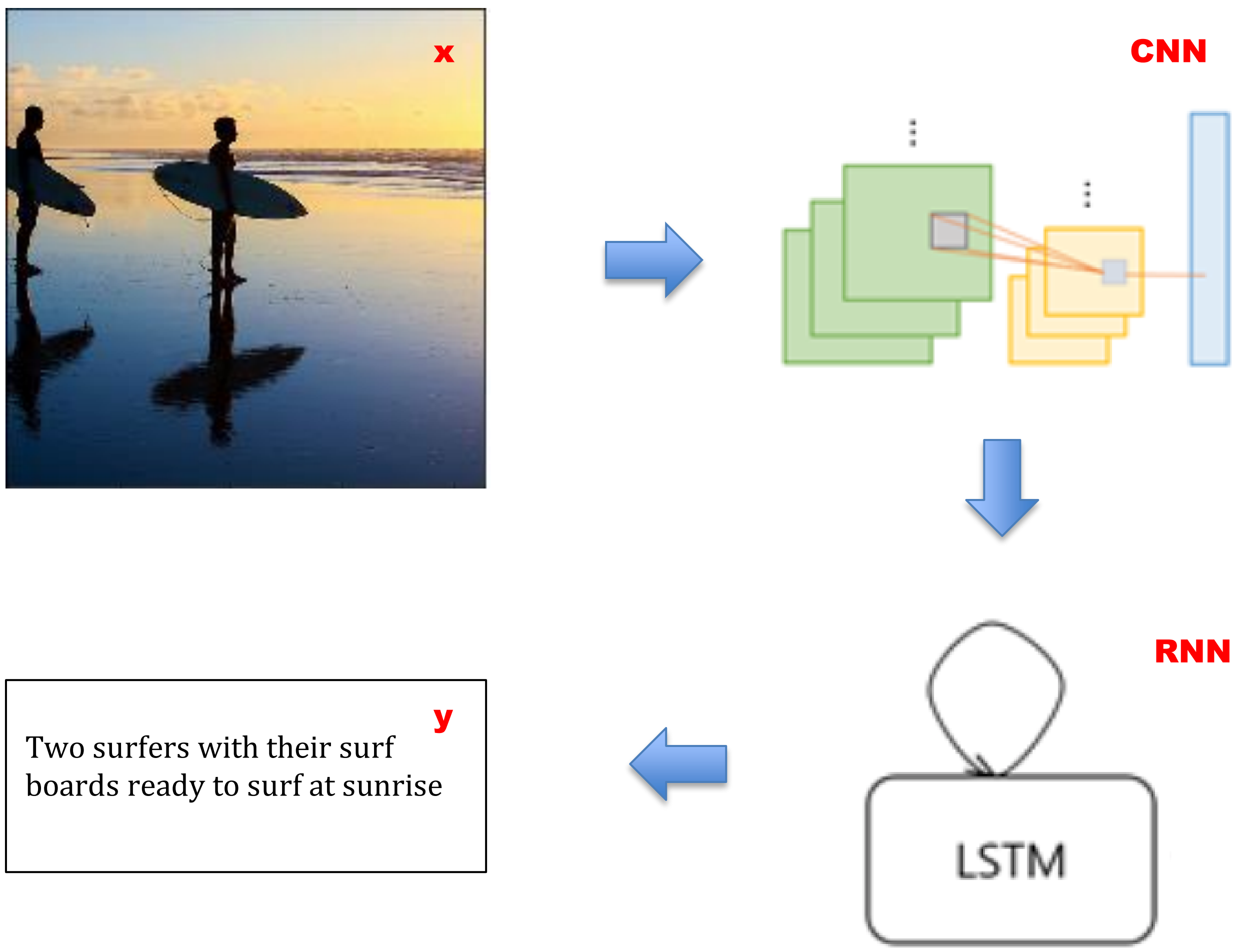
Automatic Caption Generation for Images



Capstone Project for Postgraduate Data Science Expert [2017-2018]
University: U-tad
Student: Jurjen Feitsma
Tutor: Prof. Manuel Rubio-Sánchez

Table of Contents

List of Figures

1 Introduction

For my final project I wanted to dive further into the field of deep learning. Neural networks need a lot of data and computing power (and people use GPU's instead of CPU's). So as we see an increase in these two factors we see neural networks are being used more every day and with impressive results [13].

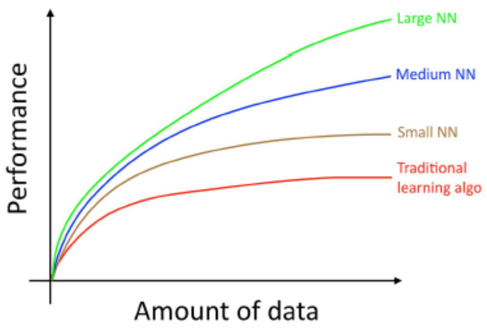


Fig 1: As the amount of data increases, the performance of deep neural networks increases

In this project I have applied my knowledge about neural networks for images and texts solving the task of generating descriptions for real world images.
Automatic caption generation for images is an important problem in artificial intelligence. Caption generation shows a relationship between image information and natural language processing. It includes not only understanding important information about an image through feature extraction, but also generating natural language by summarizing key information on images. After the 2014 MS COCO Image Captioning Challenge lots of successful research results in caption generation for images have been published. One of these papers is "Show and Tell: A Neural Image Caption Generator"[7].

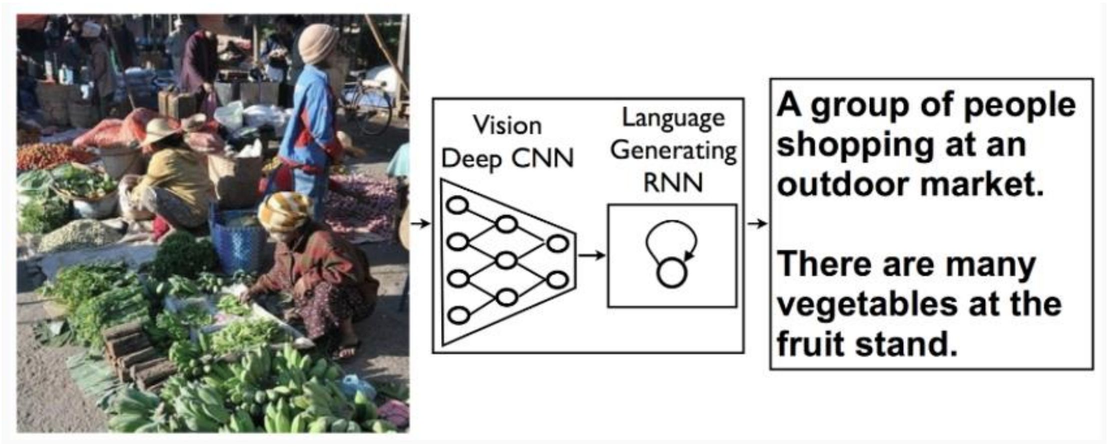

Vision Deep CNN
Language Generating RNN
A group of people shopping at an outdoor market.
There are many vegetables at the fruit stand.

Fig 2: Show and Tell: A Neural Image Caption Generator

This project is based on the capstone project of the first course [14] of an advanced machine learning specialization offered on the Coursera platform.
To prepare for this project I studied Convolutional Neural Networks, used for images, and Recurrent Neural Networks, used for text.
The code for this project can be found in the accompanying notebook [1].

The reader is assumed to have basic knowledge of machine learning, as well as basic knowledge of Python, Numpy, TensorFlow and Keras.

2 Dataset

Microsoft COCO [8] is a large image dataset designed for object detection, segmentation, and caption generation. The COCO 2014 dataset contains 80 classes, 80k training images and 40k validation images. The dataset contains images of complex everyday scenes containing common objects in their natural context:

- train2014.zip contains 82783 images for training
- val2014.zip contains 40504 images for validation
- captions_train-val2014.zip contains 5 different captions (descriptions) for each image.

3 Feed Forward Neural Network (FFNN)

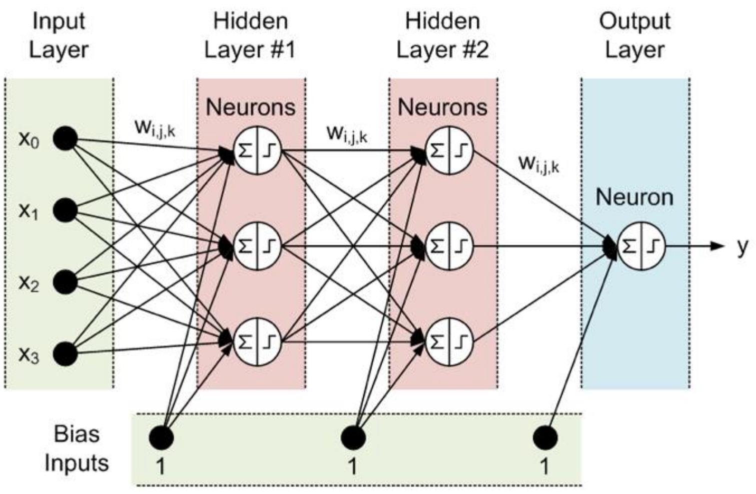


Fig 3: A Feed Forward Neural Network

The network depicted in fig. 3 is a 3-layer neural network with 2 hidden layers.
Each layer consists of a linear transformation followed by a squashing nonlinearity.
A fully-connected layer (or dense layer) has its neurons (or nodes) connected to all the other neurons in the previous layer. We can calculate the values *a* of the neurons in layer *l* as follows:

$$a^{\langle l \rangle} = f\left(W \cdot a^{\langle l-1 \rangle} + b\right)$$

where *f* is the nonlinear activation function and the matrix *W* contains the *weights* for this layer.

3.1 Nonlinear activation

Some nonlinear activation layers usually follow fully-connected layers. Nonlinearities enhance the expression capability of the network. Some examples of nonlinear activation functions are Sigmoid (σ), tanh and ReLU. ReLU is the most commonly used activation function in CNN.
In this notebook we use ELU as an activation function, a strong alternative to ReLU.
Unlike ReLU, ELU can produce negative outputs:

$$a = \begin{cases} \alpha(e^{z} - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$

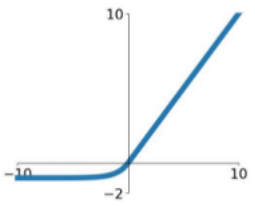


Fig 4: ELU activation function

3.2 Softmax layer

Softmax assigns decimal probabilities to each class in a multi-class classification problem. Those decimal probabilities must add up to 1.0.

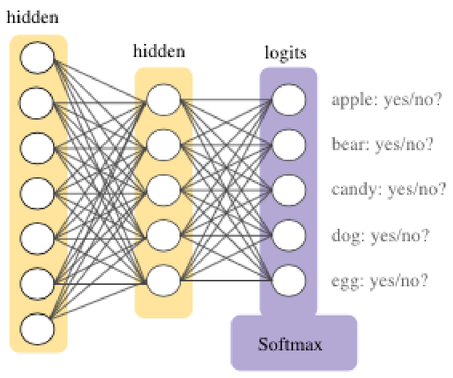


Fig 5: Softmax layer

Softmax is implemented through a neural network layer just before the output layer. The Softmax layer must have the same number of nodes as the output layer.

3.3 Learning (the weights)

We are want to learn an unknown function that maps an input *x* (in this project an image) to an output *y* (in this project a caption). Learning the function means learning these weights. To measure how well the function approximates the ground truth we use a loss function (in this project *cross entropy*). The goal is to minimize this loss function: to find the combination of weights that minimize the loss. During training we update the weights on every iteration so that the loss decreases.
The procedure (algorithm) to update the weights is *gradient descent*: using the gradient of the loss function to make a step in the right direction (the direction of the minimum). It can be shown that cross entropy loss is a convex function, which means we cannot get stuck in a local minimum.
To calculate the gradients we use *back propagation*. We calculate the partial derivative of the loss function for every weight in every layer, using the chain rule to calculate the weights in one layer based on the de derivatives of weights in other layers. Calculating derivatives comes down to multiplying and adding numbers.

4 TensorFlow

Since I use TensorFlow in this project, in this section I will mention a few things that are necessary to understand the code used in the notebook.

TensorFlow uses a *dataflow graph* to represent the computation in terms of the dependencies between individual operations.

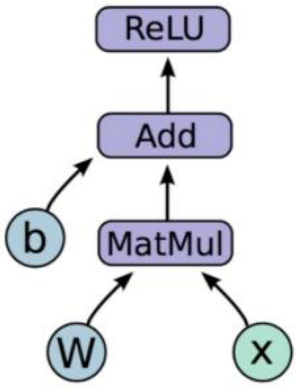


Fig 6: A dataflow graph in TensorFlow

This leads to a low-level programming model in which you first define the dataflow graph and then create a TensorFlow *session* to run parts of the graph:

```
session.run(fetches, feed_dict,  …)
```

The `fetches` argument may be a single graph element, or a list of graph elements.

The optional `feed_dict` argument allows the caller to override the value of tensors in the graph.

TF uses Stochastic Gradient Descent [2] to minimize the loss function. The *gradients* needed for backpropagation are calculated automatically.

The function `tf.nn.sparse_softmax_cross_entropy_with_logits` measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). This op expects unscaled logits, since it performs a softmax on logits internally for efficiency.

The function `tf.nn.dynamic_rnn(cell, inputs, …)` creates a recurrent neural network and Performs fully dynamic unrolling of `inputs`.

See [4] for an example of a simple FFNN with TensorFlow.

4.1 Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. It was developed with a focus on enabling fast experimentation. See [3] for examples with Keras.

5 Convolutional Neural Network (CNN)

In deep learning, a Convolutional Neural Network (CNN) is the representative model for image feature extraction. A CNN is a set of filters (feature maps) containing weights, which are applied in groups (one group of filters per layer), one group after another. Applying a filter is a combination of multiplying and adding numbers.
A schematic representation of a CNN is shown in fig. 7.

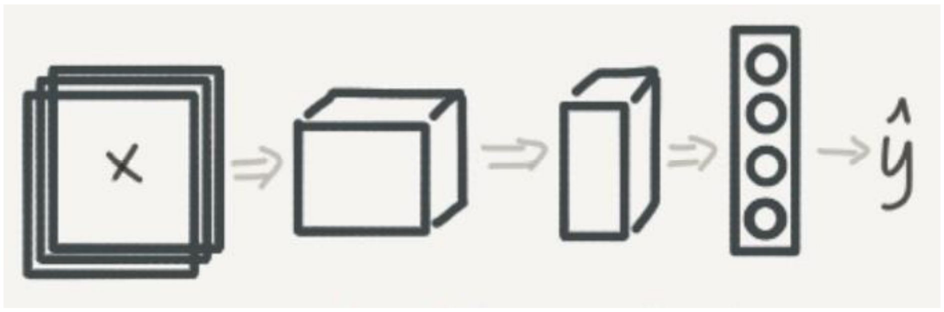


Fig 7: A Convolutional Neural Network [17]

The input *x* is a 3d-array: height, width, input_channels (usually 3 for RBG). This input data is then fed through *convolutional layers* instead of normal layers, where not all nodes are connected to all nodes (we use parameter sharing). These convolutional layers also tend to shrink as they become deeper (more to the end of the network), mostly by easily divisible factors of the input. Powers of two are very commonly used here, as they can be divided cleanly and completely by definition: 32, 16, 8, 4, 2, 1. Besides these convolutional layers there are *pooling layers*. And at the end of a CNN there are a few fully-connected layers.

5.1 Convolutional layer

This layer uses the *convolution* operation to extract (low-level and high-level) features and to discover local correlation and spatial invariance. A convolution is a dot product of a kernel (or *filter*) and a patch of an image (local receptive field) of the same size [14]:

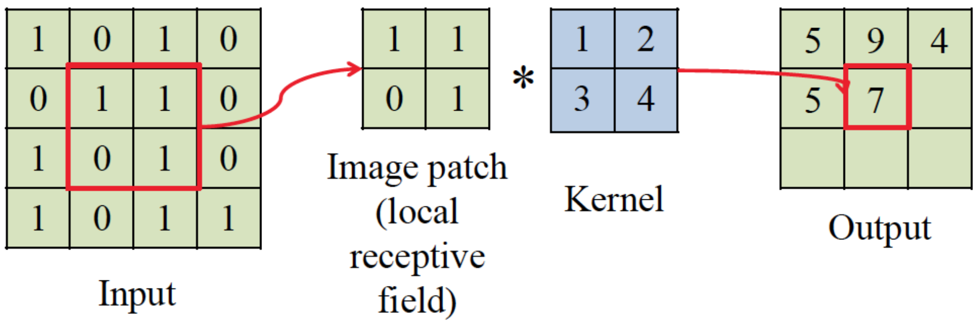


Fig 8: Convolutional layer

Because the filter is smaller than the layer it operates on, we use fewer weights than in a fully connected layer. We stride the filter across the input. Every convolution of an image patch and the kernel results in one number in the output.

In fig. 8 only one channel is shown. But if the input has 3 channels then the filter has 3 channels as well. If we chooses to apply 2 filters than the output has 2 channels.

5.2 Pooling layer

It is common to periodically insert a pooling layer in-between successive convolutional layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX or AVERAGE operation. An example of Max pooling is shown in fig. 9:

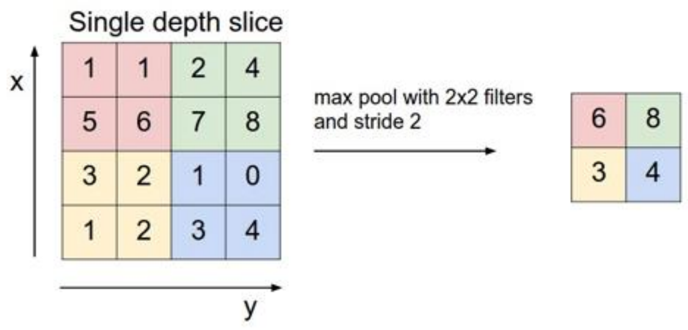


Fig 9: Pooling layer

See [5] for an example on the building blocks of an CNN using TF.
See [6] for an example of a simple RNN using Keras.

6 InceptionV3

Since 2014 very deep convolutional networks started to become mainstream, yielding substantial gains in various benchmarks. The *Inception model* [9] is particularly exciting because it's been battle-tested, delivering world-class results in the widely acknowledged ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It's also designed to be computationally efficient, using 12x fewer parameters than other competitors, allowing Inception to be used on less-powerful systems.

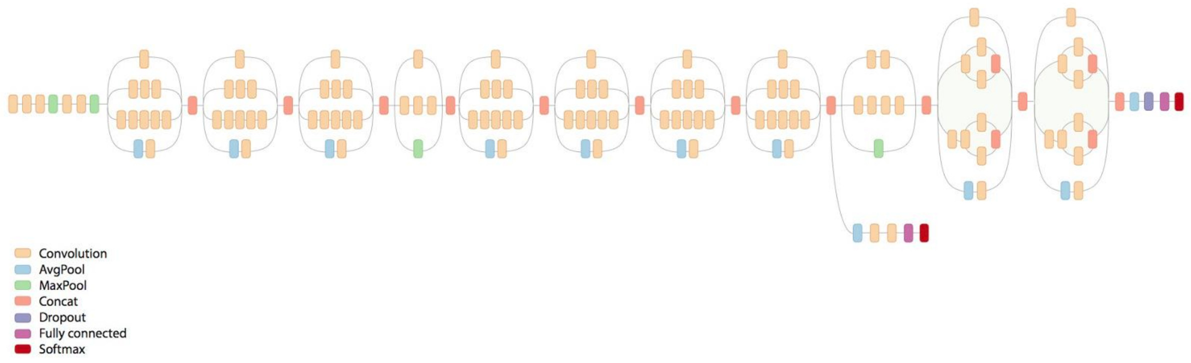


Fig 10: InceptionV3 network

7 Recurrent Neural Network (RNN)

RNNs are widely used for processing time series data like speech or text.
Predicting the next word in a caption is a sequence-modeling problem.
To model sequences, we need:

1. to deal with variable-length sequences
2. to maintain sequence order
3. to keep track of long-term dependencies
4. to share parameters across the sequence

In fig. 11 we see two representations of a RNN cell. If we unroll the representation on the left, we get the representation on the right.

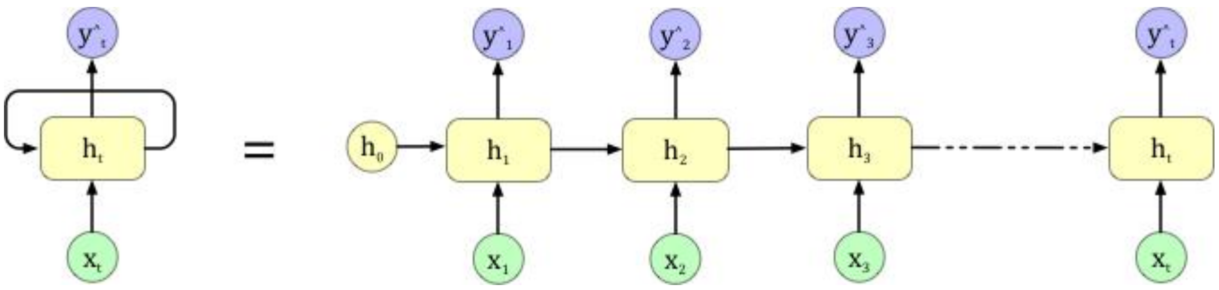


Fig 11: A Recurrent Neural Network

Every time step we feed in a word (x_t) and predict (the probability for) the next word:

$$\begin{aligned}
h_t &= \tanh\left(W_{hh} \cdot h_{t-1} + W_{hx} \cdot x_t + b_h\right) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_h\right) \\
&= \tanh\left(W \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_h\right) \\
y^{\wedge}_t &= \sigma\left(W_{yh} \cdot h_t + b_y\right)
\end{aligned}$$

where h_t is the 'hidden state' at time step t and $y^{\wedge}_t$ are the predicted probabilities. The weights W are shared for all the time steps.

A problem of RNNs is the vanishing gradients during backpropagation, because derivatives expand into a long product across the time steps. To solve this problem we can use a more complicated cell architecture, like LSTM or GRU.

8 Long short-term memory (LSTM)

A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. Each of the three gates can be thought of as a "conventional" artificial neuron, as in an FFNN: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.
In an LSTM we have two hidden states, one of which we still call hidden state h_t and the other one we call cell state c_t that serves as the cell memory, see fig. 12.

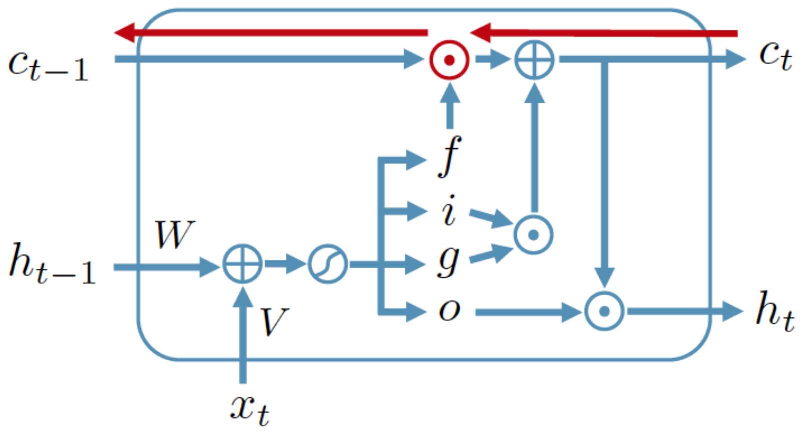


Fig 12: An LSTM unit

So how does an LSTM work?

- Forget the irrelevant part of previous state (via forgetgate *f*)
- Selectively update the cell state values (via inputgate *i* and updating values *g*)
- Output certain parts of cell state (outputgate *o* lets parts of *c* through)

$$\begin{pmatrix} f \\ i \\ g \\ o \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \tanh \\ \sigma \end{pmatrix} \left(W \cdot \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b \right)$$

$$c_t = f * c_{t-1} + i * g$$

$$h_t = o * \tanh(c_t)$$

Using a LSTM cell helps to prevent vanishing gradients, because backpropagation runs smoothly across the cell state. The only multiplication is with the forgetgate at every time step. To prevent the gradients from vanishing *f* is usually initialized with 1's.

```
# Initialize the parameters for an LSTM cell in TensorFlow:
        __init__(…,  forget_bias=1.0, …)
```

9 Image Captioning Model

For our model we concatenate a CNN and a RNN. We use the **CNN** as an **encoder** and the **RNN** as a **decoder**.

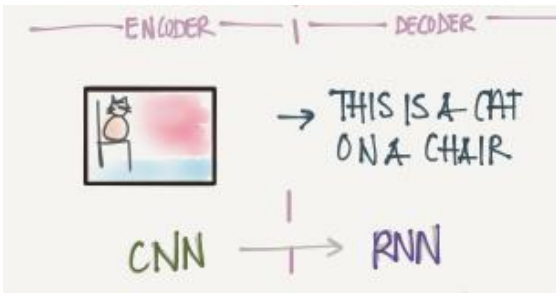


Fig 13: Using a CNN as an encoder and a RNN as a decoder [17]

The architecture of the model, tuned to be trainable on CPU, is shown in fig. 14.

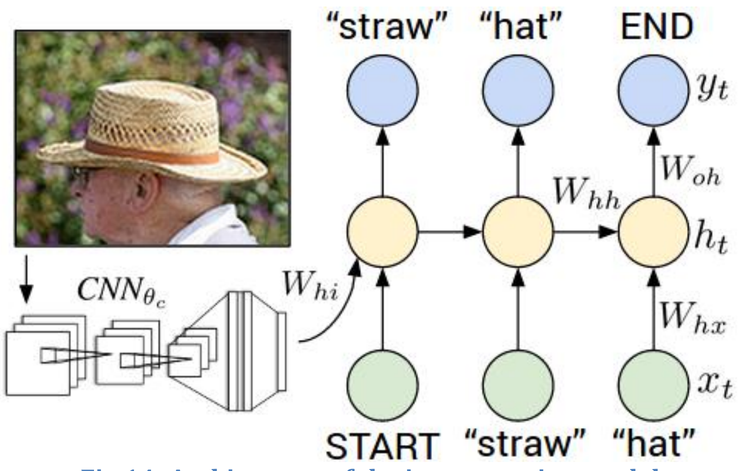


Fig 14: Architecture of the image caption model

Both the image and the caption text are mapped to the same space: the hidden state of the RNN. In the notebook this space has 300 dimensions (LSTM_UNITS).

We use the pretrained InceptionV3 model as our CNN and keep the weights fixed.
We take off the output layer and use the last hidden layer as an embedding vector for our input image. Normally, the CNN's last layer is used in a final Softmax among known classes of objects, assigning a probability that each object might be in the image. But if we remove that final layer, we can instead feed the CNN's rich encoding of the image into a RNN designed to produce captions. This turns out to work quite well.

We feed in the (embedded) image only once as h_0. The RNN has to remember the relevant information. So things we learn about the images (by putting it through the CNN) have to transfer through the RNN.

10 Decoder Loss

The decoder maps inputs (the last generated word in the caption) to probabilistic predictions (for the next word in the caption). We train our model by incrementally adjusting the model's parameters (*W*) so that our predictions get closer and closer to ground-truth probabilities (y_t). This ground truth probability is 1 for the next word in the training caption and 0 for all the other word in the vocabulary.

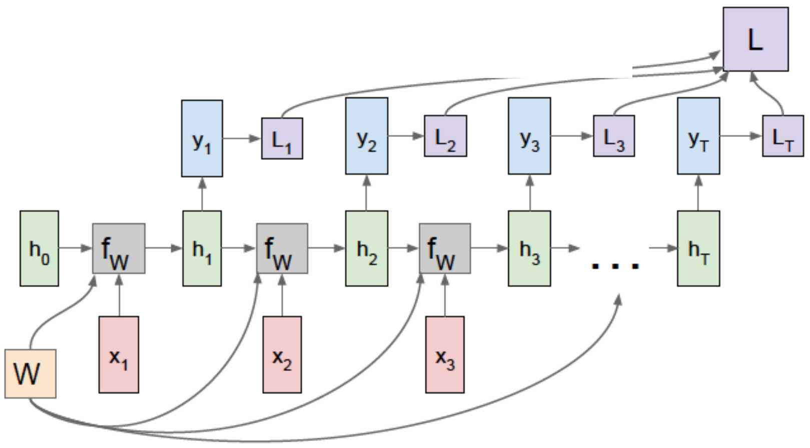


Fig 15: The computational graph for training the RNN

We use cross entropy loss as our loss function, also called negative log-likelihood. This means we take the logarithm of the predicted probability and then multiply by -1 to make it a minimization problem.

$$L_t = -\log(\hat{y}_t)$$

```
xent = tf.nn.sparse_softmax_cross_entropy_with_logits(
                labels=flat_ground_truth,
                logits=flat_token_logits)
```

"tf.nn.sparse_softmax_cross_entropy_with_logits" expects unscaled logits, since it performs a softmax on logits internally for efficiency. The logits are derived from the hidden states of the LSTM cells.
The flat ground truth is a vector of integers (the index of the word in the vocabulary).

For each mini-bath we take the average of the losses for all the words (not counting the PAD tokens) in the whole batch of captions:

```
loss = tf.reduce_mean(tf.boolean_mask(xent, flat_loss_mask))
```

11 How do the images flow through the model?

First all training and validation images are *encoded* by feeding them into the pretrained InceptionV3 CNN (with the last softmax layer taken off):

```
        model = keras.applications.InceptionV3(include_top=False)
```

11.1 Cropping and resizing

But the images have different sizes. And the InceptionV3 model expects a fixed size of size 299x299x3. So we have to do some preprocessing of the images.
First we crop the images in the center. Then we resize to 299x299x3 and use this as the input to the encoder.

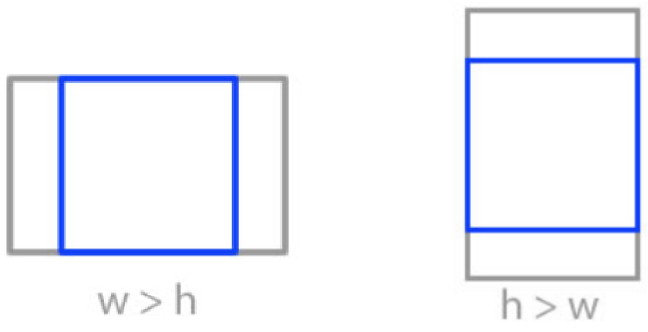


Fig 16: Cropping image in the center

11.2 Image embedding

The last hidden layer of InceptionV3 has 2048 units (neurons). This means after the encoding each image is represented as an embedding vector with 2048 elements.
To reduce the number of parameters we apply a dense layer that outputs a vector of 120 elements (IMG_EMBED_BOTTLENECK).
Note: To make the model as small as possible we use a bottleneck: we first reduce 2048 to 120 and then scale up to 300. You need 2048x120 + 120x300 parameters with bottleneck and 2048x300 without, which is 281k versus 614k parameters.

11.3 Initialize LSTM with image embedding

This is then connected with another dense layer to the initial hidden state (h_0) and the cell state (c_0) of the LSTM, which has 300 elements (LSTM_UNITS).

```
img_embed_bottleneck_to_h0 = L.Dense(LSTM_UNITS,
                              input_shape=(None, IMG_EMBED_BOTTLENECK),
                              activation='elu')
```

So both h_0 and c_0 are initialized with information from the image:

```
initial_state=tf.nn.rnn_cell.LSTMStateTuple(c0, h0)
```

12 How do the caption texts flow through the model?

The dataset contains 5 different captions for each image. The lengths of the captions vary.

12.1 Vocabulary

First we construct a vocabulary of all the tokens (words) that occur 5 times or more in the captions in the training set.

```
vocab = generate_vocabulary(train_captions)
```

This generates a Python dictionary with the tokens as keys and index integers as values. The vocab dictionary contains 4 special tokens: PAD (for padding), UNK (unknown, out of vocabulary), START (start of sentence) and END (end of sentence).
Our vocabulary contains 8769 tokens.

```
…,  'bacon': 500,  'bad': 501,  'badge': 502,  'badges': 503,  'badly':
504,  'badminton': 505,  'bag': 506,  'bagel': 507,  …
```

We also construct a vocab_inverse with the indices as keys and the words as values, which we will need later to translate predicted indices back to words.

```
…,  637: 'bbq',  638: 'be',  639: 'beach',  640: 'beached',  641:
'beachfront',  642: 'beacon',  643: 'bead',  644: 'beaded',  …
```

12.2 Index the captions

Next we replace the tokens in each caption by their index. For example:

```
A long dirt road going through a forest.
```

becomes:

```
2, 54, 4462, 2305, 6328, 3354, 7848, 54, 3107, 0
```

with 2 being the index for START and 0 being the index for END.

12.3 Padding the captions

During the training of the RNN network we train on badges of 64 images. For every image we randomly select 1 of the 5 available captions for this image.

```
for caption_list in indexed_captions[sample]:
        batch_captions.append(caption_list[np.random.randint(0,5)])
```

We want all the captions to be of the same length, which is the number of time steps in the RNN. In this notebook we use a length of 20 tokens. We accomplish this by padding the captions with the PAD-token (or removing tokens when the caption is too long).

```
batch_captions_matrix = batch_captions_to_matrix(batch_captions,
        vocab[PAD], max_len=max_len)
```

So a training sentence (translated back to tokens) would look like this:

```
#START# a living room with a sofa piano and large flat screen tv #END#
#PAD# #PAD# #PAD# #PAD# #PAD# #PAD#
```

12.4 Word embedding

We saw that each caption is padded and fed into the RNN as an array of 20 integers, the word indexes. The first layer in the RNN is an embedding layer. This layer is used to create a vector representation of the words in our sentence. We use an embedding vector with 100 elements (WORD_EMBED_SIZE):

```
word_embed = L.Embedding(len(vocab), WORD_EMBED_SIZE)
```

Embedding turns positive integers (indexes) into dense vectors of fixed size (in our case 100). Basically what happens is that the input *x* is mapped from an 8769-dimensional space (using one-hot-encoding) to a 100-dimensional space. Similar words end up 'close' to each other in this space. During training the network learns the weights for this mapping.

We embed all but the last token in the sentence (because the last token will not be used as input):

```
word_embeds = word_embed(sentences[:,:-1])
```

These embedded words are used to calculate the hidden states of the RNN:

```
hidden_states, _ = tf.nn.dynamic_rnn(lstm, word_embeds, …)
```

12.5 Word predictions

Remember we have 300 hidden states in the RNN model. We use a dense layer to reduce this size to 120 (LOGIT_BOTTLENECK), to reduce model complexity.
This is then connected with another dense layer to predict the next token in the caption:

```
token_logits = L.Dense(len(vocab),
        input_shape=(None, LOGIT_BOTTLENECK))
```

Note that we compute logits (so we don't use an activation function here). "`tf.nn.sparse_softmax_cross_entropy_with_logits`" calculates token probabilities internally.

12.6 Ground truth

To calculate the cross entropy loss we compare the predicted words with the ground truth, which are also word indexes.

```
flat_ground_truth = tf.reshape(sentences[:,1:], [-1])
```

Notice that we don't use the first token (START) here, because it's not part of the prediction.

13 Training the RNN

13.1 Training loop

We truncate long captions to speed up training and use a maximum length of 20 tokens. First we generate a random batch:

```
def generate_batch(images_embeddings, indexed_captions, batch_size,
                      max_len=None):
       …

       return {decoder.img_embeds: batch_image_embeddings,
               decoder.sentences: batch_captions_matrix}
```

This returns the dictionary we feed into the decoder:

```
train_loss += s.run([decoder.loss, train_step],
       generate_batch(train_img_embeds, train_captions_indexed, …))[0]

val_loss += s.run(decoder.loss,
       generate_batch(val_img_embeds, val_captions_indexed, …))
```

Each epoch consists of 1000 mini-badges. Each badge contains 64 training examples.

13.2 Learning Curve

After 12 epochs the learning curve looks like this:

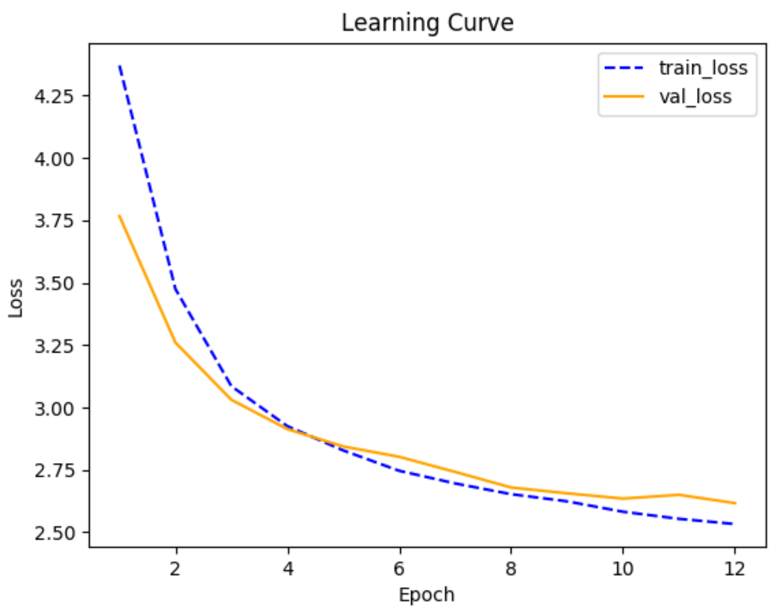


Fig 17: Learning Curve after 12 epochs

Our model has learned. Although the training loss keeps decreasing, the validation loss is almost horizontal. So this seems like a good point to stop the training.

Note1: The weights file has a size of 32,8 MB.
Note2: Training on CPU takes over an hour,
Training on GPU (Google Colab) takes 11 minutes.

13.3 Accuracy

It is not easy to automatically check whether the predicted caption is accurate. One thing we can do is count how many times the next predicted word is equal to the corresponding word in the ground truth.

```
sklearn.metrics.accuracy_score(logits.argmax(axis=1)[mask],
                                          truth[mask]))
```

For one random batch shown in the notebook this accuracy is 47%.

13.4 BLUE Score

Another possibility is using the BLUE Score [10], which stands for bilingual understudy evaluation, used to judge the quality of machine translation. Here quality is considered to be the correspondence between a machine's output and that of a human.
BLUE works best when comparing to corpuses of texts. By default, the `corpus_bleu()` score calculates the cumulative 4-gram BLEU score, also called BLEU-4. The weights for the BLEU-4 are 1/4 for each of the 1-gram, 2-gram, 3-gram and 4-gram scores:

```
corpus_bleu(references, hypothesis, weights=(0.25, 0.25, 0.25, 0.25))
```

Equally, BLEU-2 gives the cumulative bi-gram BLEU socre:

```
corpus_bleu(references, hypothesis, weights=(0.5, 0.5, 0, 0))
```

BLEU's output is always a number between 0 and 1, with 1 being hypothetical "perfect". In machine translation BLUE-4 scores over 0.3 generally reflect understandable translations, scores over 0.5 generally reflect good and fluent translations.

For the same random batch as described above we get the following scores:
BLEU-1: 0.74
BLEU-2: 0.62
BLEU-3: 0.57
BLEU-4: 0.48

Although this BLUE-4 score of 0.48 doesn't seem too bad, it doesn't necessarily mean the model generates acceptable captions. Let look at some examples:

Predicted:	a cat cat sitting on top table with a bathroom #END#
Truth:	a siamese cat sitting on a bed in a bedroom #END#

Both sentences seem similar, but 'a top table' isn't the same as 'a bed' and the cat is not in the 'bathroom', but in the 'bedroom'.

Predicted:	a person holding a black shirt holding at a cell phone #END#
Truth:	a man in a striped shirt staring at his cell phone #END#

A ‘person’ is less specific than a ‘man’, but it’s correct. This person is not ‘holding shirt’ but wearing a shirt. And ‘holding at a cell phone’ is grammatically incorrect.

In conclusion we can say that there is room for improvement, but the model has definitely learned something.

14 Final model

At training time, we had access to the ground-truth caption, so we fed ground-truth words as input to the RNN at each time step. Now we only have an image, which we use to initialize the LSTM:

```
img_embeds = encoder(input_images)

init_c = init_h = decoder.img_embed_bottleneck_to_h0(
                      decoder.img_embed_to_bottleneck(img_embeds))
init_lstm = tf.assign(lstm_c, init_c), tf.assign(lstm_h, init_h)
```

We use the word predicted at time *t* as the input for the LSTM cell at time *t+1*:

```
word_embed = decoder.word_embed(current_word)

new_c, new_h = decoder.lstm(word_embed,
tf.nn.rnn_cell.LSTMStateTuple(lstm_c, lstm_h))[1]
```

The first input is the START token and we iterate until we predict the END token.

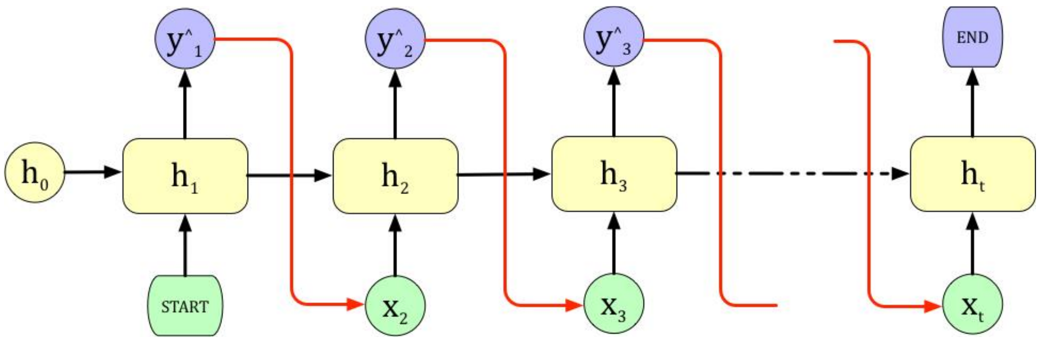


Fig 18: Final model for generating captions

Every time step we calculate for all the words in the vocabulary the probability that it will be the next word in the caption, given the input image and the words we have seen so far.

As an example consider the image shown in fig. 19.

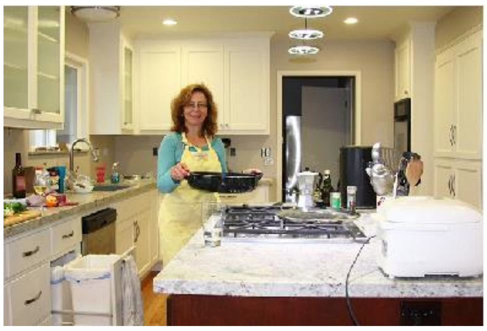


Fig 19: An image for which we want to generate a caption

Say we already predicted “a woman standing”. We then calculate:

$$P(\ldots \mid \text{<img>}, \text{“a woman standing”})$$

```
new_logits = decoder.token_logits(
                             decoder.token_logits_bottleneck(new_h))
new_probs = tf.nn.softmax(new_logits)
one_step = new_probs, tf.assign(lstm_c, new_c), tf.assign(lstm_h,new_h)
```

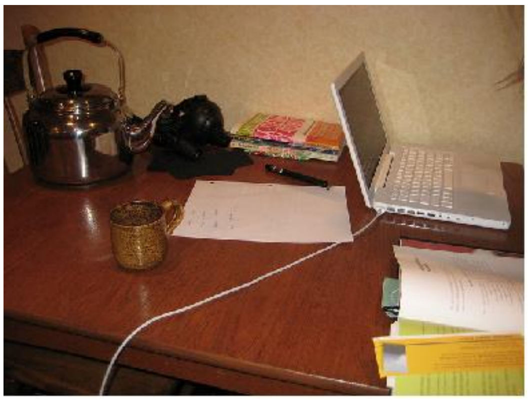
We choose the word the word that had the highest probability:

```
next_word = np.argmax(next_word_probs)
```

15 Generated Captions

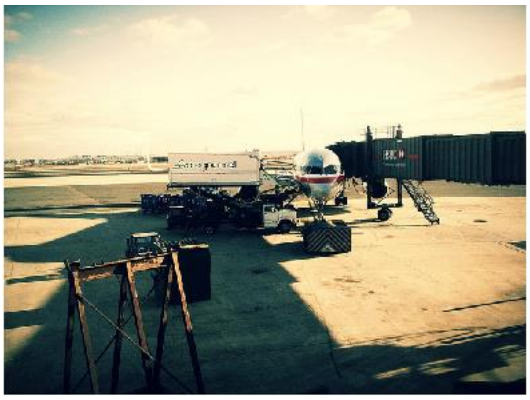
Since we have not tuned any hyperparameters, i.e. we haven't used our validation set to improve our model (only for early stopping), our model has only seen the training set. Let's look at some examples from the validation set where the model gives good results and some examples where the model fails.

15.1 Good results



“a laptop computer sitting on top of a table”

The model misses some objects on the table, but its output is correct.



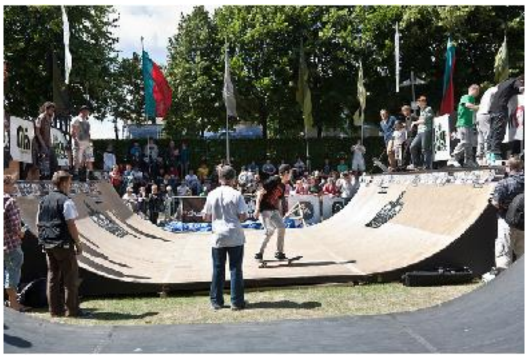
“a plane parked on the runway at an airport”

Ok, it's not the runway, but the model captures quite well what's going on in this image.

Fig 20: Examples where the model performs well

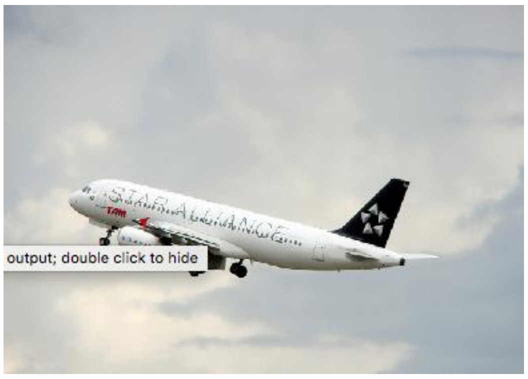
15.2 Almost right

Most captions fall in the category "almost right". There is some truth in the caption, but the model didn't nail it, or worse, didn't capture the most important part.



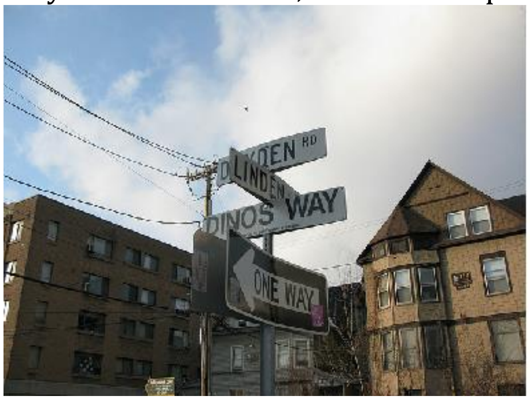
"a group of people sitting on a skateboard in the air"

The model captures that there is a group of people skateboarding, but the group is not sitting together on a skateboard and they are not in the air, but on a ramp.



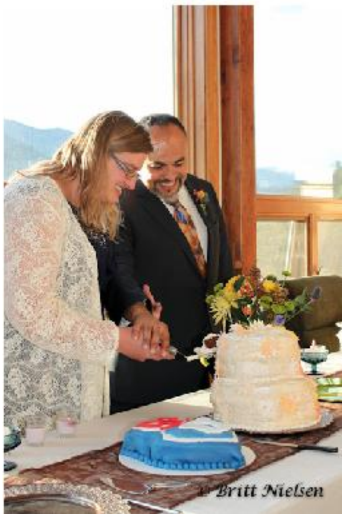
"a plane flying through the air with a plane in the background"

The model gets the most important part, but there is no plane in the background.



"a street sign that reads #UNK# st and #UNK#"

The model gets the most important part, the street signs, but it cannot read what is on the sign, so it puts 'unknown'.

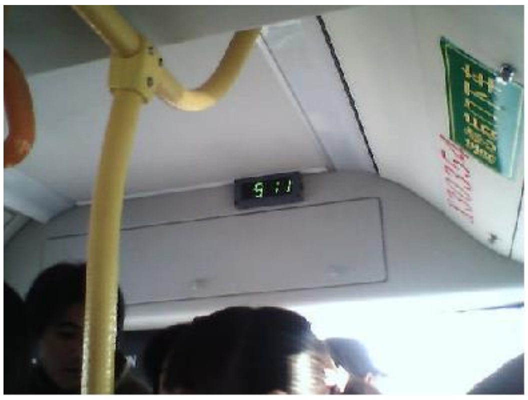


"a group of people sitting at a table with a cake"

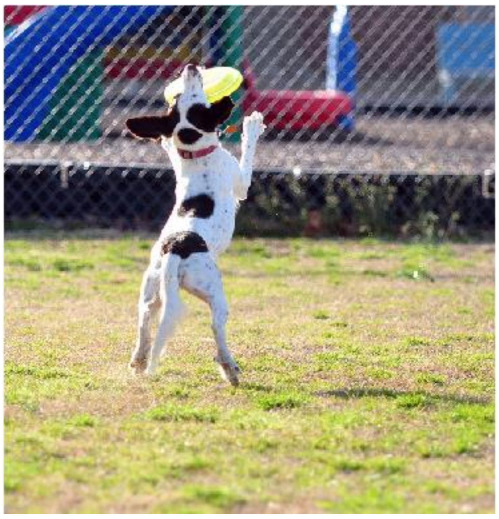
The model captures the people and the cake, but it misses the point completely.

Fig 21: Examples where the model gets it almost right

15.3 Bad results



"a cat is sitting on a car in a car"

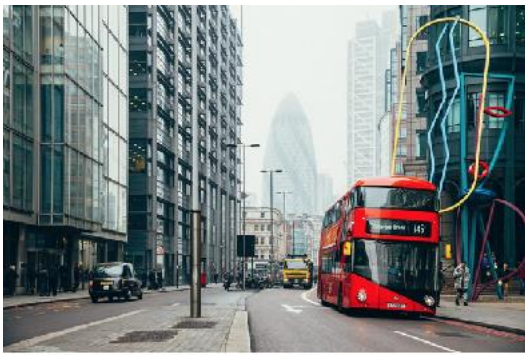


"a man with a baseball bat on a field"

Fig 22: Examples where the model gives bad results

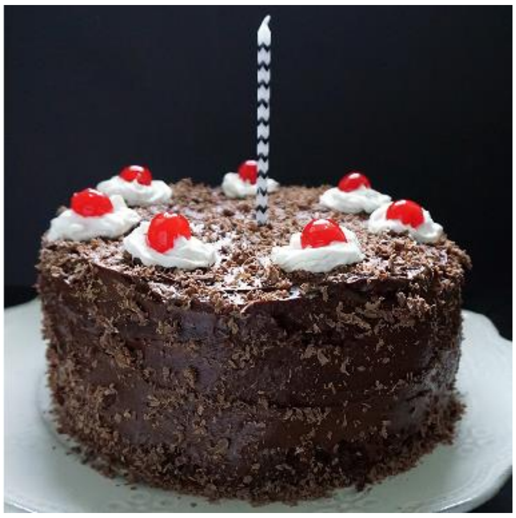
15.4 Images outside the validation set

Our model is trained on images from a certain distribution (the 80 classes of COCO2014). If we just take our own images or images from the Internet, performance is likely to be worse. But still it is interesting to see how well the model does.



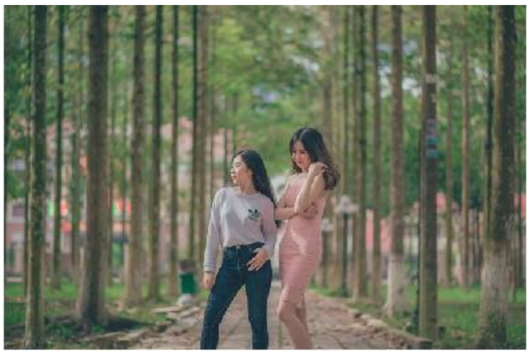
"a double decker bus is driving down a street"

Very good. There where streets and buses in the training set and the model can capture these in a photo from the Internet.



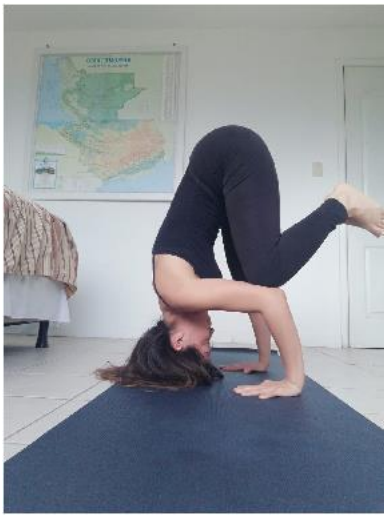
"a cake with a birthday cake on top of it"

The cake is so big, our model thinks it is two cakes on top of each other.



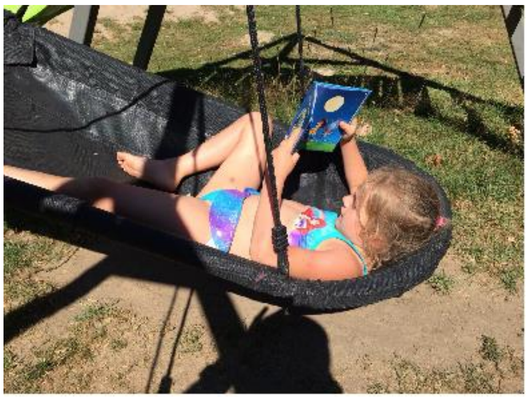
“a woman and woman are sitting on a bench”

The first part is correct, but we would say ‘two women’. It’s not clear why the model thinks that there is a bench.



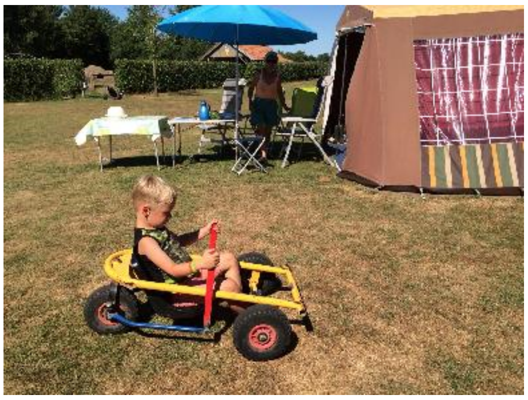
“a woman is playing a video game on a couch”

Actually this woman has gotten off the couch… It’s not clear why the model thinks she’s playing a video game.



“a little girl sitting on a skateboard in a park”

The model recognizes a little girl, and the grass could indicate a park. Clearly the model hasn’t seen the hammock-like object she is lying in before.



“a man and a woman standing next to a red umbrella”

This one is interesting. There are a (very young) man and a woman in the photo, but they are not together. And the model combines a blue umbrella and a red object into a red umbrella.
This could be a (typical) problem of the CNN, that learns to recognize objects but looses the spatial relationship.

Fig 23: Examples from outside the validation set

16 Recommendations for improving the model

The guideline for the size of this project is 60 hours. Give more time and resources (computing power) I would like to mention a number of things that we could try to improve the model:

- This architecture is tuned to be trainable on CPU. On GPU training takes less time and one can experiment more. We could use more hidden units in the LSTM and train for more epochs. And if it starts to overfit we can add dropout
- Make the last layers of InceptionV3 trainable, so the model can learn more appropriate weights for the image embedding
- Use a pretrained Word embedding (based on a larger vocabulary)
- Use a bidirectional RNN
- Instead of generating 1000 random batches each epoch, we could use a yield generator and loop through the entire training set every epoch
- This model uses argmax to generate the next word. We could use beam search.
- Using an attention model (which relates parts of the caption to parts of the image).

17 References

17.1 Notebooks

1) Final_Project_Image_Captioning.ipynb
2) Practica_Tecnicas_de_Optimizacion.ipynb
3) Practica_Deep_Learning.ipynb
4) My-fist-FFNN-on-MNIST.ipynb
5) Building-blocks-of-CNN.ipynb
6) My-first-CNN-on-CIFAR-10.ipynb

17.2 Papers

7) Vinyals et al., 2014. Show and Tell: A Neural Image Caption Generator
8) Tsung-Yi Lin et al., 2014. Microsoft COCO: Common Objects in Context
9) Szegedi et al., 2015. Rethinking the Inception Architecture for Computer Vision
10) Papineni et al., 2002. BLEU: a Method for Automatic Evaluation of Machine Translation

17.3 Books

11) Ian Goodfellow, Yoshua Bengio, Aaron Courville. Deep Learning. MIT Press, 2017
12) Aurélien Géron. Hands-on Machine Learning with Scikit-learn and TensorFlow. O'Reilly Media, 2017
13) Andrew Ng. Machine Learning Yearning. Draft, 2018

17.4 Online Courses

14) Coursera, Introduction to Deep Learning, National Research University Higher School of Economics, Faculty of Computer Science
15) Coursera, Deep Learning Specialization, deeplearning.ai
16) Stanford CS231n, Convolutional Neural Networks for Visual Recognition

17.5 Miscellaneous

17) Tess Fernandez, 2018. Notes from Coursera Deep Learning courses by Andrew Ng
18) Martin Görner, 2017. Learn TensorFlow and deep learning, without a Ph.D.
19) Adrej Karpathy, 2015. The Unreasonable Effectiveness of Recurrent Neural Networks