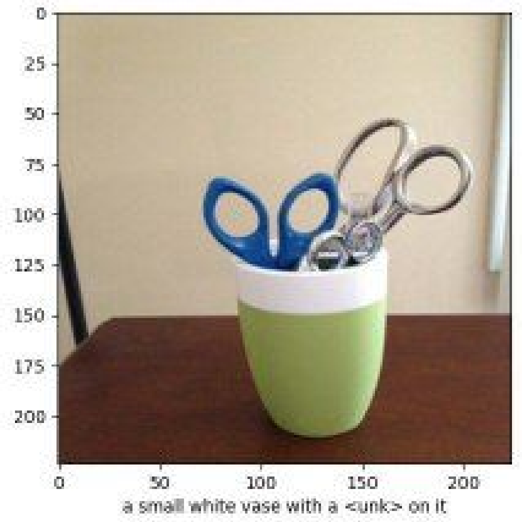
3.3. Experiments with word threshold

We conducted experiments of using different thresholds to ignore a word, ie, if the word occurrence in the dataset is less than a threshold, then we will replace the word with <unk>. We want to find a threshold which is high enough so that each word can have enough training samples while the <unk> shouldn't be too much, so it won't appear in caption generation.

We experimented on 3 different thresholds and following were the results:

- Threshold 4, total number of vocabulary 9956: With a threshold of 4, the total vocabulary is very huge. This causes the decoder model to have a lot of parameters to learn the mapping between hidden vectors to vocabularies.
- Threshold 10, total number of vocabulary 6336: The size of vocabulary is not too huge and is able to generate sequences without <unk>.

- Threshold 20, total number of vocabulary 4530: The size of vocabulary is very small, but it suffers from bad prediction since it often generates <unk> in a sequence. An example is shown below:



From our experiments on different thresholds, we decided to use threshold = 10.

3.4. Experiments with LSTM hidden size

We also tried different hidden sizes to see what the impact to the BLEU score is. The result is as seen in the table below:

	Method	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
B	Efficientnetb0-glove.200d	0.684	0.478	0.331	0.232
C	Efficientnetb0-glove.200d-hidden256	0.694	0.481	0.333	0.232
E	Efficientnetb0-glove.200d-hidden200-connected_cell	0.690	0.486	0.340	0.239
G	Efficientnetb0-glove.200d-hidden512-connected_cell	0.694	0.487	0.339	0.237

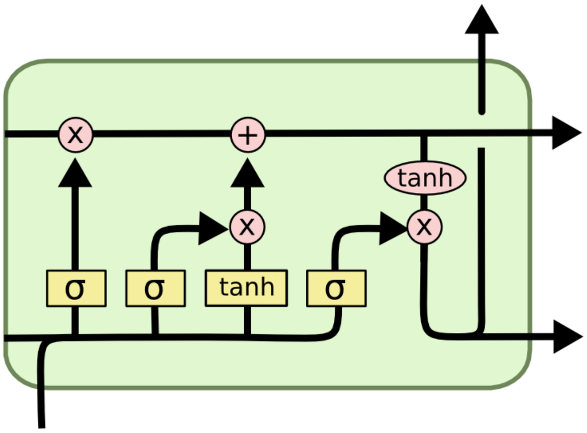
As we can see, there is not much difference in terms of BLEU score performance when we use higher hidden size. Therefore, for this project, we decided to use a hidden size of 256 for our model.

3.5. Experiments with Connected Cell and Connected Hidden

Usually the output of the encoder in the image caption model will go to the decoder via input. However, we believe that this is not ideal because the output from the encoder is very important and we shouldn't connect it only to the input because some information that is useful for propagation might be lost. Furthermore, connecting the encoder output to the input means that the dimension of the encoder must be equal to the dimension of the embedding. Therefore, we proposed to connect the output of decoder directly to the hidden state which we called **connected_hidden** and connected the output of decoder directly to the cell state which we called **connected_cell**. The table below shows this:

	Method	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
A	Efficientnetb0-glove.200d-hidden256	0.694	0.481	0.333	0.232
B	Efficientnetb0-glove.200d-hidden200-connected_cell	0.690	0.486	0.340	0.239
C	Efficientnetb0-glove.200d-hidden200-connected_hidden	0.680	0.473	0.326	0.227

From the result above, we can see that **connected_cell** gives better performance than **connected_hidden**. It make sense because hidden state used to create different gate, so, it doesn't really need to preserve the information from the start unlike the cell state



In our project, we decided to combine **connected_cell** and **connected_hidden**, so our output of the encoder will go to the first hidden state and cell state.

3.5. Summary of Experiments

In summary, the approach we took was in the same order as the table (from A to H):

The table below shows the performance of our experimented model on BLEU score as well as number of encoder and decoder parameters.

	Method	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**# Encoder Params**	**# Decoder Params**
A	Default **(yunjey)**	0.709	0.501	0.353	0.253	58,820,956	9,233,124
B	glove.200d	0.695	0.495	0.375	0.246	58,705,988	8,560,900
C	Efficientnetb0-glove.200d	0.684	0.478	0.331	0.232	4,306,614	8,560,900
D	Efficientnetb0-glove.200d-hidden256	0.694	0.481	0.333	0.232	4,306,614	5,018,884
E	Efficientnetb0-glove.200d-hidden200-connected_cell	0.690	0.486	0.340	0.239	4,306,614	4,313,956
F	Efficientnetb0-glove.200d-hidden200-connected_hidden	0.680	0.473	0.326	0.227	4,306,614	4,313,956

G	Efficientnetb0-glove.200d-hidden512-connected_cell	0.694	0.487	0.339	0.237	4,707,534	8,560,900
H	Efficientnetb1-hidden256-connected_cell_hidden-vocab10-scale_down3 **(ours)**	0.684	0.476	0.331	0.233	6,904,262	2,164,480

From the table above, we can see that our final model uses a very small number of parameters but is still able to achieve almost similar performance to our initial model.

The Inference time on **GPU: GRID V100DX-8Q** was as follows:

Model	**Inference Time**
Encoder (Resnet152)	0.019s
Encoder (Efficientnet-b0)	0.011s
Encoder (Efficientnet-b1)	0.017s
Decoder (hidden 512)	0.007s
Decoder (hidden 256)	0.006s
DecoderScaleDown	0.006s

From the table above, we see that there is not much difference between the inference time of decoder using hidden size of 512 and hidden size of 256.

4. Model Architecture

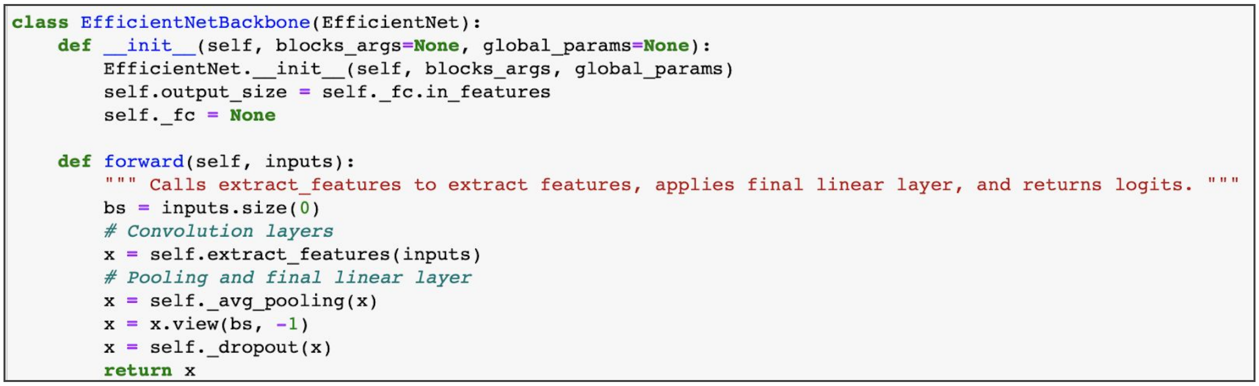
The model we used can be broadly classified into the following subcomponents:

- Encoder
- Decoder

Encoder:

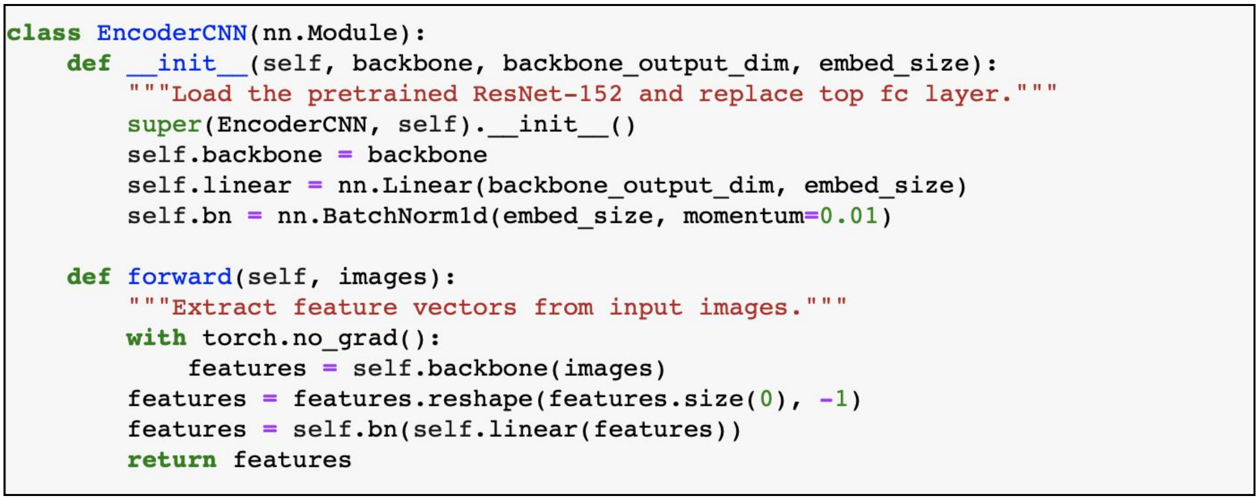
For our encoder, we decided to use EfficientNet, as it superpasses state-of-the-art accuracy with up to 10x better efficiency (smaller and faster). After comparing EfficientNets with other existing CNN models on ImageNet, AI researchers at Google confirmed that their model achieved both higher accuracy and better efficiency over existing CNNs, by reducing parameter size and FLOPS by an order of magnitude. Hence, we decided to use it for our project as our main objective was to use a model which yields in maximum accuracy with minimal computation cost.

The fact that we reduced the Encoder's size from 235 MB to 17.3 MB by using EfficientNet, without it having affected our training performance convinced us to go ahead with this as our encoder.



```python
class EfficientNetBackbone(EfficientNet):
    def __init__(self, blocks_args=None, global_params=None):
        EfficientNet.__init__(self, blocks_args, global_params)
        self.output_size = self._fc.in_features
        self._fc = None

    def forward(self, inputs):
        """ Calls extract_features to extract features, applies final linear layer, and returns logits. """
        bs = inputs.size(0)
        # Convolution layers
        x = self.extract_features(inputs)
        # Pooling and final linear layer
        x = self._avg_pooling(x)
        x = x.view(bs, -1)
        x = self._dropout(x)
        return x
```

Shown below is the screenshot of the **EfficientNet backbone** and our **Encoder** class:



```python
class EncoderCNN(nn.Module):
    def __init__(self, backbone, backbone_output_dim, embed_size):
        """Load the pretrained ResNet-152 and replace top fc layer."""
        super(EncoderCNN, self).__init__()
        self.backbone = backbone
        self.linear = nn.Linear(backbone_output_dim, embed_size)
        self.bn = nn.BatchNorm1d(embed_size, momentum=0.01)

    def forward(self, images):
        """Extract feature vectors from input images."""
        with torch.no_grad():
            features = self.backbone(images)
        features = features.reshape(features.size(0), -1)
        features = self.bn(self.linear(features))
        return features
```

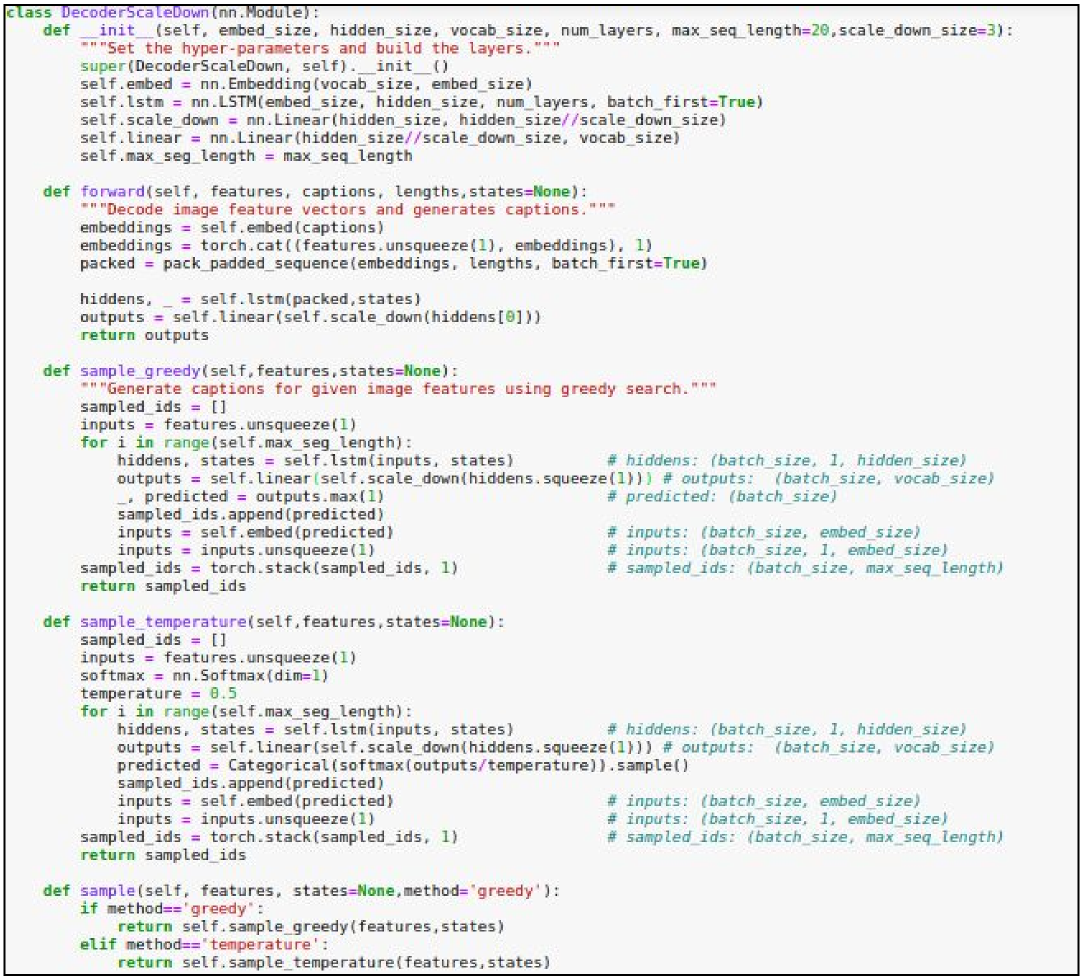
Decoder:

As we already know, the decoder is responsible for looking at the encoded image and generating the respective caption. Hence, it is a valuable component that requires a lot of thought in order to achieve high accuracy (or BLEU score in this case).

Since we are dealing with sequential data, we will need to make use of Recurrent Neural Networks (RNN) here. Thus, for the decoder in this project, we made use of LSTM. The output from the encoded image is fed into the decoder as the first hidden state and cell state **(connected_cell_hidden)**. The sequence generation starts from Consequently, we use the predicted word in order to predict the next word in the sequence until it generates <end>.

In addition to that, to reduce the parameters of our decoder, we scale down our mapping between the hidden output to the word vocab by 3 with a linear layer before another linear layer that maps the scaled-down output to the vocab.

Shown below is our **DecoderScaleDown** class:

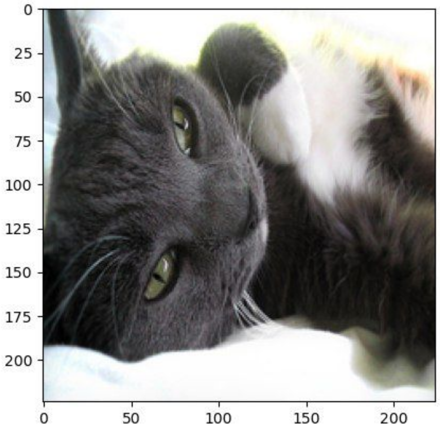


```
class DecoderScaleDown(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size, num_layers, max_seq_length=20,scale_down_size=3):
        """Set the hyper-parameters and build the layers."""
        super(DecoderScaleDown, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers, batch_first=True)
        self.scale_down = nn.Linear(hidden_size, hidden_size//scale_down_size)
        self.linear = nn.Linear(hidden_size//scale_down_size, vocab_size)
        self.max_seg_length = max_seq_length

    def forward(self, features, captions, lengths,states=None):
        """Decode image feature vectors and generates captions."""
        embeddings = self.embed(captions)
        embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
        packed = pack_padded_sequence(embeddings, lengths, batch_first=True)

        hiddens, _ = self.lstm(packed,states)
        outputs = self.linear(self.scale_down(hiddens[0]))
        return outputs

    def sample_greedy(self,features,states=None):
        """Generate captions for given image features using greedy search."""
        sampled_ids = []
        inputs = features.unsqueeze(1)
        for i in range(self.max_seg_length):
            hiddens, states = self.lstm(inputs, states)          # hiddens: (batch_size, 1, hidden_size)
            outputs = self.linear(self.scale_down(hiddens.squeeze(1))) # outputs:  (batch_size, vocab_size)
            _, predicted = outputs.max(1)                        # predicted: (batch_size)
            sampled_ids.append(predicted)
            inputs = self.embed(predicted)                       # inputs: (batch_size, embed_size)
            inputs = inputs.unsqueeze(1)                         # inputs: (batch_size, 1, embed_size)
        sampled_ids = torch.stack(sampled_ids, 1)                # sampled_ids: (batch_size, max_seq_length)
        return sampled_ids

    def sample_temperature(self,features,states=None):
        sampled_ids = []
        inputs = features.unsqueeze(1)
        softmax = nn.Softmax(dim=1)
        temperature = 0.5
        for i in range(self.max_seg_length):
            hiddens, states = self.lstm(inputs, states)          # hiddens: (batch_size, 1, hidden_size)
            outputs = self.linear(self.scale_down(hiddens.squeeze(1))) # outputs:  (batch_size, vocab_size)
            predicted = Categorical(softmax(outputs/temperature)).sample()
            sampled_ids.append(predicted)
            inputs = self.embed(predicted)                       # inputs: (batch_size, embed_size)
            inputs = inputs.unsqueeze(1)                         # inputs: (batch_size, 1, embed_size)
        sampled_ids = torch.stack(sampled_ids, 1)                # sampled_ids: (batch_size, max_seq_length)
        return sampled_ids

    def sample(self, features, states=None,method='greedy'):
        if method=='greedy':
            return self.sample_greedy(features,states)
        elif method=='temperature':
            return self.sample_temperature(features,states)
```

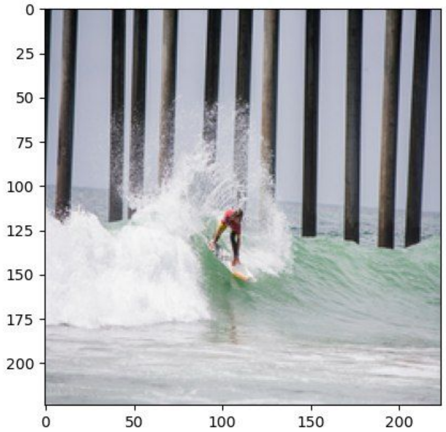
5. Results

4.1. Qualitative results

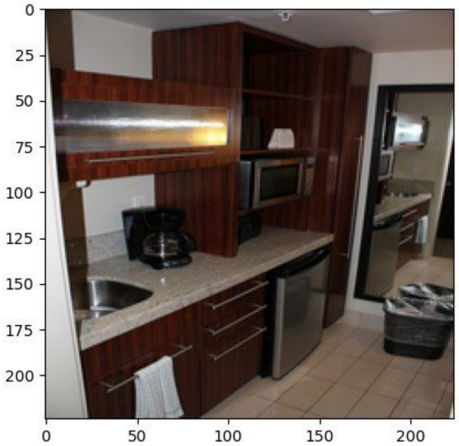
Our model is definitely not on par with the state-of-the-art systems due to our inexperience in language modelling, time and physical constraints. However, we have managed to come up with a lightweight image captioning model with performance on par with the model that we start with. Here are some of our model best predictions:



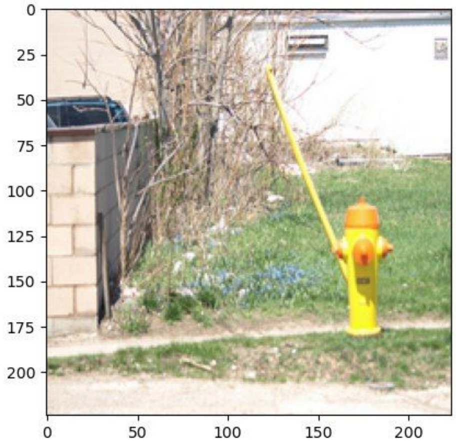
a cat is laying down on a bed



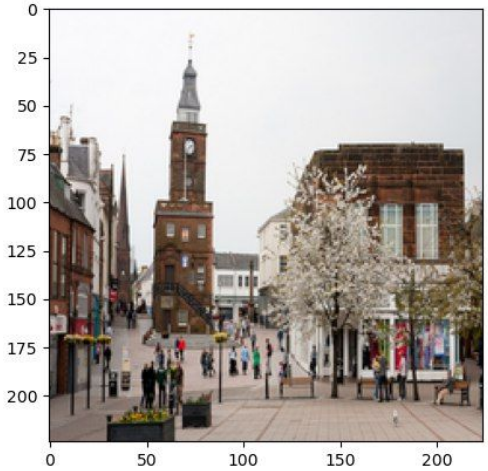
a man riding a wave on top of a surfboard



a kitchen with a stove , sink , and cabinets

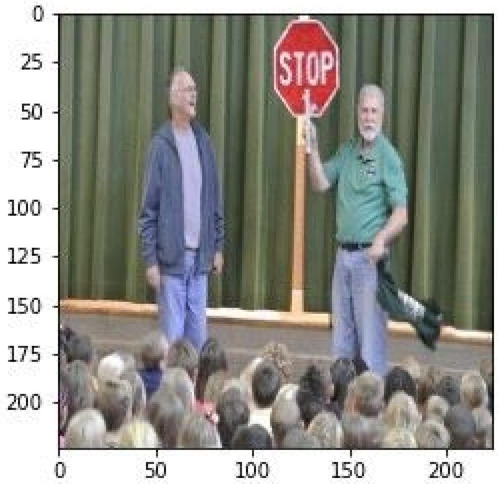


a yellow fire hydrant sitting on the side of a road

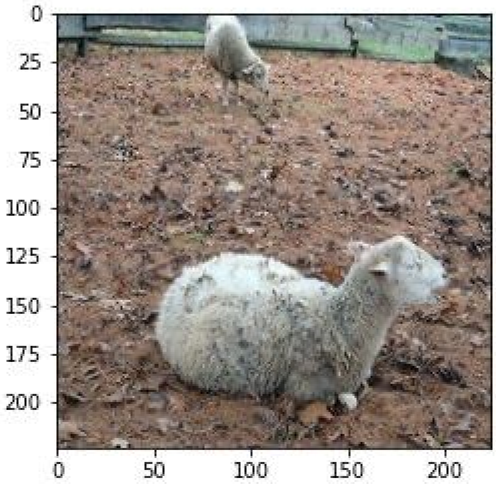


a tall building with a clock on the top of it

Here are 2 of the funniest predictions of our model:



a group of people are playing a game of soccer .



a large bird is standing in the water .

4.2. Comparisons with state-of-the-art

Comparison with state of the art
State of the art: https://competitions.codalab.org/competitions/3221#results

Methods	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
Yingwei.Pan	0.819	0.669	0.524	0.403
Meshed-Memory - Transformer	0.816	0.664	0.518	0.397
KingSoft_AILAB	0.819	0.670	0.527	0.406
CaptionLight **(ours)**	0.684	0.476	0.331	0.233

From the table above, we can see that our model is still far from the state of the art. However, we manage to achieve a lightweight model that is able to generate decent captions.

How to reproduce

Set up:

1. Create the ‘./dataset’ folder with ‘coco2014’ inside
2. git clone https://github.com/jamesandrewpohadi/LightCaption
3. cd LightCaption
4. bash setup.sh
5. Download our trained models and put it under “./models” in “./LightCaption”: https://drive.google.com/file/d/1mrbgxJ5niFvBlRjxWhYwVw2_nBFdDtZA/view?usp=sharing

Training:

```
python3 train.py
```

Testing:

```
python3 test.py
```

6. Contributions

James Andrew Pohadi

- Training and evaluation (BLEU score, param size)

Kundan Dalmia

- Incorporating Efficientnet-b0, Glove.6B

Lim Keng Hin

- Experimenting with different parameters

Sanjay Shankar

- Connected hidden and connected cell