University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation:	Spring/ Autumn semester, 20..... Open / Confidential
Author:	
Programme coordinator: Supervisor(s):	
Title of master's thesis:	
Credits:	
Keywords:	Number of pages: …… + supplemental material/other: …… Stavanger, …….. date/year

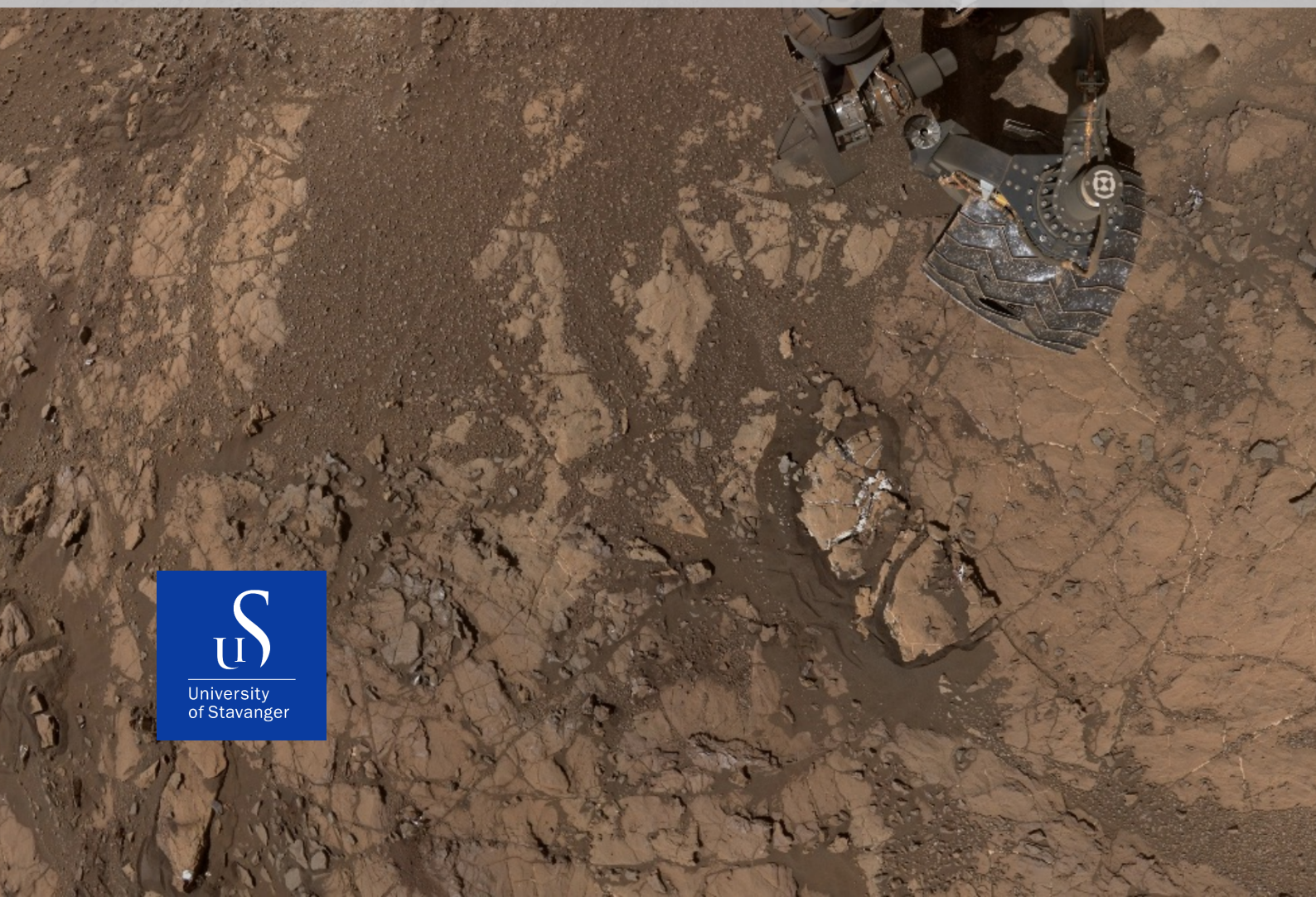
Universitetet i Stavanger

Captioning Mars Geology

geological features in images taken by the Curiosity rover

Markus Fjellheim
DATMAS

2020



University of Stavanger

Abstract

Space-crafts and their instruments tend to collect way more data on their missions than what can be transmitted back to earth in a timely manner. This leads to the need to prioritize what data is to be downloaded and what is not. This writing focuses on automatic caption generation of images from the surface of Mars taken by rovers that are sent to Earth as a way to save bandwidth and making the images searchable[Ono+19].

The high latency and low bandwidth between Earth and spacecrafts complicates communication. High latency makes it difficult to control the crafts as you don't see the results before hours later. The low bandwidth does not help either as downloading data from the crafts takes a long time. As an example, if you want to download the music video of the song "Never Gonna Give You Up" by Rick Astley [Ast09], it would take about 3.17 hours. Had it not been for the bandwidth limitation, almost all the surface of Mars would already have been photographed by the Mars orbiter, but as of now, only 3% is[Ono20]. It is therefore important to be able to choose what images to download and which images to not. Currently, the methods used to decide on what images are of interest is by downloading thumbnail versions of the images and/or highly compressed versions used to make decisions as to whether the image is of interest. Another method is to use pixel value difference between the images to prevent identical or too similar images from being downloaded[Ono20]. Future Mars missions might have modern radiation safe GPUs on board, like the Snapdragon 820/855, for the purpose of data analysis. This will allow for some of the data analysis to be done on board the rover without the need to transfer the data back to Earth, only the abstracted features. This allows for better determining similarity between images and decisions to what images are worth spending the valuable bandwidth on[Gho+20]. Machine learning capabilities and machine vision also increase opportunities for autonomic control of the rover. Automatic caption generation of geological features in the images allow geologists on Earth to search for geological features and choose the images of interest based on the captions.

Contents

List of Figures

CHAPTER 1

Introduction

The further away a craft is, the signal strength to that craft is reduced by the inverse square law, but the crafts ability to capture the data remains the same. This leads to an ever increasing need for a smarter way of transferring information across space the further into space we want to explore. Another problem related to distance is latency. The high latency between Earth and crafts like the Mars rover, makes them more difficult to control. The low feedback frequency and limited bandwidth leads to more rigid planning of the actions to be taken by the rover. As discussed in MAARS[Gho+20], this could lead to missed discoveries, described as the Unnoticed Green Monster Problem (UGMP).

Figure 1.1: UGMP. Illustration by Masahiro Ono, taken from[Gho+20]

The ability to analyse data on the fly allow for the detection of features the rover was not specifically asked to look for. By automatically captioning the images and coupling the captions with geo-data, a searchable database can be built without the need to spend valuable bandwidth downloading all the data, and valuable time spent by the geologists having to analyse every image manually. For the captioning model to work efficiently on Mars geology, it has to learn from training data obtained by Mars rovers, and labeled by geologists. A training set of 1,000 captioned images created by geologists is the available training data, provided by NASA's Jet Propulsion Laboratory (JPL). Since the target information in a caption is low, but caption generation from images is a complex task, the models tend to be complex. This cause problems of over fitting. The methods used to deal with these problems are discussed in this writing.

Research Problem Training a complex model on limited data is expected to lead to over-fitting. Regularization techniques can be applied to reduce the over-fitting, however this comes at a cost of not accurately have our model fit the dependencies in the training data.

We want to see if splitting the

more complex model into smaller parts with sub tasks and intermediate loss functions, makes it easier to find the balance between regularization and fitting true dependencies. We want to take advantage of transfer learning when possible and also design loss functions at intermediate steps with enough target information, allowing for the creation of more complex models and fine tuning of the pre-trained models. We also believe the splitting will help the debugging process as we can isolate problems to different parts of the model.

By creating a loss function more similar to the scoring methods used to validate the output of machine language models, we hope to make the model more directly reduce the validation score rather than indirectly through the loss function. We explore the possibility and output of a differentiable Jaccard Similarity loss function. Instead of having to fine tune the loss function to fit a separate scoring function, we imagine it would be more direct approach simply using the scoring function as a loss function. However, there is no guarantee such a loss function creates an easely traversable loss space, or introduces other problems as discussed in subsection BLEU score / Jaccard Similarity.

Outline Exploring the data we find the main problems is the low amounts of target information in combination with too complex models required to solve the problems. Dealing with overfitting is a common theme throughout the writing.

We try ways of increasing the amount of target information splitting the model into parts. Also using target information from elsewhere by the means of pre-trained weights using transfer learning. We explore ways of using unsupervised clustering, autoencoding, and segmentation as ways of increasing target information. We experiment using synethetic data generation to increase the amounts of training data. We introduce new information by manual image segmentation. We have varying degrees of success, but are not able to improve on the performance of the SCOTI captioning model (Scientific captioning of terrain images[Ono+19]) model developed at NASA JPL.

Comparing the results numerically turned out to be a challenge due to the discovery of test data tainted by training data. To overcome this problem, new data was downloaded to use as validation data, however, the lacking of labels made most of the comparisons done manually and visually rather than numerically. This way of comparison is not ideal as it is inaccurate. In order to further tune the performance, untainted labeled test data is needed.

Comparing results visually has the problem of human bias being at play. Considering the data is of the Martian surface regarding geological features, some of which appear frequently, a model simply fitting the distribution of the training data/test data and not the dependencies might still produce results looking reasonable. *Scramble loss* and other metrics are used throughout to compare results of different implementations and configurations of models.

CHAPTER 2

Related Work

The previous work done at JPL is splitting the captioning tasks into encoders and decoders with two decoding tasks. Two models are used together to form the final caption of the image. "Soil Property and Object Classification" (SPOC[Rot+16]) is a terrain segmentation algorithm that can identify the types of terrain used for rover navigation. It consists of an encoder, and a decoder producing the segmentation, see figure 2.1.

"Scientific captioning of terrain images" (SCOTI[Ono+19]) is a captioning model, generating the captions based on output from a layer of the encoder. Transfer learning is taken advantage of encoding feature vectors for the captioning decoder. SCOTI uses the VGG19 model pre-trained on ImageNet. The caption decoder is generating captions with the help of teacher enforcing and Bahdanau Attention[BCB14]. BLEU score is used to evaluate the performance of the model after training. This writing is exploring ways of bettering the encoder model by specializing it on data more specific to Mars features rather than images of cats and dogs and the like, that is found in ImageNet. Multiple methods were taken use of approaching the problems, many of which build on previous work. Our auto-encoder approach uses a feature loss function mentioned in Jeremy Howard's fast-ai course [How19]. He makes use of this method to improve the quality of his *decrapification* model (a model that removes noise from images). We use the function to better the auto-encoders ability to encode features rather than colors.

For the manual segmentation approach, our model is inspired by the u-net model [RFB15]. The model is used in biomedical image segmentation. It takes advantage of both low-level features and high-level features to segment images. This is similar to what we need to do when segmenting the Marian surface based on geological features that consist of a combination of low and high-level features.

Using synthetic image generation can be used to train model then applied to real data. There are a multitude of advantages but also challenges. Advantages are you can generate as many images as you want, and change the parameters controlling the generation, adjusting your training data on the go. This method works for learning projection and spacial orientation[Qi15], but can it also be used for detecting geological features?

"A DIFFERENTIABLE BLEU LOSS. ANALYSIS AND FIRST RESULTS"[CFC18] attempts at making a differentiable BLEU score used as a loss function. BLEU scoring is usually used for validating the performance of a model, and not as a loss function. We further explore the topic of the more direct method of using the valid-

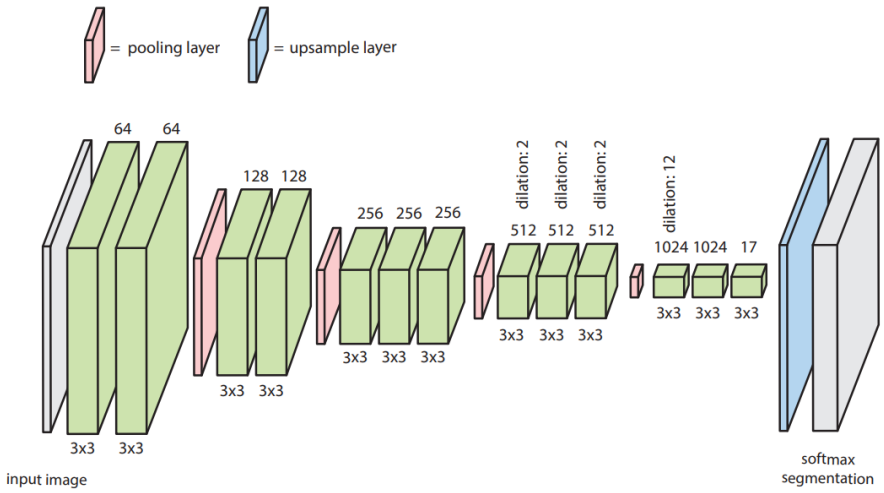


Figure 2.1: SPOCs terrain classification model. Taken from[Rot+16]

ation scoring system directly as a loss function.

CHAPTER 3

Problem Definition

We split our tasks into research goals.

Our first goal is exploring if we can reduce over-fitting and increase or take better advantage of the existing target information in training data by splitting a more complex model into different parts.

Our second goal is to see if this splitting can also help the development of the model by easier detecting the location of errors. By using different loss functions at outputting results at different stages, we can better detect where and what problems are.

Our third goal testing the possibility of using the Jaccard Similarity as a way of fine-tuning the performance of our model. Previous attempts have been performed using scoring as loss functions, for example, BLEU score [CFC18]. We want to explore the possibility of combining such an approach with a cross-categorical entropy loss function.

Our fourth goal is exploring ways of using unsupervised ways of turning information in the training data into target information. This is done as a part of splitting the full model into different parts. The unsupervised methods include clustering, segmentation, and auto-encoding.

Our fifth goal is to try to use synthetically generated data to increase training and target data. This is another way of increasing the amount of target information. We want to know if it is possible to make the synthetic data similar enough to the real data for models to work in production.

CHAPTER 4

Data exploration

Figure 4.1: Example training image taken at the surface of mars. The human created caption says: "mesas of layered sandstone outcrops and dark sand"

The training data The training data used is 1,000 images taken by the Curiosity Mars rover, captioned by human geologists. Some of the images are cropped versions of the larger images with captions corresponding with visual features. The total number of cropped images are 3067 train images, and 52 test images with captions. The data is provided by NASA Jet Propulsion Laboratory (JPL). See figure 4.1 for an example sample.

Creating a caption model that turns images into captions tend to require complex models, that is, models with a lot of learnable weights. First, features found in the image must be found, which might require the detection of low level and high-level features and combining then to classify the features occurrence, and spatial relations to other features. For these features to then be turned into captions, the understanding of basic English language knowledge is needed to produce English that is both grammatically correct and can describe the features found in the image. To train complex models, the need for large amounts of training data is often needed to prevent over-fitting and to learn the general dependencies between the input images and captions. The provided data provided pose some challenges regarding the above-mentioned needs.

Low target information The target information (captions) are low on information. The average number of words is about 8.3 and the size of the vocabulary is 148 unique words. This makes it difficult for the model to learn what complex features in the images to look for when the output is only about $8.3 \cdot log_2(148) \approx 60$ bits of information.

Large variation of training samples Besides the number of training samples not being that large, the variation is also large. Some of the images are of close up features in the centimeter scale, and others are of mountains and mounds. The large variation of

images makes over-fitting even more of a problem as simplifying the model may not be used to the same extent as a regulation technique since such a model might not be able to capture all the features in the images. Some images have different field of views, different compression rates, some have colors and some don't, the weather and lighting are different. There are also images with various types of artifacts that might make it more difficult for the model. Some of the geological features also have a low rate of occurrence. Having the model accurately detect those features while being complex enough to detect the rest of the features, would require more labeled training data.

Similar images Besides the variance of motives, some of the images also form clusters of motives. Many of the images taken are of very similar types of landscapes or the exact same motive, but a few degrees of in panning angle. The cropped images also contribute to the clustering.

Test error leakage During the training of the models to be described at a later point in this text, some suspiciously good results came up. Captions were generated predicting features that were only seen once in the training data. It turned out there was some data leakage between the test and training data, probably due to the way the croppings were done. Figure 4.2 shows an example of this, where an image taken from the test and train data had overlapping regions. Figure 4.3 shows one of the automatically generated captions matching the real caption too well. Even a model perfectly fitting the dependencies in the training data would not be able to perfectly caption an image describing the features in the identical order using identical wording

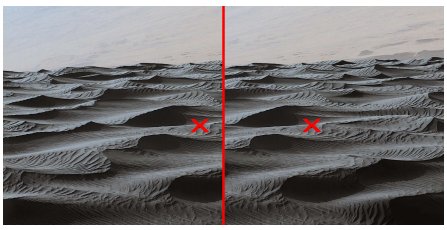


Figure 4.2: Example of two images found in the test and train data that are too similar. Use the red x as a landmark.

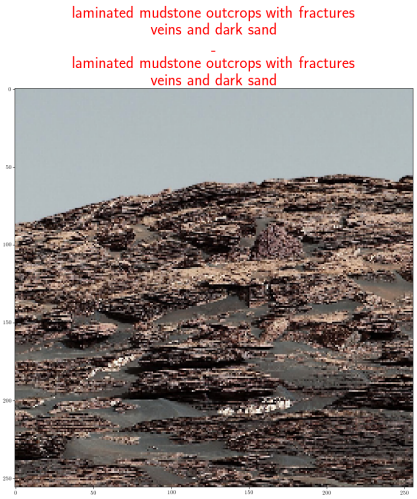


Figure 4.3: An example of a generated caption suggesting train to test data leakage. The top caption is the true caption and the bottom one is the automatically generated one.

unless by accident. The frequency of images like in figure 4.3 occurred at a too high frequency.

To validate the suspicion, new images taken by Curiosity at a much later date (to prevent similar motives) were downloaded and used as validation. This data is not labeled so the performance of the model is manually evaluated. The performance of the model dropped on this new data which

led to the further tweaking of the model to prevent over-fitting.

CHAPTER 5

Method

5.1 Increase target information

As mentioned in chapter 4, the target information when captioning the images is low. This in combination with relatively low amounts of training data, a model with many trainable parameters and some features with a low occurrence rate, leads to difficulties balancing the model's ability to be complex enough to fit the dependencies, but not over-fit the data, even using other regularization techniques. To solve this problem, the model is split into several models consisting of encoders and decoders of data and features with images entering one side of the stack, and captions exiting at the other end. Different loss functions and target data are used to train each sub-model that makes up the final caption model. One of the loss functions was related to segment the image on a pixel by pixel level by features. Multiple different ways of doing this segmentation are mentioned in the following sub-sections, but the idea behind them is the same. By doing some kind of image segmentation, you increase the target information from simply a caption to that of a large number of pixels belonging to one or more classes. With more target information, the model can learn more from the training data. After learning the segmentation, the segmented image can then be sent to a captioning model doing the captioning. Since this part only needs to fit the English language and interpret what is the relative spatial relations of the features already detected in the image by the segmentation model, fewer weights are needed to detect the complex features by itself. This reduces the possibility of the captioner from over-fitting, by moving more of the more difficult job of recognizing the geological features to a part of the model that has greater ability to learn these features based on more information-rich training data.

5.2 Transfer learning

Performance specification Bandwidth is the limiting factor when it comes to available resources. This stays true as long as the machine learning models have an inference time smaller than the transfer rate. As the machine learning models are trained on earth, the time to train the models is also not much of a problem. Memory is not much of a problem either. The choice of machine learning models, therefore, depends on accuracy rather than low inference time or size of models. [Ono20]

Choosing the model The first part of the model consists of a pre-trained convolutional neural network architecture. During initial testing of architectures, mobileV3net[How+17] was used

for speed. At a later stage InveptionV3[Sze+15] and ResNet[He+15] was used for higher accuracy. The architectures was pre-trained on ImageNet[Den+09]. Using transfer learning, one introduces knowledge learned from training data different from the one you use for a specific problem. InceptionV3 and ResNet consist of multiple convolutional and other types of layers, reducing the input image from a tensor of image width by image height by 3 color channels down to a much smaller width and height, and much larger depth value. At the end of this model is a "head" which results in the classification task that is the target of the ImageNet data set. We use only the "body" of this model, as the "head" part, the classification is of no use for classifying the Martian terrain.

Fine tuning for Martian features Even though Mars is much different than the types of images in ImageNet (cats, dogs, and cars), some of the low-level features (basic shapes and patterns) will probably be useful for both tasks. The next step is to fine-tune the pre-trained weights to more fit the task at hand, learning the more common features of the Martian terrain rather than those of cats and dogs. Several methods were attempted to perform this fine-tuning, discussed more in the sub-section 5.4 Segmentation model. Some of these methods were successful and some were not.

5.3 The general architecture of the model

The captioning model consists of multiple parts. The first part is an image segmentation model. This model takes in an image tensor as input and has a tensor describing the size and location of the detected features found in the image. These features are then sent to a captioning model that generates a caption based on the features.

The segmentation model consists of an encoder and a decoder. The encoder takes advantage of transfer learning to encode the image into multiple output layers describing detected features of various levels of abstractions. The decoder turns these layers of features into segmentations. By fine-tuning the encoder to do segmentation well, we can fine-tune the pre-trained encoder to better fit the Martian training data.

The caption generator takes the output of the segmenter as input and produces captions as outputs. The captioner also has an encoder and a decoder. The encoder encodes the segmented data, reducing dimensionality, passing the important information to the decoder. The decoder consists of an RNN that produces the final caption. See figure 5.1 showing this architecture.

Each part of the model (segmentation encoder, -decoder, caption encoder, -decoder) is first trained separately, before being trained together for fine-tuning. Training the models separately allows for fewer trainable weights leading to less over-fitting. It also allows for more rapid testing of new ideas as the training time is faster. Debugging the model is made easier as problems might be noticed earlier. See figure 5.2 as an example of this. In this figure, one can see the model is detecting a rover in the outcrops and also detect some sky in the sand. Splitting the model into multiple parts limit the search-space of where the problem could be. If one simply had the full captioning model outputting there being a rover where there is none, it would be difficult to know what part of the model has a problem. Now we know the problem is the model has problems identifying rovers since we see from the figure that bedrock outcrop is detec-

ted fine, but the rover is being falsely detected.

5.4 Segmentation model

The segmentation model is not using the captions to segment the image. Instead, various methods were used constructing loss functions rating the segmenter by its ability to learn something from the images themselves. By using the train data to create target data, more target information is introduced compared to the low information in captions only.

5.5 Unsupervised segmentation

Due to the large number of training data, avoiding having to manually label images were attempted. The unsupervised segmentation approach is using a pre-trained image model to segment the images into similar-looking features. After the segmentation, the idea was to either use the segmentations themselves as input to the captioning model or fine-tuning the pre-trained image model on the segmentation task and then using the features in the captioner.

The segmentation model The segmentation model consists of, as mentioned earlier, a decoder and an encoder. The encoder is a pre-trained on ImageNet convolutional model. Both MobileNet, ResNet, and InceptionNet was tried, with similar results. In the case of InceptionNet, the input dimensions were images of size 512 by 512 by 3 color channels in size. Black and white images were tiled/replicated along the red, green, and blue dimension and treated the same way as the colored images. The 92nd layer was used as the output to the encoder. This layer has dimensions of 61 by 61 by 96, so this is the dimensions of the output of the segmentation encoder. The decoder is a simple fully connected layer applied along the filter dimension (of size 96). The output of the segmentation model is then 61 by 61 by the number of classes. The classification is a type of clustering since the names of the classes are note specified. The hope is that the segmentation will cluster/-classify the parts of the image covering the same or similar geological features to the same class. See figure 5.3 for how the segmentation encoder and decoder looks like.

The loss function The loss function is designed to reward features in the images being given the same class. The name of the class is not important, in how to interpret the classes is up to the caption model to decide.

Correlation loss Since similar features tend to occupy the same area, the first iteration of the loss function gives a larger loss if pixels close by are classified as the same class. This will incentivize the model to segment parts of the image of the same class. The output of the segmenter is 61 in width and height. For each output to be similar to its neighboring outputs, it must use the found features it shares with its neighbors to predict the same class. The model and loss function is implemented in TensorFlow[Mar+15]. The implementation of the similarity loss between neighboring pixels took advantage of TensorFlow's convolutional layers to make a Gaussian blur function. The difference between the non-blurred predicted classes and the non-blurred predicted classes is small if many pixels close by are of the same class. If the image is fractured into many classes, the edges when one class end and another one begins will bleed over each other in the blurred version,

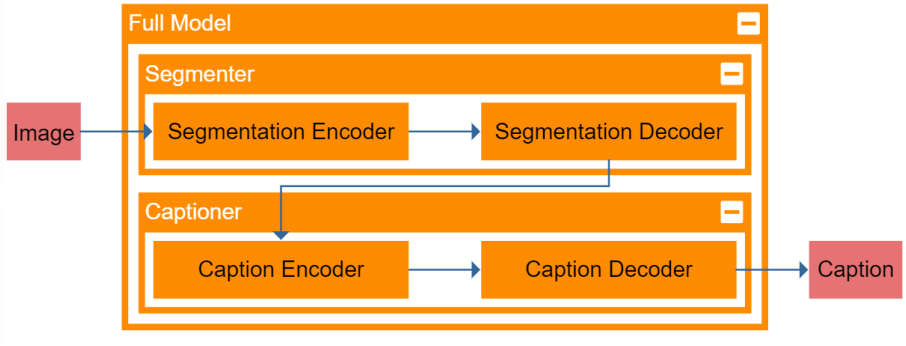


Figure 5.1: High level of abstraction diagram of pipeline.

leading to a larger difference between the blurred and none-blurred predictions. The implementation in TensorFlow looks like this:

```
# segmentedImage.shape = (
  batchSize, width,
  height, numClasses
)
blurrySegmentedImage =
  guassianBlur(segmentedImage)

# Correlation error.
correlationError =
  tf.reduce_sum(tf.math.abs(
    segmentedImage -
    blurrySegmentedImage
  ))
```

segmentedImage is already put through the softmax function along the *numClasses* dimension.

We can see in figure 5.4 an example of calculating this loss.

The input image (of size 512 by 512 by 3) is sent through the segmenter model (output is 61 by 61 by the number of classes). The number of classes in this example is two. The first class is represented with the color red, the second one is represented by the color green. The loss function also puts the predicted classes through a softmax function along the filter/class dimension (size 96) making the predictions' probabilities between 0 and 1 for each class. The prediction is then sent through the Gaussian filter. The difference between the blurry image and the segmented image is calculated. The result is an edge detection filter. The more edges, the less separated the classes are. The edges are summarized and used as the loss function. Let us call this loss the *correlation loss*. After training the model, the results are as seen in figure 5.4.

We see there is a problem. All the pixels are classified to be of the same category. The reason is this is the easiest way for the model to get a perfect score. To fix this error, a new additive to the loss function is introduced that punishes unbalanced categorizations.

Balance loss To prevent the model from simply classifying everything as being of the same class, a class balancer additive to the loss function is made. Let us call this loss the *balance loss*. The total loss is now the sum of the *correlation loss* and the *balance loss*. The *balance loss* is zero if all classes are given the same area. If one or more classes are approaching zero area, the balance loss will increase. *Balance loss* is calculated across each batch, so the batch size must be large enough for there to be an approximately even distribution of the various classes. The samples of the batch must also be

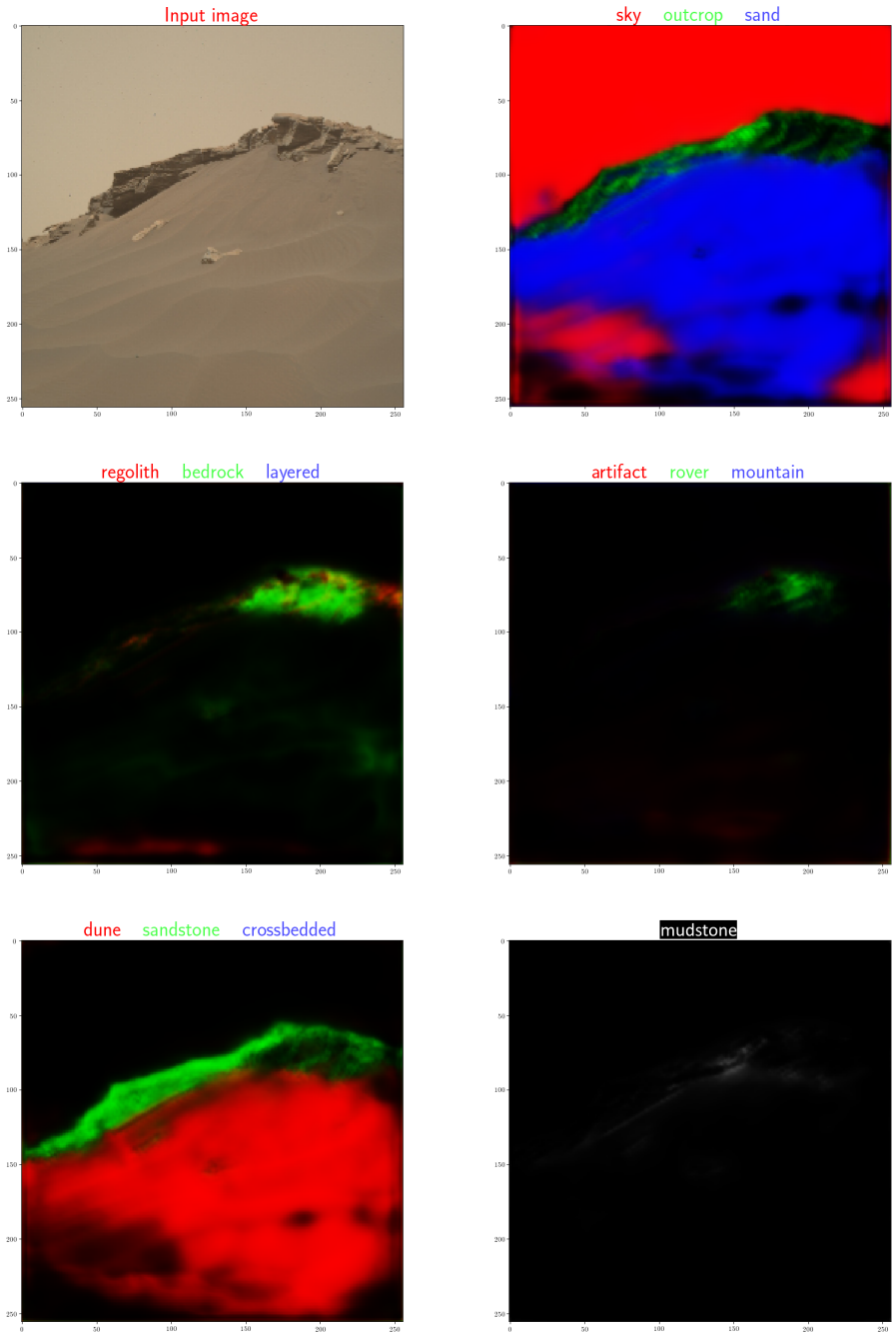


Figure 5.2: Example of image labeling.

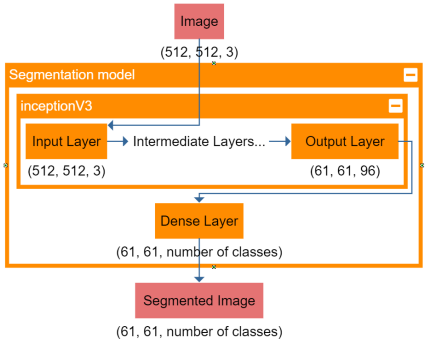


Figure 5.3: The unsupervised segmenter architecture. The numbers below the blocks show an example of the output tensor shape. Different configurations lead to different numbers.

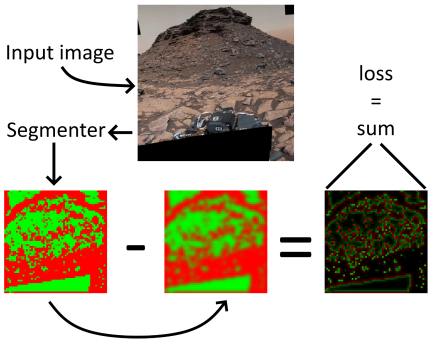


Figure 5.4: The process of calculating correlation loss.

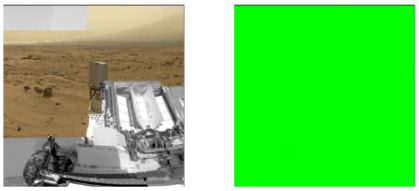


Figure 5.5: Correlation loss function leading to a single class prediction. The left side shows the input image, the right side shows the prediction.

representative of the data set as a whole. The *balance loss* is designed to give a small punishment if the difference in area is small, but get progressively greater as the classes are getting more uneven. An example of such a function is the following:

$$balance\_loss_0 = \sum_{i=0}^{n-1} \frac{1}{a_i}$$

n is here the number of classes. a_i is the area of class number i in the batch relative to the total area of all the images of the batch. To be able to compare the score of a varying number of classes, a constant is added to the loss function to make the loss zero when all classes are balanced.

$$a_1, ...a_n = \frac{1}{n} \quad \Rightarrow \quad \sum_{i=0}^{n-1} \frac{1}{a_i} = \sum_{i=0}^{n-1} \frac{1}{\frac{1}{n}} = \sum_{i=0}^{n-1} n = n^2$$

The final *balance loss* is:

$$balance\_loss = \sum_{i=0}^{n-1} \frac{1}{a_i} - n^2$$

Applying this additive to the full loss function, the result looks like in figure 5.7

The implementation of this loss in TensorFlow:

```
# Class balance error.
# Reduce along batch, with, height
# dimension.
classSizes = tf.reduce_sum(
  segmentedImage, axis=(0,1,2)
)
# Linear softmax.
classSizes = classSizes /
  tf.reduce_sum(classSizes)
classBalanceError =
  tf.reduce_sum(1./classSizes) -
  segmentedImage.shape[-1]**2
```

As we can see, the results are still not good. Yet again did the model find a way to reduce the loss without doing actual segmentation. In the

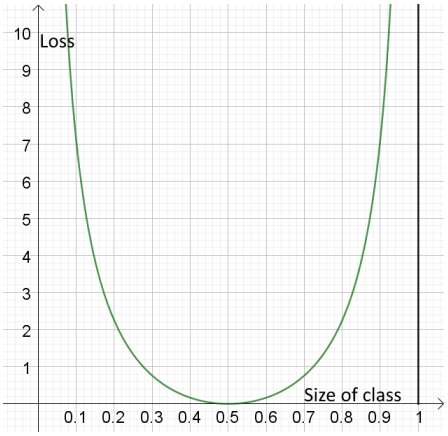


Figure 5.6: *Balance loss* function of two classes. The horizontal axis is the size of one of the first class. The other class size is $1 - size\_of\_first\_class$. The vertical axis is the loss.

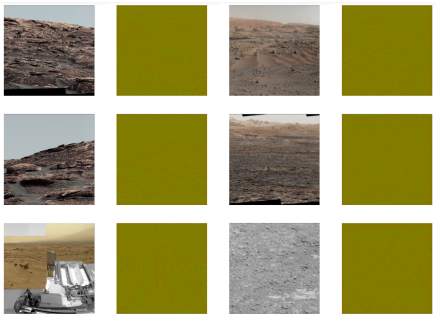


Figure 5.7: Example results when both correlation loss and balance loss is applied. The leftmost and 3rd from the left column shows the input images. The images to the right of each input image shows the prediction.

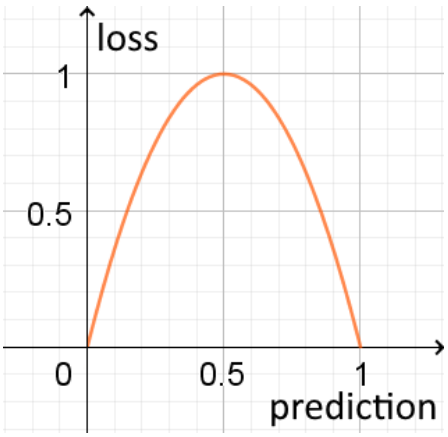


Figure 5.8: Loss function for indecisiveness. The more confident the guess is (closer to 0 or 1) the smaller the loss. The function is $4 \cdot prediction \cdot (1 - prediction)$.

figure, there are two target classes. The classes are represented with the color red and green. In this case, the model predicted a 50%-50% mix guess for classes. This minimizes both the *correlation loss* and the *balance loss*. The *correlation loss* is minimized since there is low fracturing of the predictions and the *balance loss* is minimized due to both classes having equal representation. The problem is the segmentation model does not do any classification since none of the predictions have confidence. To fix this issue, a new additive to the loss function is introduced. One that punishes indecisiveness.

Indecisiveness loss Without an *indecisiveness loss* additive to the total loss function, the easiest way for the model to minimize the loss is by predicting 50%-50% on all pixels. To prevent this, a loss is introduced that punishes predictions that are far from either 0 or 1.

The *indecisiveness loss* is summed

together for all the predictions. The following is a TensorFlow example code.

```
# Indecisiveness error.
allPredictions = segmentedImage /
  tf.reduce_sum(
    segmentedImage, keepdims=True,
    axis=-1
  ) # Linear softmax.
indecisivenessError = tf.reduce_sum(
  allPredictions * (1-allPredictions)
)
```

Balancing the losses The full loss function for the unsupervised segmentation is a sum of additives. Each additive is summed together multiplied with various factors to form the final loss. In figure 5.9 we see the changes of the three losses over time. The graphed losses were compared to visualizations of the predictions as seen in figure 5.10.

Balancing the losses required a lot of tweaking. Eventually, factors were found that resulted in images that were segmented by having nearby pixels belonging to the same class, not unbalanced, and the predictions were confident. See figure 5.10 as an example prediction on test data.

As seen in the figure, segmentation is happening. However, after further inspection, we notice the segmentation is actually just some form of edge detection rather than segmenting the images by features. By segmenting the image by edges and surfaces, the model is able to separate the model into two classes that have similarly classified neighbors, the classes are balanced and the predictions are confident. This leads to a low loss, but it is not solving the problem as intended, that is to cluster based on geological features. Different approaches are attempted to make the segmenter actually segment the images by these features. An example is replacing the dense layer in figure 5.3 with a convolutional layer. This however leads to further edge detection. Different pre-trained models, for example, MobileNet, ResNet, and InceptionNet, made no difference either. Another approach is using lower level or higher-level layers of abstraction extracted from the pre-trained models (higher layers being close to the model's head (end), and lower levels being closer to the model's input). The idea behind extracting higher-level layers from the pre-trained model is that edge detection at this stage would lead to a too fractured image for doing edge detection being a good way of lowering the loss function. The idea behind extracting lower-level layers from the pre-trained model is that the model has at this point abstracted away low-level features such as edges, as this is not necessary for doing ImageNet classification (ImageNet being the data set the pre-trained model is trained on). Neither of these strategies worked. If extracting lower level features (from layers earlier in the pre-trained model), the model tends even more towards edge detection. Maybe due to the model not being able to recognize features at this low level of detail, it would instead need more perspective. If extracting higher-level features, the model achieves a good score by classifying the center of the image as one class, and the edges of the image as the second class.

No tweaking was successful at making the unsupervised segmenter segment the images by features. Another possible attempt was to treat each slice in the filters' output from the pre-trained convolution model as a vector and then use a clustering algorithm to segment the vectors. If the filter were of shape 61 by 61 by 96, there would be $61 \cdot 61 = 3721$ vectors of dimension 96 each. No reason why this approach would not also lead to edge detection over geological feature detection could be found. Therefor another unsupervised approach was made.

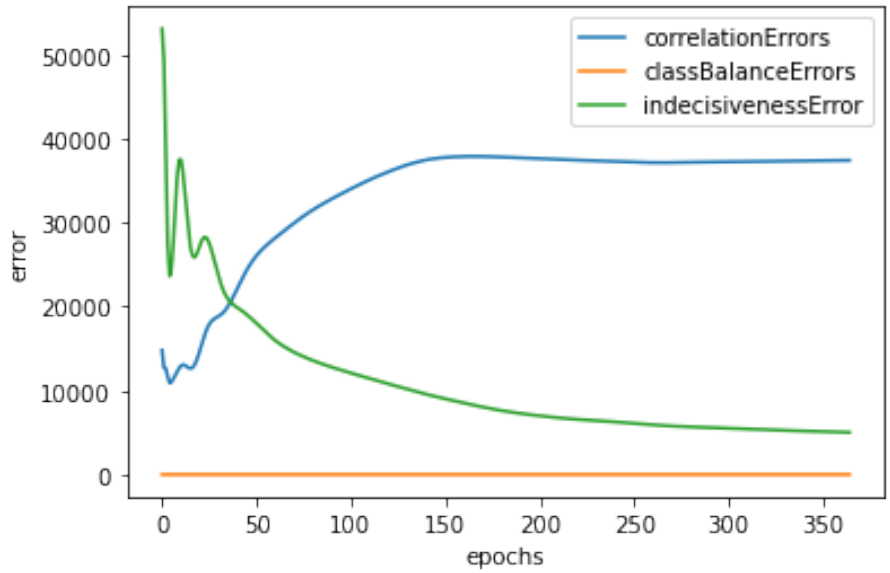


Figure 5.9: Example of balancing between the partial losses of the unsupervised segmenter. Here we see the correlation loss is increasing while the class balance loss is small. Maybe this means more emphasis (greater factor) should be used for the correlation loss.

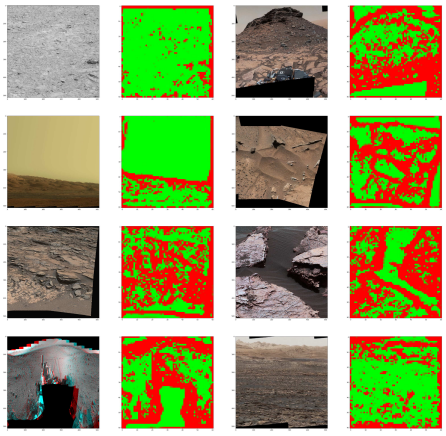


Figure 5.10: Example of segmentation model using the full loss function. The leftmost and 3rd from the left column shows the input images. The images to the right of each input image shows the prediction.

5.6 Autoencoder

The idea behind using an autoencoder is similar to using the unsupervised segmentation model. An autoencoder saves time as it uses the input training data itself as target data. All that is needed is images of Mars, no need for labeling. As with the unsupervised segmentation approach, we want to fine-tune the pre-trained convolutional image model to better fit the geological features on Mars. The finely tuned model might then be able to give better feature vectors to the captioning model, like the captioner part of SCOTI.

The autoencoder consists of a down-convolution encoder and an up-convolution decoder. For down convolution, models pre-trained on imageNet were used. After the end of the encoder, there is a fully connected layer

creating the encoding. The encoded vector is then passed to the decoder consisting of up-convolutions. The full architecture can be seen in figure 5.11.

Initially, the featurizer part of the model is not trainable, but the rest of the model is. The reason for this is since the rest of the model probably will not produce reasonable results in the beginning, this could corrupt the featurizer model. It is better to initially freeze the featurizer, and then unfreeze it during the tuning step.

The feature encoder model takes the output of the featurizer and puts the result through two fully connected layers. The first layer is the "Fully Connected Feature Selection" layer as can be seen in figure 5.11. This layer reduces the number of features to the ones the model learns is useful for the encoding by doing a linear combination along the feature dimension (the dimension of size 64 in figure 5.11). The next fully connected layer is the "Fully Connected Encoder". This layer fully connects the remaining nodes along all dimensions (except the batch size which is omitted in the figure). This layer is responsible for creating the encoding of the image. The reason the Feature Encoder consists of two fully connected layers instead of just one connecting the full input (output from the featurizer), is that the number of connections would be way too large. Dealing with only 64 of the 576 features reduces the number of weights in the "Fully Connected Encoder" layer from
$(8 \cdot 8 \cdot 576) \cdot (64) + 64 = 2,359,360$
to a more manageable
$(8 \cdot 8 \cdot 64) \cdot (64) + 64 = 262,208.$

In figure 5.11 between the encoder and decoder the bottleneck is located. In the example configuration in the figure, the number of dimensions used is 64.

The "Decoder" has a "Fully Connected" layer as the first layer interpreting the encoded image. This fully connected layer scale up the encoded vector and by the following reshape distributes the information along the width, height, and filter dimension of the first input to the "Up Block" layers. These layers do logic as well as scaling the image up to its original size. The initial loss function used to compare the output image and the input image is mean square error (MSE).

Blurriness problem When applying the model to the data, some of the images end up looking blurry. See figure 5.12.
The blurriness probably due to mean square error being used as a loss function. The way the autoencoder model encodes the image is remembering the average colors of features found in images instead of the features themselves. Figure 5.13 illustrate this problem.

The figure shows three graphs and three images. The top image illustrates a part of the target image. This image contain sand feature. The top (black) graph shows an example of the brightness of the sand surface along the width dimension (this is just an illustration, the brightness is not mapping to the image accurately). The second image shows what would be good decoding after encoding of the top image. The sand feature is captured in the encoding. Even though the exact location of the sand waves are wrong, the only thing that matters concerning caption generation is the encoder part of the autoencoder's ability to capture the feature and general location of the feature itself. What actually the autoencoder produces at its current state is the bottom image and graph (green). Instead of remembering the feature, the autoencoder instead encodes the average color. The reason for this can be seen looking at the black, red, and green brightness curves. There is a smal-

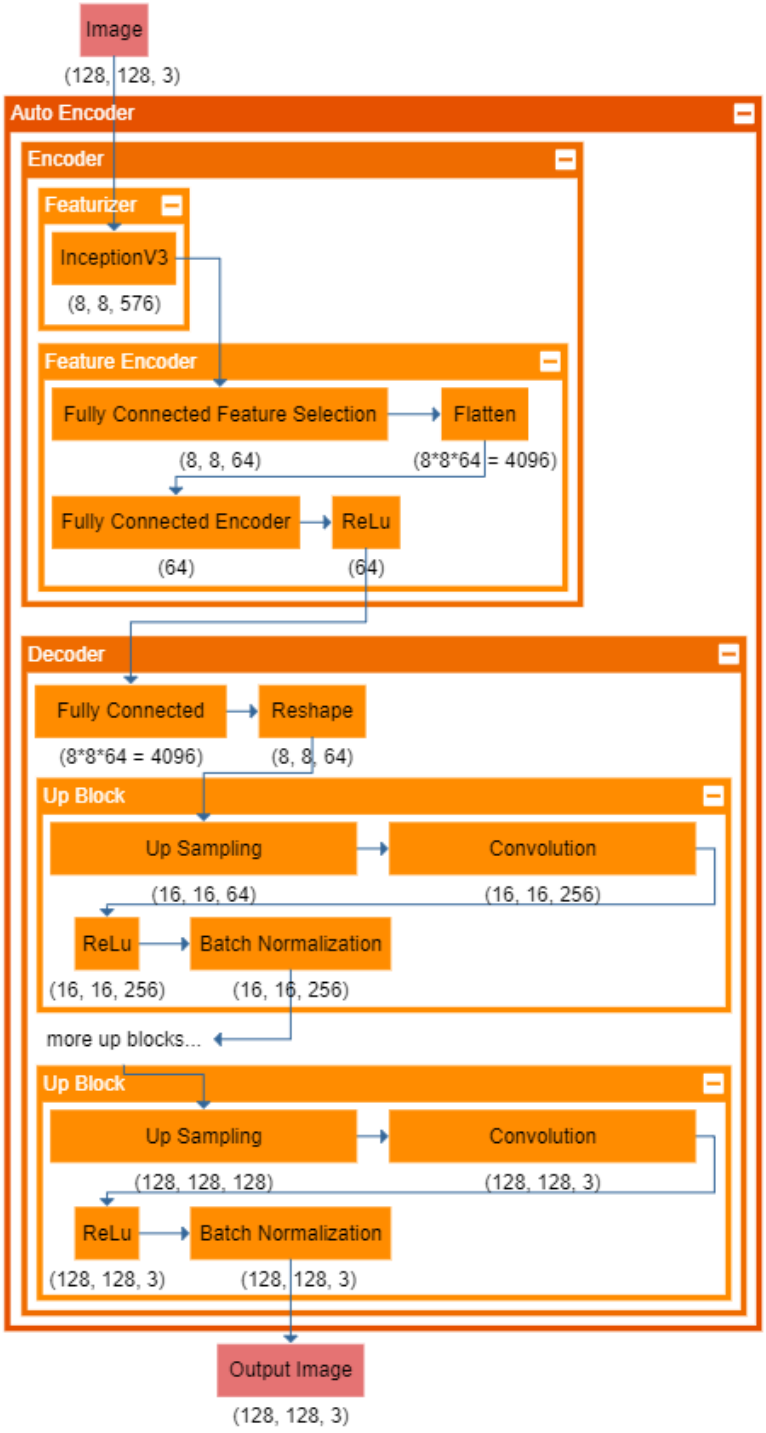


Figure 5.11: The autoencoder architecture. The orange blocks represent functions. The numbers show the tensor sizes for an example configuration. Batch dimension is omitted.

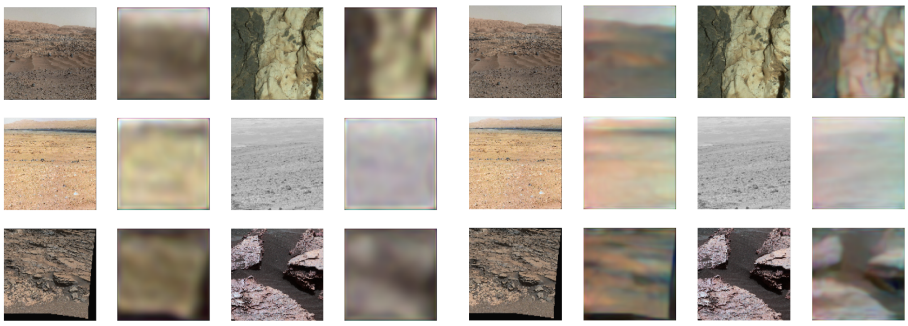


Figure 5.12: The leftmost and 3rd to the left columns is input images. Each image to the right of the input images show the result after the autoencoding and decoding.

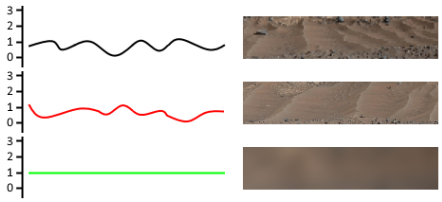


Figure 5.13: Illustration of the problem using MSE as loss function in the autoencoder.

ler MSE between the black and green curves than the black and red one. So even though a human might say the red curve looks more like the black curve, the loss is still greater.

Feature loss function In Jeremy Howards 2019 fast-ai course[How19], he created a *feature loss function* in his decrapification model. He trained a model on removing artifacts and blur from images using a loss function taking advantage of a pre-trained convolutional model. Instead of using MSE comparing the target and prediction image, he puts both the target and prediction through the pre-trained model. Different layers were then extracted

Figure 5.14: The leftmost and 3rd to the left columns is input images. Each image to the right of the input images show the result after the autoencoding and decoding. The loss function uses a *feature loss function* to capture the features of the target.

from this pre-trained model and compared using MSE. Using this method, you are not only trying to achieve the approximate correct color at each pixel but also trying to replicate a higher level of abstraction features. This method was implemented and tested to see if it could reduce the blurriness of our autoencoder. See figure 5.14.

As we can see from the figure is the images look sharper. Tests were also done comparing layers of higher abstraction as loss function. By higher level of abstraction, we mean layers closer to the input layer of the pre-trained model. See figure 5.15

Balancing the feature loss layers Using a feature loss function, a target image and a prediction image is compared. Both images are put through a pre-trained convolutional network. Layers within this network is extracted and compared using MSE. Each comparison between each layer we choose to extract leads to different losses. The most low-level of abstraction layer loss is simply the pixel loss. The higher level losses are from comparing losses

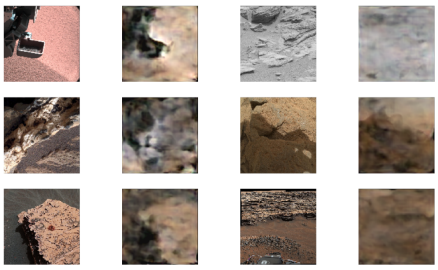


Figure 5.15: The leftmost and 3rd to the left columns is input images. Each image to the right of the input images show the result after the autoencoding and decoding. The loss function uses a *feature loss function* to capture the features of the target. The *feature loss function* extracts a layer closer to the input image of the pre-trained model.

between layers deeper in the model (further away from the input end of the model). We wanted the model to, at any time epoch, equally weigh each of the losses. By dividing each loss by the losses size of the sum of all the losses, each loss is scaled to the same value. To prevent the solution landscape from becoming flat, this dividend is is disconnected from the back-propagation graph. See below code example for implementation.

```
# Normalize.
sumOfFilterLosses = sum(
  [float(filterLoss) for
  filterLoss in filterLosses]
)
invSumOfLosses = 1/(
  float(pixelLoss) + sumOfFilterLosses
)
#
pixelLoss = pixelLoss/(
  float(pixelLoss)*invSumOfLosses
)
for i in range(len(filterLosses)):
  filterLosses[i] = filterLosses[i]/(
    float(filterLosses[i]) *
    invSumOfLosses
  )
```

The above code example is part of the loss function. *filterLosses* is a list of TensorFlow tensors, each of shape *(batch size,)*. These are the MSE of each compared layer. *pixelLoss* is a single tensor of the same shape. This is the pixel MSE of the target and predicted image. The normalization makes all values the same size by multiplying by *invSumOfLosses*. Since *invSumOfLosses* is disconnected from the back propagation graph, there will still be back-propagation gradients.

```
loss = pixelLoss
for i in range(len(filterLosses)):
    loss += (lossFilterFactors[i] *
      filterLosses[i])

return loss
```

At the end, the losses are summed together and multiplied by factors balancing the impact of each filter. These factors are chosen manually based on the output. See figure 5.16 for example output given different balance.

In this figure the loss function used is balancing two different features at two different layers. The low level feature has a balance factor of 1. The high level feature has factors as shown in the image. Increasing the high level feature balance factor from 0.25 to 0.50 makes the image sharper. Increasing it too much and the color balance is skewed.

Data cleaning. The model seemed to put a lot of efforts into fitting artifacts of the images. Some of the images are composite images with black areas where the shape of the image does not fit into a rectangular shape. The artifacts were removed by making the loss function ignore loss over those regions. The artifacts are found by looking for pixels darker than a threshold. See figure 5.17 for an example of this.

In this figure the input image produces a prediction. The model is not trained to make the error larger, making it easier to see the effect of the

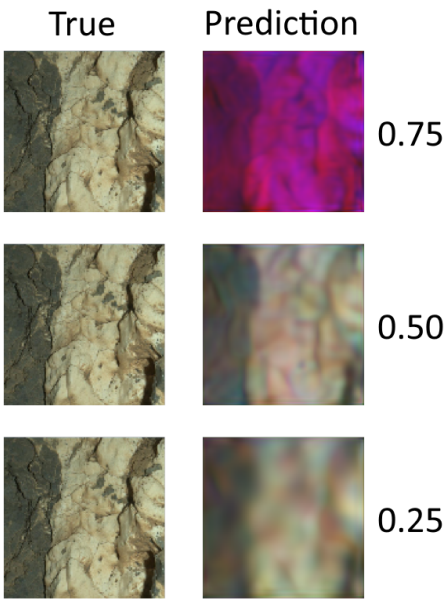


Figure 5.16: The outputs given different configurations of the loss function. The inputs are to the right and predictions are to the left. The numbers represent the relative high level filter factor compared to the low level filter factor.

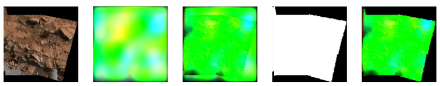


Figure 5.17: The leftmost image show the true image. The next image show the prediction. The next image is the MSE. The next image is the mask. White is inside, black is outside. The last image is the error after the mask is applied.

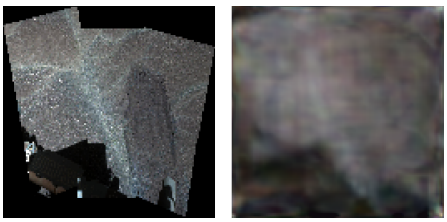


Figure 5.18: The image to the left is a training sample. The image to the right is a prediction. The prediction ignores the black borders as they are ignored by the loss function.

mask. After the mask is applied, the parts of the prediction outside the mask is ignored. See figure 5.18 for an example output.

As we can see, the model puts no effort into detecting the dark out-of-bounds area.

Optimalization To minimize the training time, we pre-featurized the images using the featurizer (see figure 5.11) before training the rest of the network architecture. We can do this as we do not train the pre-trained weights in the featurizer initially anyway. The featurized images are stored on disc and batch for batch is loaded during training dynamically using TensorFlow's data set object. We noticed training the model was slow at the beginning of each new epoch. We suspected this may be due to a problem with our training pipeline. The problem is since our model is not so complex that running a single batch through the model takes a long time, the loading of the features from disc becomes the bottleneck. Luckily our featurized images does not take so much space, so we instead fit all the training data in memory, doubling the training speed.

Results We had some problems with over-fitting. To deal with this, standard regularization techniques is performed, but this did not solve the problem. We also tried changing the sizes of input images. 512, 256, and 128 square sized images were tested with no difference. ResNet, InceptionNet, and mobileNet pre-trained on ImageNet in the featurizer also did not better the results. One attempt that help with over-fitting was reducing the encoding dimension, but this results in blurry predictions. Looking at figure 5.11 at the end of the encoder, we tried to add more dense layers for the encoder, but this only slowed the training but achieved no better test score. No good balance could be found between over-fitting, and a model simply fitting the distribution of the input data. By fitting the distribution of data, the model adjust the output to the average color at the given location of the image. Besides the overfitting problem, there is also a conceptual problem with the autoencoder approach. Using pre-trained models in the loss function as a way to fine tune a pre-trained model to detect features in the image data, is like trying to make the pre-trained model learn features from itself. The only thing the model might learn from the data is what combinations of features are relevant in the images. One way the autoencoder approach could still be salvaged, is by using a discriminator as in a generative adversarial network (GAN). This path was not explored as it too might lead to over-fitting issues, as already encountered using the *feature loss function*. Over-fitting seems at this point to be one of the more challenging problems. Approaches from this point of focuses on methods avoiding this issue.

5.7 Segmentation on synthetic images

The advantage of a synthetic image generator is the ability to generate practically infinite amounts of training data. If one wants to make tweaks to the training data, this can be done by adding small adjustments to the generator. The generator is in the case of image generation, a render engine. Another advantage is perfect labeling. The tiresome labeling process can be done much more accurately by the program, and if new labels are to be removed/introduced, new data can be generated automatically. The main challenges of synthetic images, is making the images similar enough to the production data. The data both needs to capture the features of the real data and be varied enough to fit the variation of the real data. Also, the synthetic data must not contain features that can be used by the model to do segmentation, that does not exists in the real data.

In figure 5.19 we see an illustration of potential distribution differences between real data and synthetic data. The horizontal axis (*feature axis 1* and *feature axis 2*) illustrate the different features of training data. The vertical axis illustrate the probability distribution of the data (the probability that a sample finds itself having this combination of features). Assuming prior probabilities for each class to be the same, the boundaries of a sample being classified as of a class is at the intersection of the classes density functions. In figure 5.19, *class A real* is the density function of features to be classified as class A (green), and *class A synthetic* is the density function of the synthetic data for class A (dark green). The same for class B. In this illustration, we see that the real and synthetic data is different as they

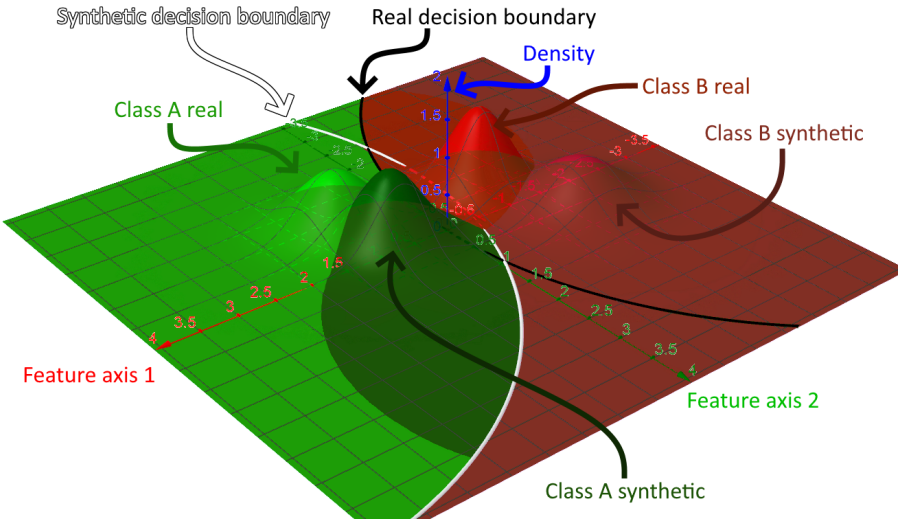


Figure 5.19: Illustration of probability density functions of real and synthetic data to be segmented into two classes. The black decision boundary separate the real samples, and the white decision boundary separate the synthetic samples. Figure created using [Hoh+13]

occupy different areas in the feature space. But the synthetic classes have density functions similar to the density functions of the real classes. This leads to decision boundaries separating the classes to be similar to each other. If a model is trained on sufficient numbers of real training data to classify samples of the two classes, the black decision boundary is used by the model to determine what class a sample is of. If synthetic data is used, the model will instead find the white boundary. As long as the model trained on synthetic data make boundaries that are separating the real classes, the model can be used on real data as well, alternatively fine tuned to fit real data.

Potential problems For complex models segmenting pixels to different classes, there will probably not be as simple decision borders as seen in figure 5.19. Anything outside the distribution of the data trained on may lead to random classification. A way to reduce this problem, is increase the variance of the synthetic data. This can be done by varying parameters during rendering. Varying field of view, orientation of motives, color balance, weather, and adding noise, are some examples of ways of ensuring the synthetic data covering the variation of the test data.

Some features in the synthetic data that does not exist in the real data, might have features that can be used for classification. If the model start using these features, the model will start making mistakes on the real data. The synthetic data having features the real data does not have, means there will be parts of the distribution of the real data outside the distribution of the synthetic data. See figure 5.19. We see that *class A synthetic* has distribution leaking to the right of *class B real*. To remove/reduce this leakage, both the synthetic and real data can be pre-

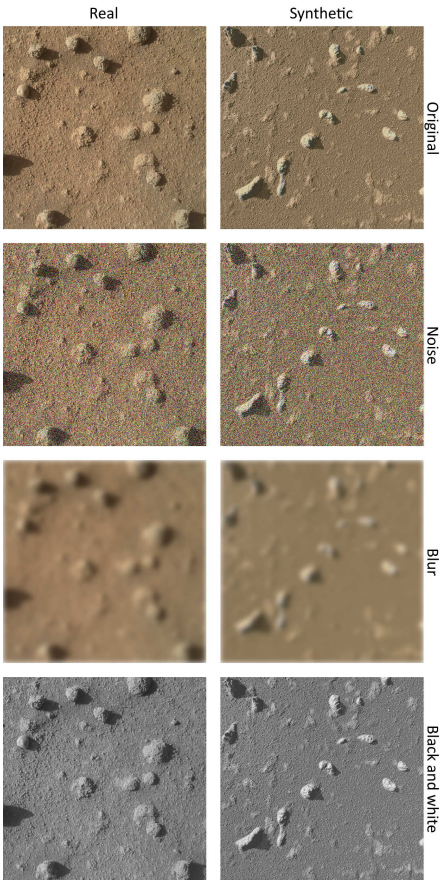


Figure 5.20: The right column is a synthetic rendering of the mars surface. The surface is rendered using noise functions. The left column is of a real image from the surface of Mars. Each row is different pre-processing filters added to both images

processed the same way. This pre-processing step reduces the variance of both distributions, reducing the non overlapping parts. See figure 5.20 as an example of such pre-processing.

As we can see from this rendering, is the images look more similar the more pre-processing is applied. As long as the features are still recognizable by the model, this will make the model better at fitting both real and synthetic data.

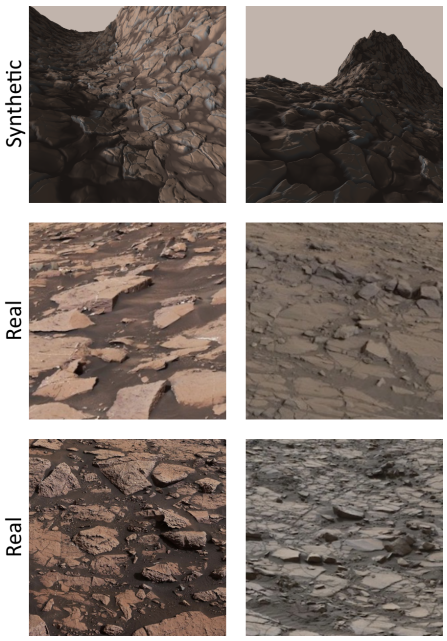


Figure 5.21: The top row show synthetic renderings. The second and third row show images taken by the Curiosity Mars rover.

Textures A one problem using synthetically generated images becomes apparent in figure 5.21.

In this figure, we see an example of great variation within a single type of geological feature, both in terms of color, and in terms of texture. Creating code for generating all these types of features, could take long to do. Another strategy is to use textures from the real training data and use this to generate synthetic data. Step one is then to extract textures from all variants of each of the features in the training images. The next step is then to wrap these textures onto 3d geometry and render the geometry in different angles, light settings, and weather. The problem with this approach is two-fold. The first problem

is that re-using textures could lead to the model over-fitting specific textures instead of learning the more general patterns in these textures. Even if the textures used are flipped, rotated, color-shifted, blended, and warped due to differences in perspective during rendering, there is still the same patterns. A way of circumventing this problem is collecting a larger number of varied textures, which leads to the second problem. Collecting all these textures is taking as much time as it would simply manually labeling the images. Doing manual image segmentation plus some image augmentation, could might be a more efficient approach.

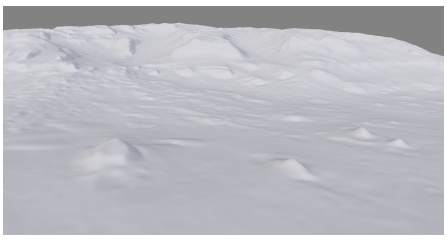


Figure 5.22: 3d rendering of parts of the Gale crater, where the Curiosity rover landed, based on a height-map generated by pictures taken by the Mars Reconnaissance Orbiter using its HiRISE camera. Height-map obtained from . NASA/JPL/University of Arizona. The height-map is rendered using Blender[Ble19]

Potential ways of salvage The main problem is capturing the great variation found in the real data. After fiddling with parameters in combination with a sufficient number of textures, one can make terrain looking pretty similar to the real ones, but we might still encounter real data that looks nothing like the synthetic data. It is not sufficient making synthetic data that is similar to the real, if there is real data not looking sufficiently like the synthetic, but getting there is simply a matter of effort. Once a terrain generator is created, the number of training images with pixel perfect labeling is unlimited, which may require less effort than the manual labeling required to achieve the same level of accuracy. Since one of the goals of this writing is exploring the possible advantages in splitting complex models with small size of training data into multiple parts as a way of increasing the amount of target information, a manual segmentation approach is instead adapted for the segmentation model.

One way the rendering could still be of use is taking advantage of a renderer's ability to generate 3d geometry from certain perspectives. The features we focused on detecting in our approach is low level features, like sand, bedrock, sandstone, regolith, and so on. Larger scale features like layered strata, mountains, mounds, and planes is something that would be recognized based on their shapes, made out from a certain perspective based on some lightning. This is an approach yet to be explored.

5.8 Manual segmentation

The final approach to creating the segmentation model was making one that learn features of images from manually labeled training data. As can be seen in figure 5.1, the segmentation model is part of the full captioning model. Its potential purpose is twofold.

The first purpose is to combine it with a pre-trained convolutional network to do the segmentation. Then after the segmenter is trained on the manual data, it is fine tuned. Test data is used to ensure the model does not over-fit. The fined tuned model is

then inserted to the SCOTI captioning model to see if a pre-trained model, fine tuned on Mars data, performs better than one that is only trained on ImageNet.

The second purpose to be explored is one where the result of the segmentation is itself used in the full segmentation and captioning model as seen in figure 5.1.

Let us start going through a high level of abstraction data pipeline.

The first step The plan is to first manually create masks segmenting the training images into different geological features. Such features is for example regolith, bedrock, layered, sandstone, sand, and so on. Some of these features can also overlap. For example, you can have bedrock, layered bedrock, sandstone, or layered sandstone. To know what features should be given focus, the captions and image data is explored. The most common features are the ones prioritised for the segmentation. The number of images is large, about 3000. Some of them are croppings of each other, removing those there is about 1000 left. This is still too many to segment, so what images are segmented must be prioritised.

The second step The second step is creating a segmentation model using the images as training data, and masks as target. Different architectures is explored, but is inspired by the u-net architecture, as segmentation requires understanding of low level and high level features of the image to correctly classify the different parts of the image. Compare this architecture with our earlier approach doing unsupervised segmentation, see section 5.5, where we tried doing segmentation extracting only low level or high level features for our segmentation, using both has the advantage of classifying parts of the image looking at both low level features and large level features at the same time. If we were to use an u-net architecture while working on the unsupervised image segmenter, the model might take advantage of whatever information available doing edge detection lowering the loss function instead of actually separating geological features. Which is why a u-net architecture would not work in that case. In this case the model is not told to simply segment the image with correlation between close by pixels regarding the class it belongs to, but classify the pixels by a manually chosen class. Giving the model more information in this case does not open up for the models ability to cheat, as in the unsupervised segmentation model, but instead provide our model more information to use to understand the observed geological features in the image. Of course a more complex model does open for the ability to over-fit, but this problem can be detected by using test data, and battled using regularization techniques.

The third step The third step is either fine tune a pre-trained convolutional model based on the second step which is then used for captioning, alternatively see if the segmentations from the segmenter themselves can be used for captioning. Different variations of captioning models are tested and tweaked to better accuracy and prevent over-fitting.

The fourth step The fourth step is to encode the the segmentation of the image in some format decodable by the captioning decoder. The caption is then judged using a loss function. Different loss functions were tested, with different performance.

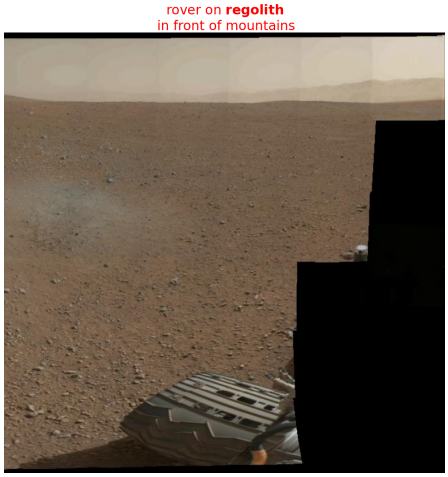


Figure 5.23: The text is the caption describing the content of the image. The bold word is the caption searched for.

Data exploration

The data consists of 3,067 training images and 25 test images. Each image has one caption describing the image. Many of the training images is various croppings of larger resolution images, or of the same motive with different panning. This leads to overlapping information in the images.

For the manual segmentation, masks is created segmenting the image by geological features. The training images are explored looking for what features are more common and what separates them from each other.

All the training captions are separated into n-grams of different sizes. Images containing these n-grams are visualized like what can be seen in figure 5.23.

The most common n-gram of size 1 is "regolith". The images containing "regolith" is visualized to get a sense for how to best segment the feature by the manual masking and by making a segmentation model that can learn the feature. For the most common features sorted by frequency, see figure 5.24.

As we see in this figure, the top 30 frequent n-grams cover most of the occurrences of features in the caption. The most common frequency by far is 1. To achieve proof of concept regarding this writings research goals, only the most frequent features are accounted for. This saves a lot of time. The features is chosen amongst the most common n-grams of all sizes. The most common n-gram of size 1 is *regolith*, with a frequency of 1624 occurrences in the 3,067 training captions. For n-gram size 2, the most common n-gram is "float rocks", see figure 5.25

It was also noticed during the exploration, n-grams like *in front of* occurs frequently. This suggests it is important not only to detect geological features in images, but remember where in the image the features are found so the captioning part of the model can describe their relative positions.

VIA-image segmentation

VGG Image Annotator (VIA)[DZ19] is a manual annotation software for image, audio and video. This is the software we decided on using as it supported polygonal masks with support multi class masking. The software runs in as an offline web application in the browser. We modified the software slightly to fit our specific task.

In figure 5.26 we see an example screenshot of VIA in use. The modification is seen on the lower right side of the figure, where the caption corresponding to the image currently being labeled is displayed with highlighted n-grams. There is also a search bar for determining the frequency of occurrences of substrings of all training captions. In figure 5.26 we see the searched string *regolith* occurs in the captions 1624 times.

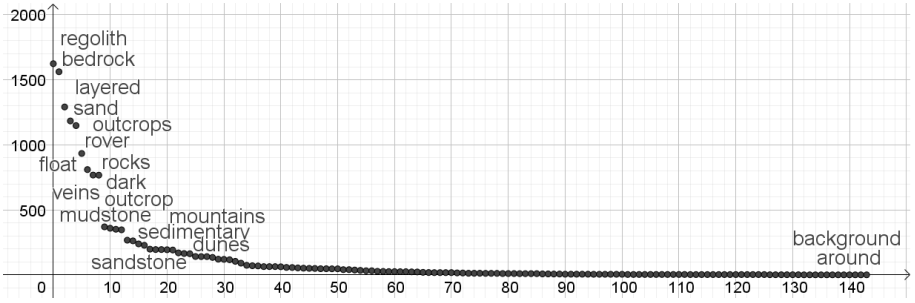


Figure 5.24: The most common n-grams of size 1. The horizontal axis is the words index sorted from most common (left) to least common (right). The vertical axis is the frequency of occurrence in the training data.

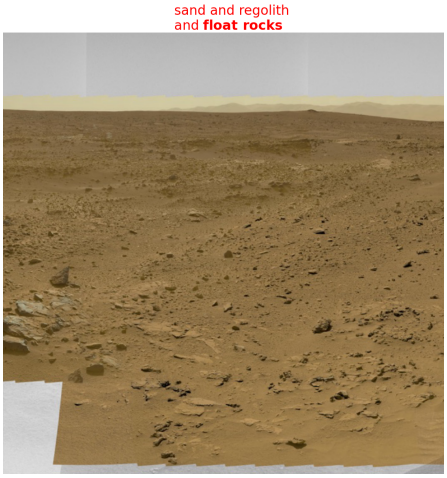


Figure 5.25: An example output during the data exploration with n-gram size of 2.

Too much data, too little time The number of images to segment using the VIA application is large. If it takes 5 minutes on average to label one image, labeling 3,067 images would take $5\ \frac{minutes}{image} \cdot 3,067\ images = 15,335\ minutes = 10.65\ days$. One approach considered reducing this time is using active learning. Active learning consists of training a model using a small number of training samples. The trained model then classifies the remaining training data. The data the model is the most uncertain of is given to the human label-er to label. The human labels some more images and the cycle repeats. This way, only the training data the model is in most need of is labeled first. This can lead to the human only needing to label 10%-20% as much training data [Dr 19]. The problem using active learning in this project is that the variance of the images is too big. Probably simply manually making sure the images labeled are varied enough is a sufficient approach. Probably taking advantage of the pre-trained model combined with a smaller number of manually masked samples than the number of images required for making that first model using active learning, is sufficient for creating a model with a good enough accuracy for proof of concept regarding our first research goal. It is expected that the number of manually masks required will be small enough that manually making sure they have a sufficient variation is manageable. If this is not the case, hopefully augmentation of the masked images (flipping, cropping... etc) will make up the gap.

To make sure as many ways of augmentation is possible on the manually masked data, mostly colored images

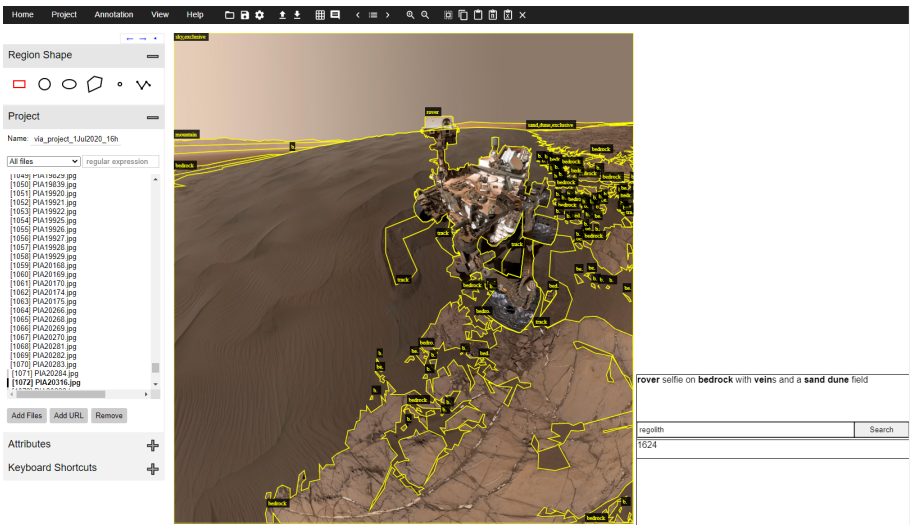


Figure 5.26: Screenshot of the VIA application.

were masked. We want our model to be able to detect features in both colored and gray-scale images. If a colored image is gray-scaled, we can automatically grey-scale it, doubling the training data. The same can not be done if we mask a grayscaled image. Some gray-scale images still had to be masked. This is due to some of the Rover cameras, like the Navcam images. These images might have different resolutions and compression rates. Some of these images are also labeled to make sure the model can also predict these kinds of images.

Image augmentation

As mentioned earlier, some of the images are croppings of larger versions of the images. The images chosen for manual segmentation are the uncropped versions. Larger images are better since this allow for more freedom doing manual cropping of the images, allowing for a greater number of croppings.

In figure 5.27 we see an example of the way the images are cropped.

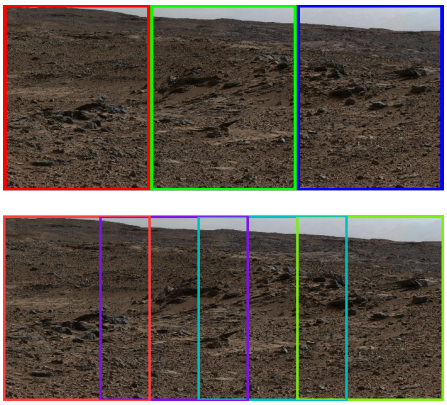


Figure 5.27: Two cropping strategies for image augmentation. The top image show non-overlapping three-fold cropping. The bottom one show cropping with overlap. The bottom image has space for one more cropping.

The top image in figure 5.27 is an in efficient version of cropping the image. The bottom way of cropping allows for one more cropping. Allowing for some overlapping of the images allow for the model to learn patterns crossing the borders. These dependencies would be lost using non-overlapping cropping. If the image to be cropped is tall, the overlapping cropping is also done vertically. If the image is of high resolution, which also typically means the motive is highly detailed, zoom is also applied. Zoom allow for even more croppings as more croppings can fit within the image. The overlap ratio used is about 1/3 (as in the example in figure 5.27) both vertically and horizontally.

Beside making the images gray-scale and cropping, the images are also flipped horizontally. Rotation is also augmentation that could be used, but the for-mentioned augmentation sufficiently increases the number of training data (from about 100 manually masked images to 6000) so that over-fitting becomes manageable.

Preventing test train leaking

During the splitting of test and training data, we made sure that a single image being augmented did not have variations of itself distributed amongst both test and train data-sets. This is due to the similarity between augmentations coming from the same image, and this will lead to over-fitting. We noticed this over-fitting when we saw the model sometimes predicting masks making the exact same errors as the errors within the manual maskings. Seeing this phenomena appear in test data, suggested test train leakage.

Multi classification There is many features to be detected in the training image. Examples are *bedrock*, *sandstone*, *outcrop*, *sand*, etc. Often the features are combined together, for example *layered bedrock*, *layered sandstone outcrop*, *crossbedded bedrock outcrop*, *sand dunes*, etc. If our model were to classify each pixel as one of all of these classes using one-hot encoding, the output would be quite large. For a 256 by 256 image, there would be $256 \cdot 256 \cdot number\_of\_classes$ dimensions to the output of the model. The target data would also be rather large. Multi class classification is used instead, allowing for combined classes. For example, this allow us to have simply one class for *sandstone*, one for *bedrock*, *layered*, *outcrop*, *crossbedded...* etc, instead of every combination, *sandstone*, *sandstone bedrock*, *sandstone bedrock outcrop*, *sandstone outcrop...* etc.

Segmentation model

The training data is stored in the format of tensors of image width by image height by number of classes. These tensors are fed to the segmentation model during training. The full segmentation model can be seen in figure 5.28.

The figure shows the configuration we ended up with at the end after exploring different variations and doing tweaking of those. The Full model architecture can be seen in figure 5.1. Figure 5.28 shows the segmenter part of that model. The inspiration for the model comes from the u-net model[RFB15].

Down stack

On the left side of the model we see the pre-trained convolution down stack. The model is InceptionV3 pre-trained on ImageNet. The reason we chose InceptionV3 net, is due to it being both accurate and has a fast infer-

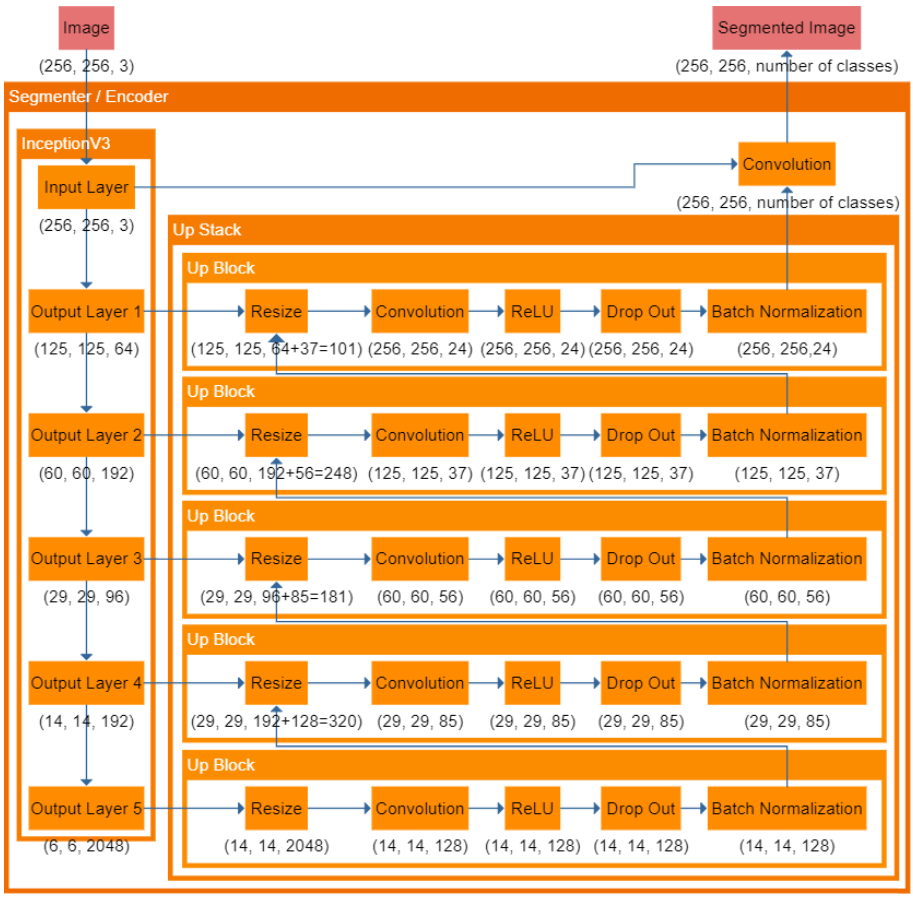


Figure 5.28: The final configuration of the segmentation model. The input is the image at the top left, light red box. The output is seen at the top right, light red box. Orange boxes represent functions taking tensors as inputs and outputs. The numbers below the boxes are the dimensions of the output tensors. The batch dimension is omitted.

ence[KUR20]. Fast inference is useful for testing during implementation. As discussed in section 5.2, inference time does not matter much on the space craft as long as it is lower than the transfer rate of the craft, which is the current bottleneck when it comes to data collection. A more accurate but possibly more expensive model is therefore better used on board the space crafts.

As the InceptionV3 model we used is trained on Imagenet, it has a head (single final dense layer responsible for the final classification classifying the images of the Imagenet model). This head is not useful for us, so it is removed. Slices of outputs of the model is extracted as outputs. The outputs are chosen at from the model at points just before the width and height of a layer is reduced, which also is the layers just after where multiple residual layers are concatenated in the InceptionV3 architecture. Another concern choosing what layers from InceptionV3 to

extract is making sure the dimensions of the layers are covering the entire spectrum of low level of abstraction features being extracted (typically those layers have a large width and height dimension, but a low number of filter dimensions), and high level of abstraction features being extracted (typically those layers have a smaller width and height dimension, but a larger number of filter dimensions). The reason we want to cover the entire spectrum is we want our segmentation model to to pick up on geological features that appear both as low level features and also high layer features, and sometimes combinations of those.

Up stack

On the right side, we see the up stack. This part of the model is trained from scratch. The down stack part of the model with its pre-trained weights are only made trainable during fine tuning. The up stack consists of *up blocks*. These blocks take in two tensors that are concatenated. The bottom *up block* takes the output from *output layer 5* from InceptionV3. This is the last output layer of InceptionV3 before the head (ImageNet classification dense layer). The output goes though resizing with bilinear interpolation and a convolutional layer of kernelsize 3. The idea of the *up stack* is for the segmentation to use both high and low layer features to determine the class of each pixel. The final output layer provide the high layer features, but the spacial resolution is low (only 6 by 6 width and height). On the other hand the final output layer has a large filter depth. The initial output layers has low filter depth, but has large spacial resolution. For example *output layer 1* is 125 by 125 width and height, but only 64 depth. We want to combine all this information for each pixel, but stacking all the filters of all the output layers at the target spacial resolution will lead to an enormous tensor. The convolutional layer in each *up block* allow for selecting the feature depth. The convolutional layers kernel size of 3 also allow for some logic. Since higher layer of abstraction features typically take up more physical space (and thereby a larger portion of the field of view of the picture and then number of pixels on the screen), one might thing the kernel size of the convolutions of the bottom of the up-stack should have a larger kernel size to capture these larger scale features. This is not necessary as since the width and height resolution is smaller, the same sized kernel size will cover a portion of the image invertedly proportional with the image horizontal or vertical resolution. Different sizes of kernel sizes were tested. A too large kernel size lead to overfitting, a smaller kernel size lead to under-fitting. Kernel size of 3 minimized the test error.

Fully connected block

The bottom part of the down stack has width and height of size 6 and 6 (see bottom left of figure 5.28). This output is resized to 14 by 14 and moves through a convolutional layer in the first up block (bottom up block in figure 5.28) with a kernel size of 3. This means that in the up stack (the part that is trainable before the fine tuning), the tensor values will only have an effect on pixels 1 step apart. This makes it so our network has never a full communication between pixels too far apart. This makes it so the classification of pixels will only depend on an area roughly $frac16$ of the image width/height in size. The size of the croppings are large enough, that this a size of area should be sufficient to recognize geological features. Having a fully connected layer at the bottom of the down-stack, would allow for combining

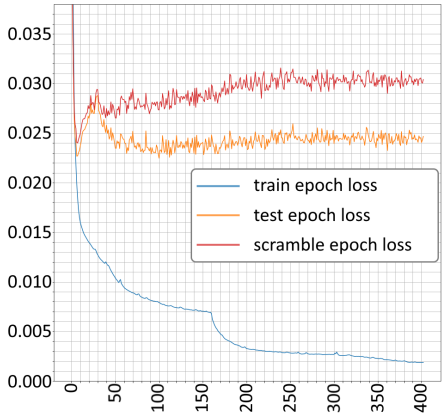


Figure 5.29: Graphs showing the MSE test losses during training when the segmentation model uses a fully connected layer connecting the down-stack and up-stack. See figure 5.28 for the segmentation model (not including the fully connected layer).

information regarding all the higher level of abstraction features, but would also open up for more over-fitting. Limiting the effect some pixels on one side of the image influences the classification of the pixels on the other end of the picture, will separate the segmentation logic to limited areas. If some feature on the left side of the image is used to recognize one training sample used for over-fitting, the right side of the image would not be aware of this feature. Limiting the reach of information, leads to a better test loss. Experiments using a dense layer between the down-stack and up-stack resulted in a lowered higher test loss. See figure 5.29. The scramble loss shows the models ability to fit the distribution of data. See subsection Scramble loss for explanation of this metric. As we see from this model, including a fully connected layer, significantly lower the test score, barely being better than the scramble loss. This block was omitted from the final model.

Up block logic

Without the ReLU function, the Up-Stack would be nothing but a semi connected layer, that is, it is a linear combination of the output layers, with some of the factors being zero. The ReLU add non-linearity improving the test score.

Multiple convolutional and ReLU layers interlaced did also not improve test error, but rather caused more over-fitting.

Batch normalization is put at the end of each block, stabilizing the layers to prevent exploding or vanishing gradients and speed up training[Kur18].

For each *up block* the data propagates, lower and lower level features is concatenated with higher level features. The last convolutional layer in figure 5.28 combines all the high and low level features for the final classification. The input image is concatenated before the final convolution to make sure the final convolution is aware of the original image's color in case this information is lost in the *down stack*.

Adding a dropout layer turned out to be a significant improvement to the test loss by reducing over-fitting. The minimum test loss went from 0.01014 to 0.007375. As suggested in this paper [PK16], we added dropout after our ReLU activation function. We used stochastic dropout with a rate of 0.1.

The test loss is the mean square error of each predicted class and actual class weighted by the relative area covered by that class in the training data. The presence of a class in a pixel is represented by a value 1 and not percent is 0. The predictions are decimal numbers between 0 and 1.

Class weights

The need of the class weights became apparent after observing the following problem. Since the loss function re-

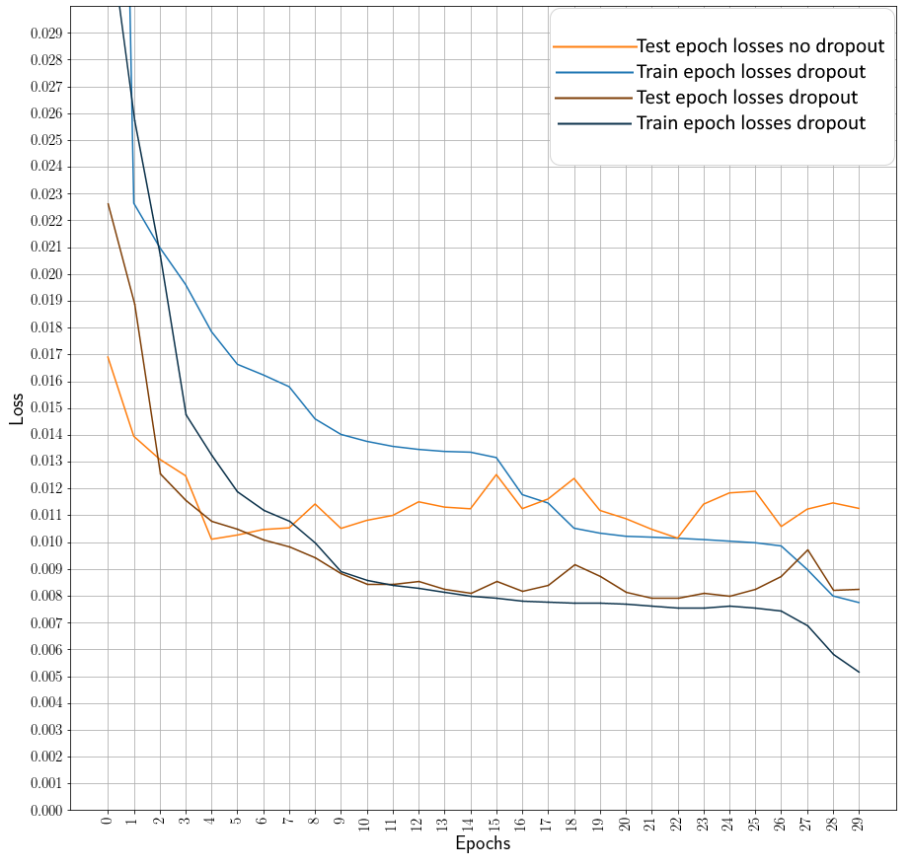


Figure 5.30: Graph showing the MSE loss with or without dropout.

warding and punishing based on square error per pixel, classes that generally cover a small area, will contribute less to the total loss. An example of such a small class is parts of rovers. See figure 5.31. Rover parts are only present in some images, and difficult to classify since there is such a large variation in colors and shapes. A class that is much more common, is regolith or sky. If the model, instead of spending effort predicting a class of low occurrence, fit the distribution of occurrence of the class (guessing the expected average pixel value of a class), the extra loss is small. This leads to the model focusing on features of a large area rather than on features important for captioning. To overcome this problem, the total area of each class is summed, and a weight to the loss of each class invertedly proportional to the size of the class is factored in. For example, if there is only three classes, A, B, and C, with respective areas of 1, 2, and 3, the loss for each respective class will be multiplied with $\frac{1}{1+2+3} \approx 0.167$, $\frac{2}{1+2+3} \approx 0.333$, and $\frac{3}{1+2+3} = 0.5$. These factors make the model balance the weights for the classes better.

Rare classes

Some classes are very rare. Not just in area, as talked about in subsection

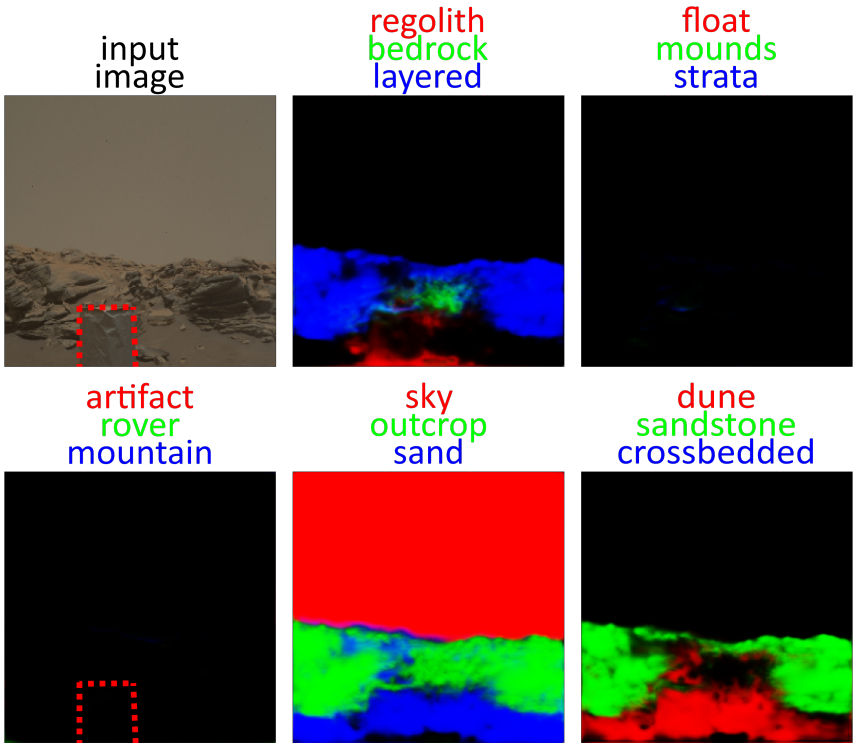


Figure 5.31: Illustration of the problem of rarely occurring classes. In the example, the input image is to the top left. The colored text above each image shows the color code of the features detected in the image. For example, the top mid image detects *layered* (blue). The model detects bedrock sandstone outcrop, sand, and sky. What the model fail to recognize is the rover. The striped red box shows where the rover should have been detected.

Class weights, but in occurrence in images. The few training samples makes it so there is very difficult for the model to recognize these features. Any effort by the model to lower the loss regarding miss-classification of these features, is simply over-fitting. Some classes were so important, more manual labeling were done on training images containing these classes. The classes that were too rare, even in the unlabeled training data, were ignored as they only introduce noise. For the model to recognize these geological features, more training data is needed.

Over-training

It is common practise to apply early stopping as a way to prevent over-fitting leading to the model fitting individual samples in the training data rather than the dependencies in the data. The over-fitting can hurt the models ability to fit the dependencies. The goal of early stopping is stopping the training just when the test data reaches its minimum. At this point the models prediction ability is not hurt more by the over-fitting than its prediction ability obtained from the training at this point. A common approach for detecting such a minimum is stopping

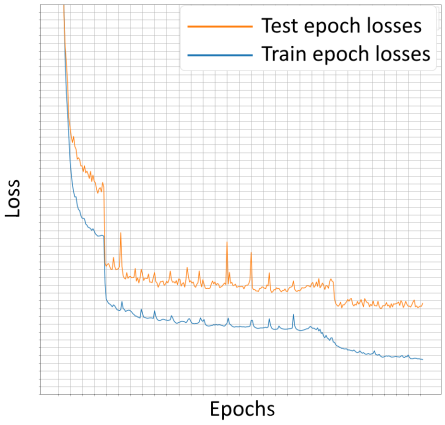


Figure 5.32: Example of the model reaching a plateau, where the test error stops improving for a while before making progress.

training when there has been a successive number of test errors without improvement to the score. The problem with this approach, is sometimes there could be periods where the model reaches a plateau in the loss landscape where little to no improvements are made before a sudden improvement is made. Depending on the time spent on this plateau and the number of successive steps required before the model is automatically stopped, the model might stop learning too early. See 5.32 for an example of this problem occurring.

Another potential problem unsung the successive steps of no improvement threshold, is the model will not stop learning before a few steps after the optimum. The approach we choose to avoid the above mentioned problems is saving the model weights every time the test error is better than any test error in epochs before that. The advantage using this method is we can train the model for as long as we want without the problems of overtraining because after the training, the models weights are reset to the last checkpoint. A potential problem using this method, is due to the variation of the test score, it is likely that the model will checkpoint a moment the model just happened to fit the test score well. Due to the principle of regression toward the mean[Wik20], the test error we get at the end of the training is probably a little better than what the true performance of the model.

Test loss variance

Sometimes when training a model, the test loss stops improving, but does not get noticeably worse over time. Visually comparing the output of two models with similar test scores showed that the model trained for longer, had a more confident predictions. We worry that there is a possibility for the model to both fit samples and dependencies at the same time, leading to a test loss that does not improve or worsen during training. This over-trained model might have the same test score, but the variance in test error amongst the test samples might increase. If the test score is the same, we want a smaller test variance. See figure 5.33 for an example output of this metric. We can see that even though the test error does not change much, the variance increases.

Scramble loss

The test loss is used to validate the models ability to fit dependencies in the data. If the model can not map input samples to the target value either because there is no dependency between the two, or the model is too simple to fit the dependency, or there is some kind of training problem, the test error still drop usually drop during training. The reason for this drop is because the model fits the distribution of the data rather than the depend-

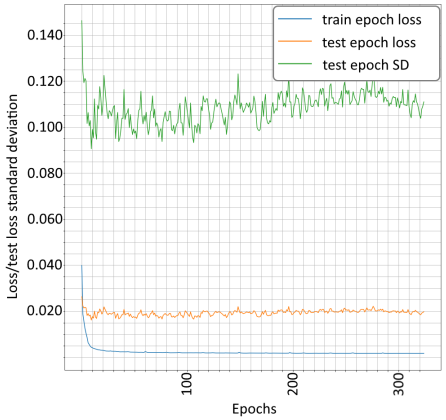


Figure 5.33: Graphs showing the training of a model. The blue graph show the training loss, the orange one shows the test loss. The green graph shows the standard deviation in the test predictions.

encies. For example, if $\frac{1}{8}$ of the area of the models is sky, one way for the model to reduce the loss without using the dependencies between samples and target, is simply guessing a $\frac{1}{8}$ probability of sky for each pixel (using MSE loss functions, guessing the expected value of the target distribution minimizes loss if there is no dependencies between samples and targets (the proof is trivial)). For classes more common, the probability would be set higher. Even better accuracy could be achieved by guessing with an even higher probability of sky in the top portion of the image, as this is usually where the sky is located. To capture the models ability to learn the models ability to fit the actual dependencies between input samples and target, a new metric was made. This metric calculates the test error, after scrambling the targets and the samples so that each sample, is associated with a random target (we also ensured no samples would be pared with the original target). We call this metric the scramble loss.

Optimization

Doing machine learning using images, tensorflow's data-set objects are useful for creating a data pipeline feeding the image data through the model during training. Since there us usually a large number of training images, it is common to store the data on disc, then pre-load batch by batch of training images before loading them on to the GPU for training. We noticed that using this approach, the beginning of each new epoch, there was a large delay before training of the first batch began. Further inspection found that more than half the time spent in the training loop was spent at loading the data from disc. Since the data is not pre-loaded at the beginning of each epoch, this resulted in a larger delay at the beginning of each epoch. Since our model has a small inference time, and the size of the training data is not too large, we moved the training data to memory instead of storing it on disc. Image data sent through machine learning models is usually stored as 32 bit floating point values. Compressing this to a single unsigned integer of size 1 byte, we could fit all the training data in memory. As casting the data back to a 32 bit floating point tensor takes way less time than loading the data from disc, we were able to increase training speed by about 2.5 times.

Fine tuning

After the model was sufficiently trained using our final architecture (See figure 5.28), using a pre-trained down-stack of non-trainable weights, we freed the weights to fine tune the model. The fine tuning lead to a test loss drop from 0.007387 to 0.004573 (61.9% of initial loss). See figure 5.34 for the training process. The model was first trained for 300 generations, then the weights at the point of best test score was loaded

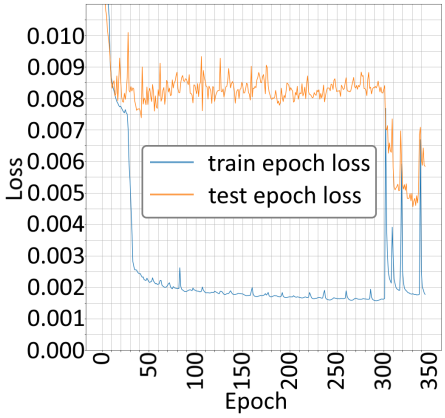


Figure 5.34:

into the model before the fine tuning began. After epoch 300 the model is trained with all weights trainable.

Results

Sorting out good test data is a challenge. As mentioned before, the initial split of test and train data was found to suffer from over-fitting due to identical motives and croppings of the same larger images can be found both in the test data and the train data. Also as explained in subsection Preventing test train leaking, methods were used to prevent this leakage. In case there still is some test-train leaks, we also test the model on images taken at Mars at a later point in time to validate there being no leakage. To visualize the results we used colors. If you look at figure 5.35, you see an example output.

In figure 5.35, we see 10 predictions. The images are random selections from the test data. Each image and corresponding predictions take up one row. All the predicted images are test images. Different images have different zoom levels, as they are croppings of larger images. At the top of figure 5.35, there is explanations of the meaning of the colors. For example, in the top left image (see figure 5.36 for close up), we see some sand and bedrock outcrop. To the right of this image, we see some of the target classes visualized (see figure 5.37 for close up)). The green represents bedrock, the red represent regolith (no regolith is classified in this image), and the blue represent layered (no *layer* is classified in this image). The image to the right of this image is the prediction (see figure 5.38 for close up). As we see, the prediction is pretty close. If you look at the image at the top, third most to the right (see figure 5.39 for close up), we see there is a prediction of rover in a place there is none. This could confuse the caption generator to believe there is a rover in this image even though there is none.

See figure 5.40 for the predictions done on the validation images. These images are not labeled. In this figure we can see some more classes than in figure 5.35. Some of these classes are less frequently occurring, and therefore the accuracy is lower.

5.9 Caption generation

The main challenge regarding the caption generation was the problem of over-fitting. Reasonable captions were generated, usually with the correct grammars and sensible syntax. The problem is the large difference between the performance of test and train data, or more specifically, the poor performance on test data. Different techniques were used attempting to reduce the over-fitting without reducing the networks ability to generate sensible captions.

The architecture

Figure 5.41 shows the final captioning architecture showing the flow of data (tensors) and the output dimensions of the tensors for each operation. The

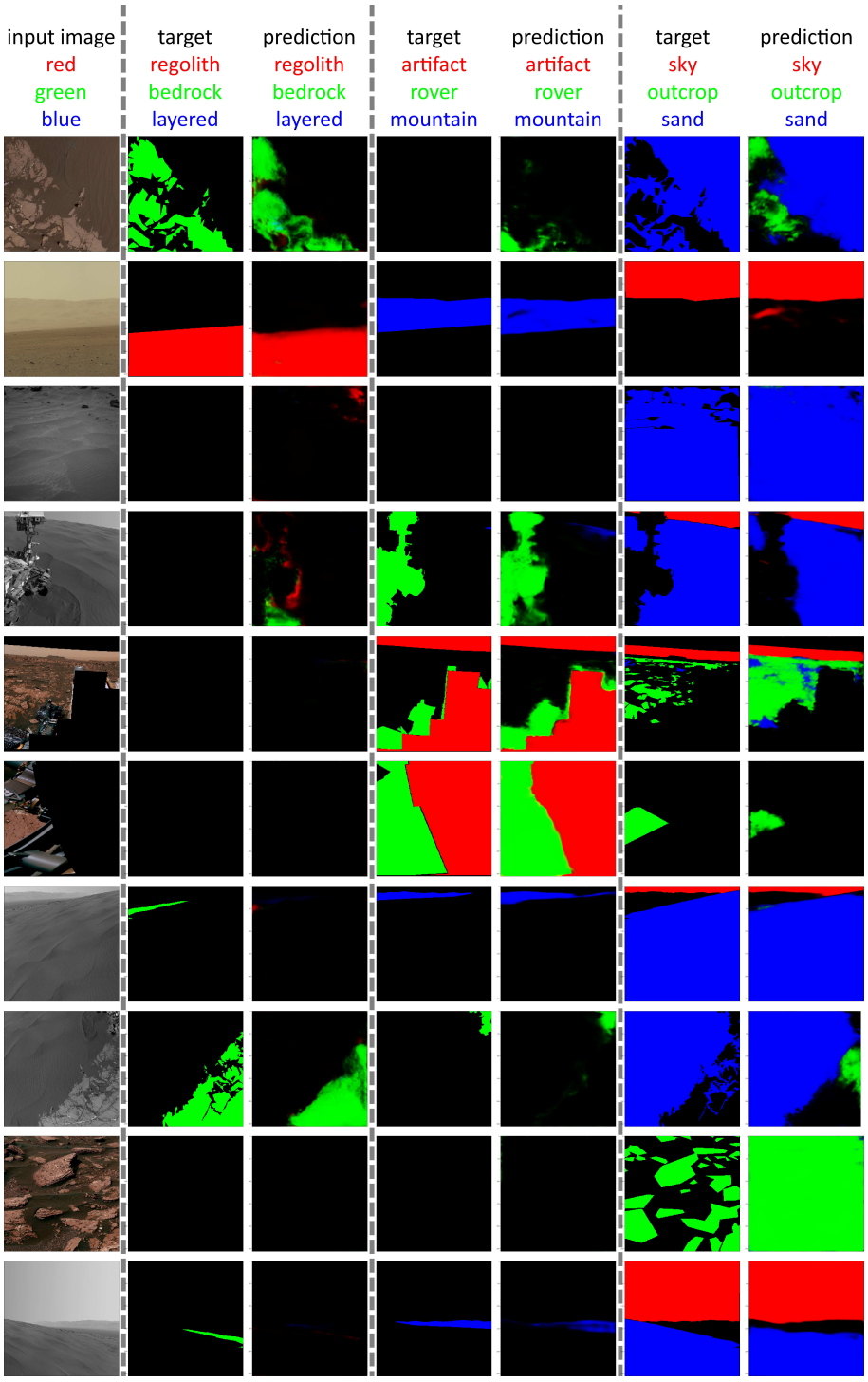


Figure 5.35: Visualized test predictions. Each row is one prediction. Each column shows three of the predicted classes in red, green, and blue.

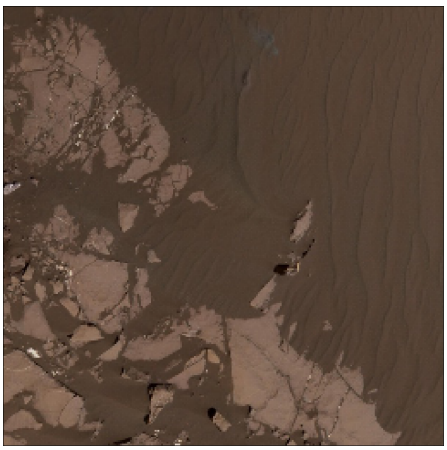


Figure 5.36: Input image.

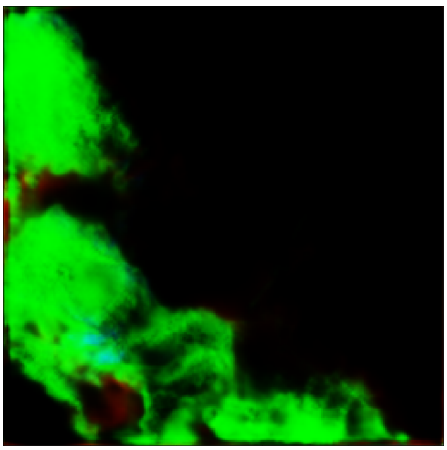


Figure 5.38: Predicted class (bedrock).

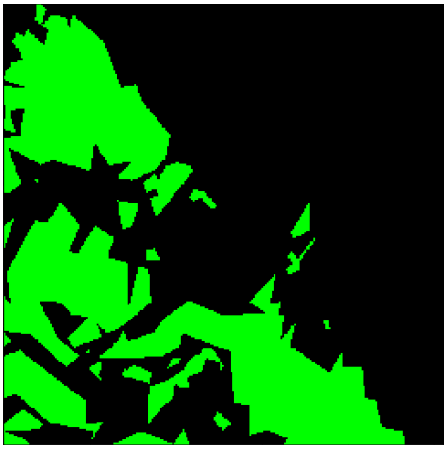


Figure 5.37: Target class (bedrock).

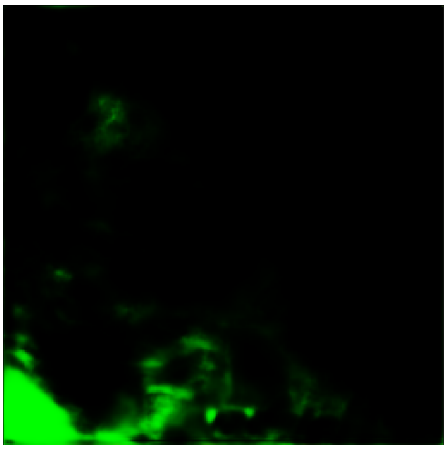


Figure 5.39: Predicted class (rover)

model is implemented in TensorFlow. Since adding a batch dimension to the tensor pushed through through the model, save performance (fewer iterations of the python code and fewer interactions between the CPU and GPU), then real tensor size of the data should have a batch dimension to them. For example, the input image should have a size of (batch size, 256, 256, 3) instead of (256, 256, 3), but since this in not important for explaining the logic of the model, the batch dimension is omitted. The caption generation algorithm takes in two data-points. One is the segmented image. The size of this segmentation is the same as the image in terms of width and height. The depth is the number of classes. The original color information of the image is not included in the segmentation. Therefore the original image is included and concatenated. This allow the captioning model to see the difference between things like dark sand versus just sand.

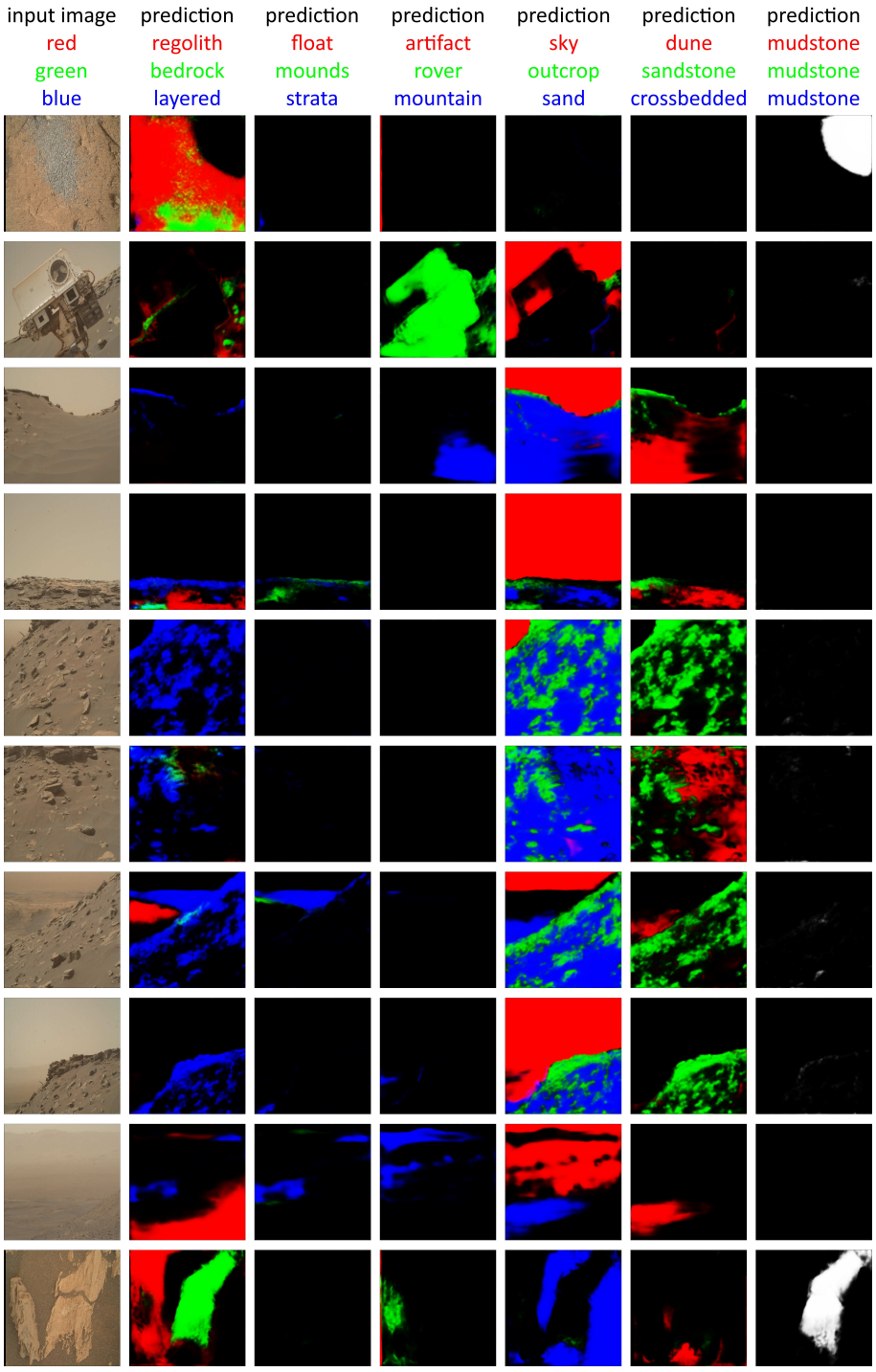


Figure 5.40: Visualized test predictions. Each row is one prediction. Each column shows three of the predicted classes in red, green, and blue.

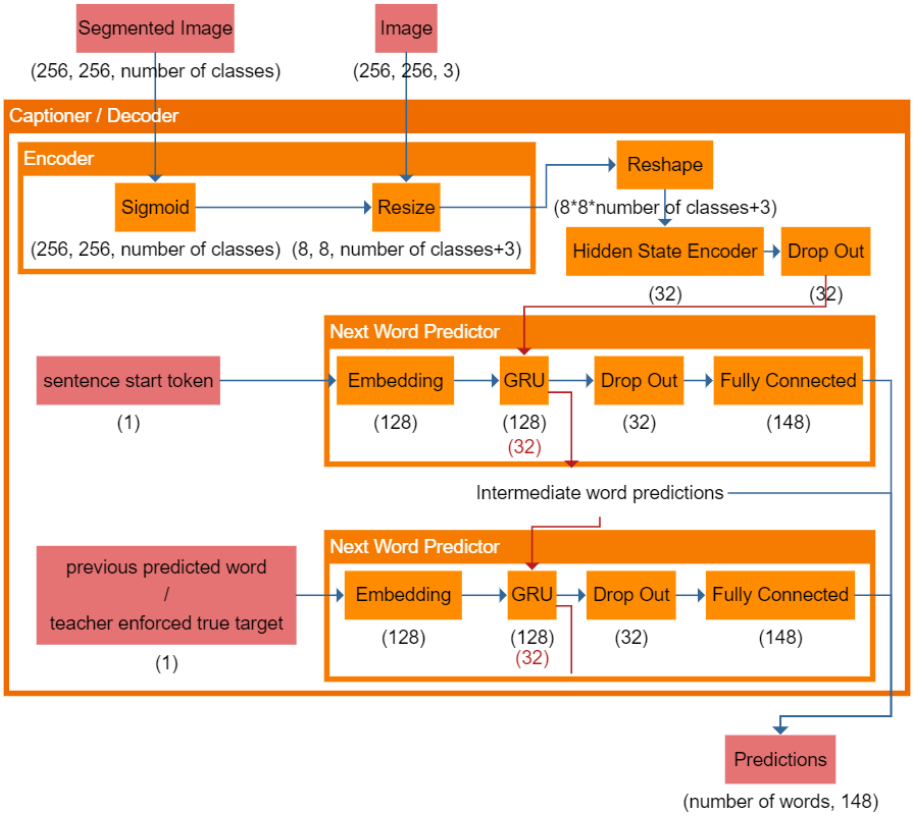


Figure 5.41: The final configuration of the captioning model. The input is the image at the top left, light red box, and the segmented image at the top middle, light red box. The output is seen at the bottom right. Orange boxes represent functions taking tensors as inputs and outputs. The numbers below the boxes are the dimensions of the output tensors. The batch dimension is omitted.

Encoder

The encoder block as can be seen in figure 5.41 must not be confused with the segmentation encoder. In the full model architecture as can be seen in figure 5.1, we see that there is a segmenter and a captioner part to this full model. Both has an encoder and a decoder. The job of the encoder (see *Encoder block* figure 5.41) is to create a tensor containing information of the image, including spacial information, used by the rest of the captioning model to create captions. When creating the encoder, different approaches were considered. The first approach were put the input image though a pre-trained image model and extract a layer from there. This layer is concatenated together with the segmentation before being sent of for captioning. Due to problems of over-fitting, the approach were simplified to simply concatenate the segmented image and the image.

Image encoding

For generating the captions we used a recurrent neural network (RNN) since caption generation require variable length output. To mitigate the vanish-

ing/exploding gradients problem, we used a GRU RNN. We also considered using an LSTM, but chose GRU due to its generality and simpler (and thereby faster) design. For the GRU to have access to the image information, the encoded image must be turned into a format the GRU can understand. We initially used an attention model based on the SCOTI caption generation [Ono+19] model. This model uses a version of Bahdanau attention for images. The images were encoded into a width by height by filter tensor. The width and height were small (8 by 8). That is, there will be 64 vectors describing their respective part of the image. Lets call each of these 64 vectors a tile. For each word the GRU predicts, it will use this encoded image. Based on the previously hidden state of the previous prediction, a context vector will be created from the attention given to each of the tiles. This context vector is combined with an embedding of the previously predicted word is put through the GRU model. The output from the GRU creates the hidden state for the next word to be predicted, and an output that is the currently predicted word.

Due to the problems with over-fitting, we also tried another method for encoding the image. GRU is an RNN capable of reducing the problem of vanishing/exploding gradient problem. For very long pieces of texts however, a GRU generating text based on some encoded input, the GRU might still forget some information. This is where the attention model comes into play. It allow for the generation of longer pieces of text without forgetting the initial context. For each prediction, the model is paying attention to the relevant parts of the context for the relevant predicted word.

Since the captions generated for the problem at hand, is not particularly long, we also explored replacing the attention model with an alternative way of feeding the encoded image to the GRU. In figure 5.41 we see that after the data is encoded the shape is 8 by 8 by number of classes + 3. The segmented image into classes and the original red, green and blue color channels are concatenated and ordered in $8 \cdot 8 = 64$ tiles. This structure is then flattened and encoded to the initial state of the GRU. Dropout is applied for further regularization. The *next word predictor* model is an RNN module containing multiple functions. The initial hidden state is propagated through the *next word predictor*. The idea is that the hidden state will both contain the information of the encoded image, and language related information keeping track of grammars and the current progress in producing the sentence.

RNN caption generation

The *next word predictor* model is an RNN with a hidden state, input, and an output. Each iteration of the rnn produces an predicted word.

During the first iteration, the hidden state contain the encoded image. This is a 32 dimensional vector. Besides encoding the image, there must also be room within this vector for the *next word predictor* RNN to remember language related information. Initially, we used an encoding of 128. This was reduced to 32 due to over-fitting. See subsection Test train leaks for more on the problems of over-fitting related to captions. During the first iteration, a *start sentence token* is embedded and used as the input to the GRU. Based on the hidden state vector (size of 32), and the embedded start token (size of 128), a prediction is produced. A prediction is a vector of size 32. Dropout is performed to reduce over-fitting before the fully connected layer creates a one-hot encoded prediction of the first

word. The dropout is applied before the one-hot encoding, and not to the hidden state of the GRU because applying the dropout to the hidden state can harm the GRU's ability for long term memory[G18]. The one-hot encoded words, are 148 dimensional due to the vocabulary is of size 147 plus the *end of sentence* token. After the first word is predicted, the next hidden state is transferred to the next iteration of the *next word predictor*'s GRU. Depending on whether the propagation of the network is part of the training forward propagation, or inference in production, will decide how the next iteration receives the previously predicted word. For more on this, see subsection Teacher enforcer. During inference, when the *next word predictor* predicts the *sentence end token*, the predicting is over. During training, when the training sentence has come to an end, the predicting is over.

The result of the captioner model, is a tensor of size *number of words* by 148. The first dimension represents the words in the sentence, while the second dimension represents the probabilities of each word in the vocabulary finding itself at that place in the sentence.

Loss function

To compare the correctness of the prediction with the target, cross categorical entropy loss is used. The more confidently the model predicts a sequence of words similar to the target, the lower the loss. A problem with this loss function, is it only compare word by word similarity. It does not compare context. If the true caption is "a rover on regolith" and the prediction is "rover on regolith", the loss will be large. The reason is that the loss function will compare "a" with "rover", "rover" with "on"... etc. The whole sentence is shifted, so even though the grammar and content of the caption is perfect, the large loss does not reflect this. We try different techniques to deal with this problem. See subsection Teacher enforcer and BLEU score / Jaccard Similarity.

Teacher enforcer

During inference, the *next word predictor* (figure 5.41) predicts words based on a hidden state memory, and embedding of the previously predicted word. For the first predicted word, a start token is used, as there are not previous words. The first prediction is a 148 (148 is the size of the vocabulary + 1 (the +1 is due to the *end of sentence* token)) sized vector with values representing the probability of each word. The word with the largest probability is chosen and fed into the embedding layer of the next iteration of the *next word predictor* model (see figure 5.41). The predicting is finished, when the *next word predictor* predicts the end of sentence token. The result is presented by picking the word with the largest probability for each prediction.

During training, instead of the *next word predictor* predicting words that are used as the *previous word* for the next prediction, teacher enforcing[Won19][20] is used. The teacher will replace the previous word prediction, with the true first word in the sentence. There is several advantages with teacher enforcing.

The first advantage is that it speeds up training. If we are not using teacher enforcing during training, we will have to use the previously predicted word as the *previously predicted word* for the next iteration. When the model is untrained, the first predicted word will probably be wrong. This will lead to the second word also being wrong, even if the model at this point had good weights. For example, let us consider an image of a rover on regolith. Let us say the caption is "rover on regolith".

If the first predicted word is "drill", it does not matter how well the rest of the model works, if the input is garbage, the output will be garbage. If we are using teacher enforcer, the first word "drill" will be replaced with the correct word "rover". This gives the model a reasonable chance at describing the rest of the caption.

The second advantage using teacher enforcing, is it helps reducing the problem described in subsection Loss function. If the true caption is "a rover on regolith" and the prediction would be is "rover on regolith", the loss would be large due to the loss function comparing word by word. When all the words in the sentence is shifted by one word, all the words will be considered wrong. The way teacher enforcing helps in this case, is when the *next word predictor* model produces the first word "rover", it will be replaced with the word "a". The next word predictor, will then know that following the word "a", the next word should be "rover". This makes it so the prediction is "rover rover on regolith". This prediction will have a relative small loss, leading to a model that would in production (and thereby not have teacher enforcing) predict "rover on regolith" when the true caption is "a rover on regolith", having a smaller loss. It will still be punished for the first word, even though is should not, but it is better than being punished for getting all the words wrong.

A potential problem with using teacher enforcing is the model becoming dependent on the teacher. The teacher enforcer ensures the model is stable by constantly correcting mistakes. During inference, this stabilization factor is not there, causing instability to the model (mistakes earlier can propagate though the network causing larger and larger mistakes). Since the model has no training being unstable, it is less able to stabilize. Another problem was encountered using BLUE score as loss function. See subsection BLEU score / Jaccard Similarity. for more information.

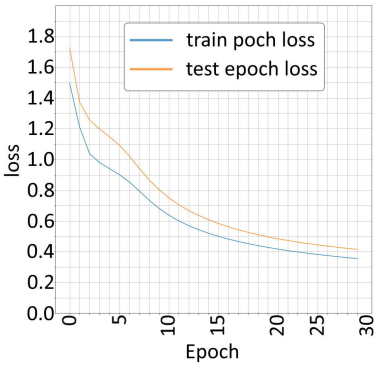


Figure 5.42:

Test train leaks

As with the training of the segmentation model, we sampled the weights whenever the test loss is the lowest during training. See figure 5.42

As can be seen in this figure, there does not seem to be much of a problem with over fitting. However, after visually inspecting the results, we see there is a problem.

In figure 5.43 we see a visualization of a test sample being captioned. The true caption and the predicted caption is identical. The problem is, the caption is too perfect. Not only are all features captured, but also the wording is identical. This suggests there is a test train data leakage. To check for this leakage, captions are generated by the model based on data taken by the Curiosity rover at a later point. None of these validation images can be found in the training data, so there should be no leakage. The captions generated is compared with the features found

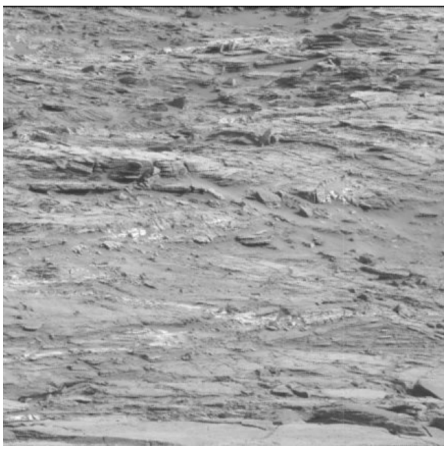


Figure 5.43: The true caption is the human generated caption describing the geological features in the image above. The predicted caption is the one generated by the model.

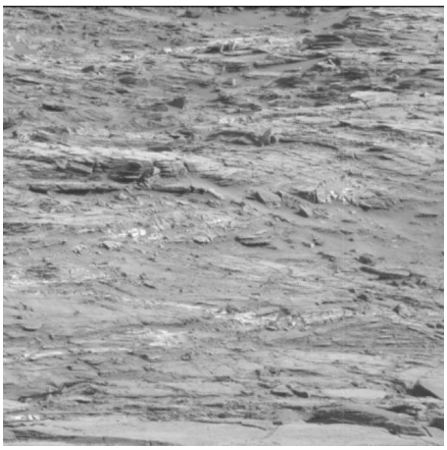


Figure 5.44: The true caption is the human generated caption describing the geological features in the image above. The predicted caption is the one generated by the model.

by the segmentation model. Since the segmentation model might contain errors, this comparison is not perfect. To properly fix the test train leakage, validation images must be labeled by human geologists. See figure 5.45 for an example of test captioning.

As we can see in this figure, there are captions that are suspiciously close to the true captions. Again, not just in content but also in wording. Let us now compare to the test data to the validation data.

If figure 5.46, we see some example images and their captions. We also see some of the segmentation models segmentations. The resolution is reduced to 8 by 8 to reflect the resolution received by the captioning model. See figure 5.41 encoder blocks resize function. The top row describes the content of each column. Each row represent each predicted caption. The leftmost column shows the generated captions, the next column shows the input images, the next ones show the segmentations. The meaning of the red, green, and blue colors can be read from the top row.

Comparing the image and the segmentations, we see that most of the images features are captured by the segmentation model. However, the captions does not reflect this well. For example, if we look at the top row of figure 5.46, wee see the image of a rover being captioned. The segmentation model recognizes the rover, however, the caption does not mention any rover. Comparing the captions gener-

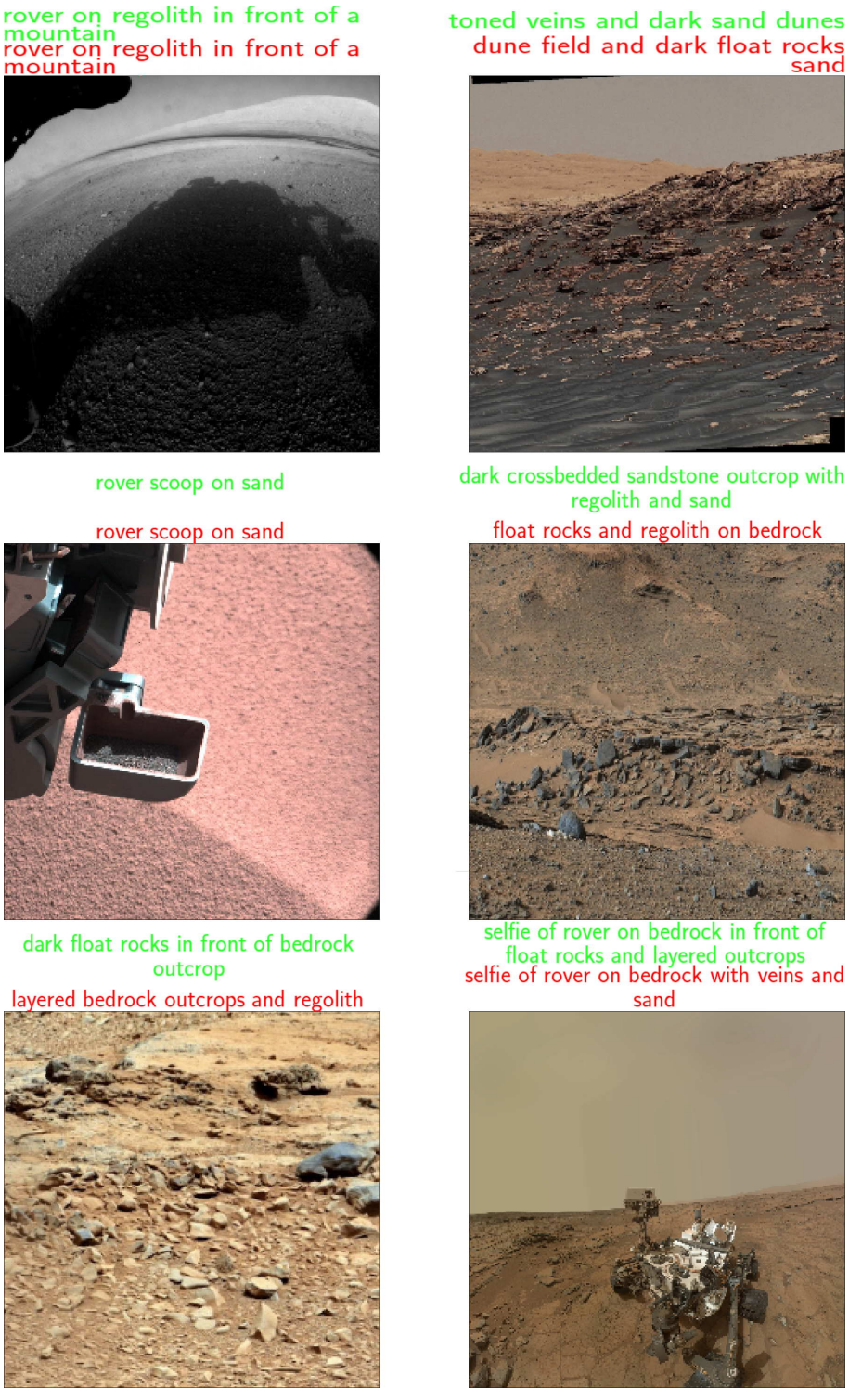


Figure 5.45: Green are true captions. Red captions are predictions.

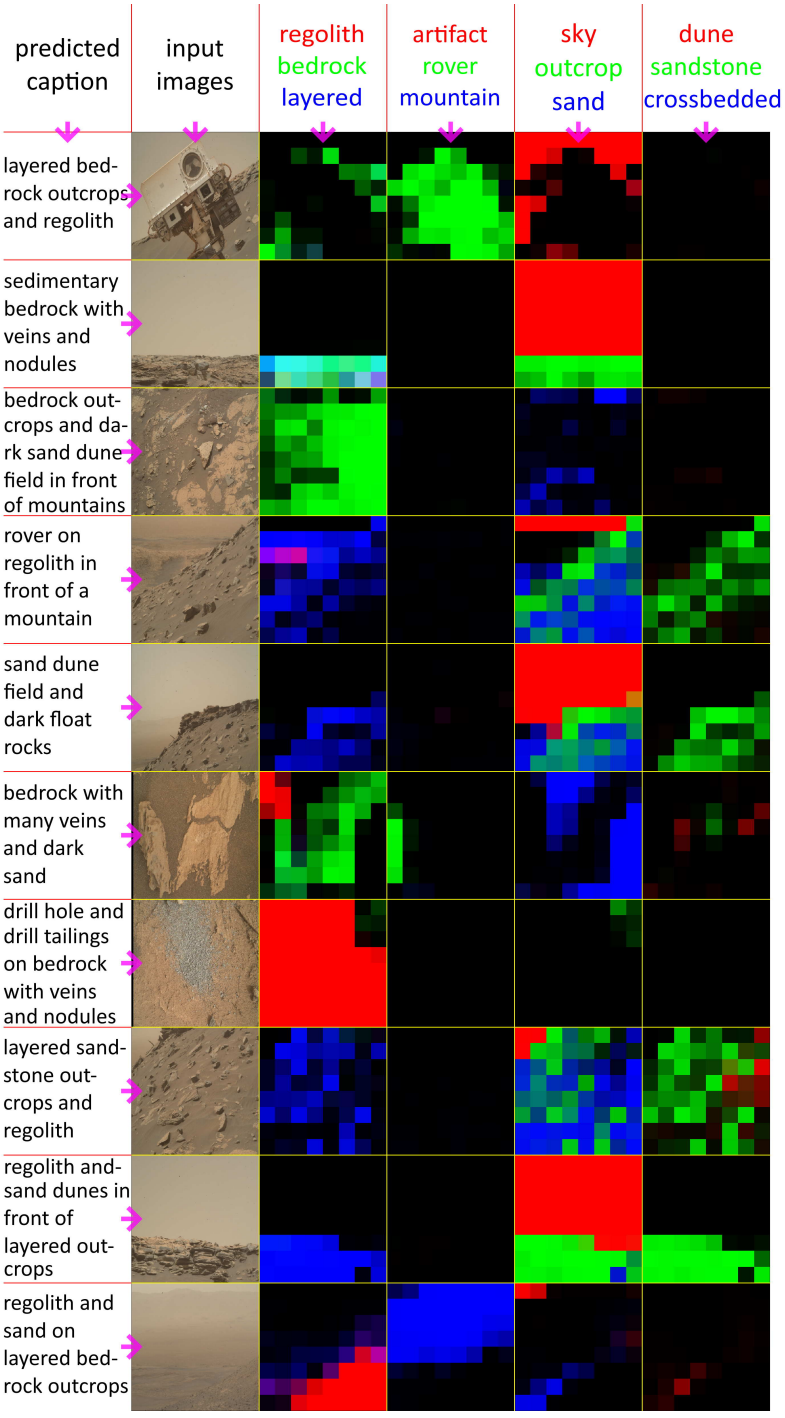


Figure 5.46: The left column is caption, the next is input image, the next ones are some of the segmentations.

ated on the test data (see figure 5.45 with the captions generated on the validation data (see figure 5.46), we see that our suspicion is confirmed. Due to the degradation of quality of the caption generation between the test and validation data, we conclude the test-train leakage is significant. To reduce the discrepancy, we introduce regularization techniques and manually compare the discrepancy to see if our model improves. This method is not efficient, but with the lack of labeled testing data, this is the best we can do. Further optimization of the captioner model, would require captioned test data not tainted by the train data.

One of the regularization techniques were to remove the attention model refereed to in subsection Image encoding. Instead encoding the image as the GRU's initial hidden state reduced the test-validation discrepancy. As mentioned in subsection 5.9, reducing the size of the hidden state vector, also reduced the discrepancy. After the discrepancy, we noticed the test loss increase, and the predictions becoming more directed towards the features the segmenter could detect. Figure 5.43 shows the prediction being identical to the true prediction. Figure 5.44 shows the prediction being more reflective of the features the segmenter is looking for (layered, bedrock, outcrops, regolith). In figure 5.43, alternation halos is mentioned, this is not a feature the segmenter is looking for. Reducing the size of the hidden state vector, leads to the features found by the segmenter being prioritised. See figure 5.47 for the captions after regularization.

As we can see in the captions in figure 5.47, the captions are much better than in figure 5.46. The segmentation model is the same in both examples, so the visualizations of the segmentations in both figures are identical. We see the improvement of the captions come from the captions being to a greater degree dependent on the features detected by the segmenter. The only caption that is not improved is the one 6th row of images. This caption changed from "bedrock with many veins and dark sand" to "rover on bedrock in front of layered bedrock outcrops". There is no rover in this image, so the caption is clearly wrong, however, when we inspect the segmentation, we see it actually classify the left part of the image as a rover. This shows the problem at this prediction is not with the captioner, but with the segmenter. Splitting the model into a captioning model and a segmentation model improve our ability to debug the models and isolate problems. This is in accordance with our second research goal.

BLEU score / Jaccard Similarity

Bilingual Evaluation Understudy (BLEU) score[Pap+02], is a method for evaluating the quality of machine translation. The score compares two snippets of text and based on a similarity quantify this similarity. A score of 0 means no similarity, while identical sentences would give a score of 1. A BLEU score is typically different from the loss function, as the loss function is differentiable and used in the back propagation, while the BLEU score is used to validate the result. A higher BLEU score comparing two sentences correlate well with how humans would rate the similarity of meaning between sentences. One idea we want to test, is if we can use BLEU score directly as a loss function, instead of using another loss function, and then tweak it and the model to make the BLEU score lower.

The main problem with using the BLEU score as a loss function, is the BLEU score is not differentiable (technically, all algorithms is differentiable, however, using the BLEU score as a loss function does not have a continu-

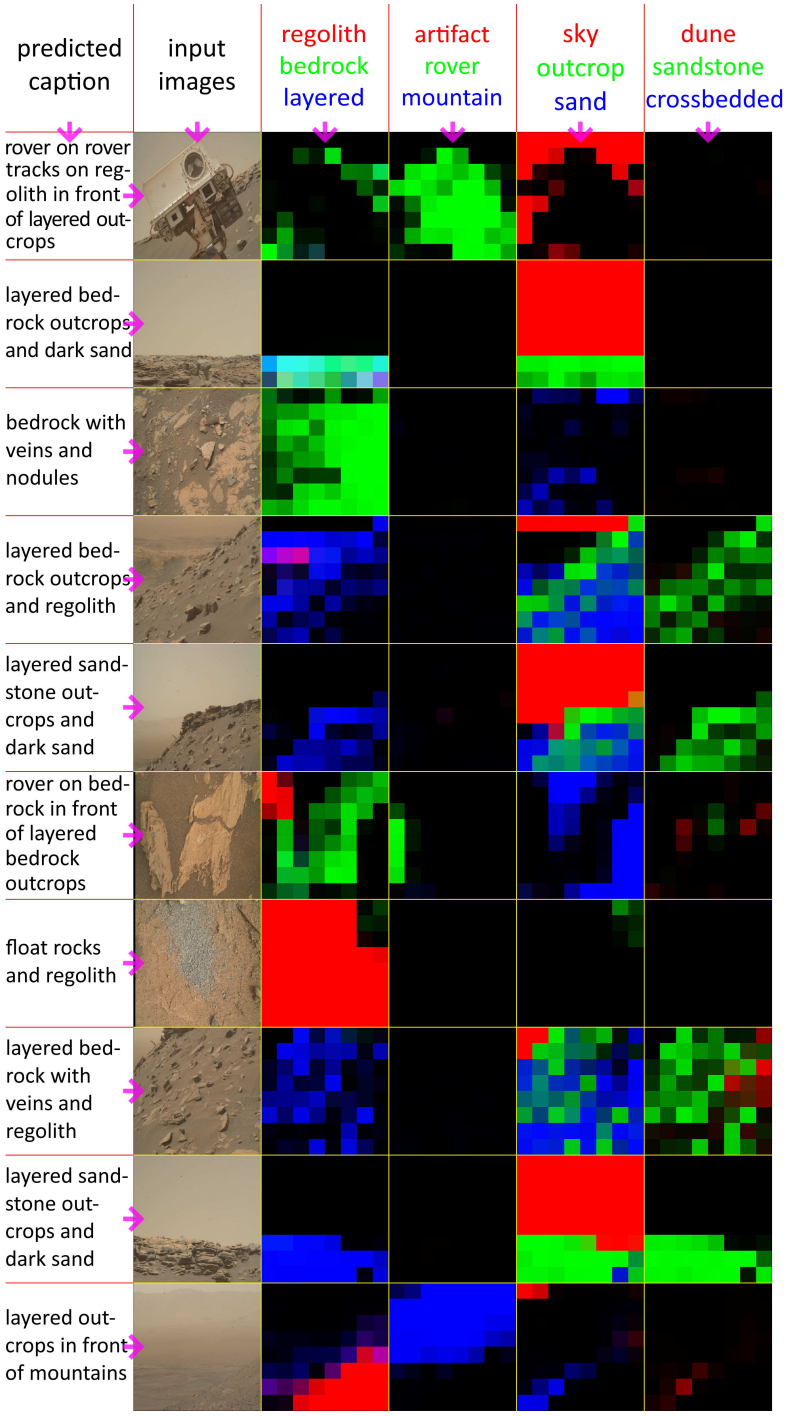


Figure 5.47: The left column is caption, the next is input image, the next ones are some of the segmentations.

ous gradient vector, and that is what is usually meant by a function not being differentiable).

Directly calculating the BLEU score function, leads to loss landscape with steps. Gradient descent does not work on such functions. To make the BLEU score smoother, we will instead of calculating the BLEU score for each prediction, we will calculate the expected BLEU score. One predicted caption for one sample is a $number\_of\_words \cdot 148$ dimensional tensor. See figure 5.41 at the light red box named *predictions*. The *number of words* dimension depends on the size of the sentence. The sentence stops when the *end of sentence* token is given the larges probability. The 148 sized dimension contain probabilities the model thinks a word should be. See example below.

$$\begin{array}{cccccc} \textit{<end>} & \textit{on} & \textit{rover} & \textit{regolith} & & \\ \left(\begin{array}{cccccc} 0.01 & 0.02 & \mathbf{0.90} & 0.01 & \cdots \\ 0.00 & \mathbf{0.95} & 0.00 & 0.01 & \cdots \\ 0.01 & 0.02 & 0.03 & \mathbf{0.87} & \cdots \\ \mathbf{0.69} & 0.07 & 0.09 & 0.01 & \cdots \end{array}\right) & \begin{array}{c} word_0 \\ word_1 \\ word_2 \\ word_3 \end{array} \end{array}^{4x148} \tag{5.1}$$

The matrix above shows an example *prediction* matrix. Each row represent a word in the sentence and each column represents a potential word. This matrix is 4 by 148 in size. The bold numbers represents the largest value for each row. The sum of each row is 1. The column indexed 0 represents the word the *end of sentence* token, the next columns represent the word *on*, the next the word *rover*, the next *regolith* and so on. If we select the words of largest probability, the sentence would be *"rover on regolith <end>"*.

Compared to the paper "A differentiable BLEU loss. Analysis and first results"[CFC18], we tried a similar approach. A problem they encountered is their predictions would generate a vectors of predictions averaging between the predicted n-grams. Splitting a sentence into n-grams, means grouping n and n number sequential words together. The BLEU score will be better the more of these n-grams two sentences share. An example of creating n-grams is as follows. Let us say we want to split the sentence "a rover on regolith" into n-grams of size 1 (monograms). Then we get "a", "rover", "on", and "regolith". Using size 2 (bigrams) we get "a rover", "rover on", and "on regolith". To avoid the same problems they encountered, we take a different approach making the BLEU score differentiable. Since our captions are pretty short, we only have n-grams appearing in the predictions once or more contribute positively to the score. This, plus some other configurations we made to make the score easier to calculate lead to our score being more similar to Jaccard Similarity. Hopefully this will lead to us avoiding some of the problems encountered in their paper. If we encounter similar problems, we will explore the possibility of combining the Jaccard Similarity with the cross categorical entropy loss to achieve better overall BLEU score. First let us go through our implementation.

The BLEU score is calculated by the following equation.

$$BLEU = BP \cdot e^{\sum_{n=1}^{N} w_n \cdot \log \frac{|TP_n|}{|TP_n|+|FP_n|}} \tag{5.2}$$

BP is a brevity penalty to punish too short predictions. w_n is the weight given to the nth n-gram. The weighted sum of N number of scores are added. T_n is the set of true positive n-grams of n size, in other words, n-grams found in both the prediction caption and target caption. $|TP_n|$ is the size of that set. FP_n (false positive) is the set of n-grams found in the prediction caption, but not the target caption.

The Jaccard Similarity is calculated by the following equation.

$$J_n = \frac{|\mathbf{T_n} \cap \mathbf{P_n}|}{|\mathbf{T_n} \cup \mathbf{P_n}|} \tag{5.3}$$

T_n is the set of n-grams of size n in the target caption. P_n is the set of the n-grams of size n in the prediction caption. J_n is the Jaccard Similarity for n-grams size n between the target caption and prediction caption. The Jaccard Similarity does not have a brevity penalty (BP). This penalty is used to punish short sized prediction captions. Without this penalty, the short predictions could get a large score even though they are not being similar to the target. The Jaccard Similarity does not have this problem as a short prediction would make the intersection between the target and the prediction much smaller than the union (see equation 5.3), thereby reducing the score.

To make the loss landscape using Jaccard Similarity differentiable, we interpret the prediction tensor (see figure 5.41) as an array of words of different probabilities of occurrence. If we look at the example prediction tensor 5.1, we see a sequence of words that strongly suggests the sentence "rover on regolith <end>". We want to reward this sentence (if this truly is the correct caption) when the model is as confident as possible with the probabilities corresponding to that sequence of words (the numbers in bold in equation 5.1). To achieve this, we will use the expected score as the loss function (actually $1 - \textit{score}$ as a loss should be minimized but the Jaccard score is larger (closer to 1) when similarity is larger).

$$loss = 1 - E(J_n) \tag{5.4}$$

If we want to combine multiple n-grams together we can sum the losses for different n parameters. During this derivation we will consider a fixed and single n-parameter to reduce notation clutter.

$$E(J_n) = E(\frac{|\mathbf{T_n} \cap \mathbf{P_n}|}{|\mathbf{T_n} \cup \mathbf{P_n}|}) \tag{5.5}$$

To simplify calculations, we will assume the gradient of the expected value of the right side of the above equation (5.5) is similar to the gradient of the expected numerator divided by the expected denominator.

$$\nabla E(\frac{|\mathbf{T_n} \cap \mathbf{P_n}|}{|\mathbf{T_n} \cup \mathbf{P_n}|}) \to \nabla \frac{E(|\mathbf{T_n} \cap \mathbf{P_n}|)}{E(|\mathbf{T_n} \cup \mathbf{P_n}|)} \tag{5.6}$$

Or at least the direction of the $\nabla E(\frac{|\mathbf{T_n} \cap \mathbf{P_n}|}{|\mathbf{T_n} \cup \mathbf{P_n}|})$ vector will be similar to the $\nabla \frac{E(|\mathbf{T_n} \cap \mathbf{P_n}|)}{E(|\mathbf{T_n} \cup \mathbf{P_n}|)}$ vector. Therefore we can simplify the expected Jaccard Similarity as in equation 5.6, and expect the gradient calculated during back propagation to be similar. Before we break down how to calculate expectation, let us take a look at some notation.

Consider the matrices TM_n and PM_n. TM_n describe the true caption. PM_n describe the predicted captions and its uncertainty. Each row in the matrices represent an n-gram in the sentence. Each column describe the probability an n-gram occupies that space in the sentence. See the below example of such matrices when n is 1.

$$PM_1 = \begin{bmatrix} 0.01 & 0.02 & \mathbf{0.90} & 0.01 & \cdots \\ 0.00 & \mathbf{0.95} & 0.00 & 0.01 & \cdots \\ 0.01 & 0.02 & 0.03 & \mathbf{0.87} & \cdots \\ \mathbf{0.69} & 0.07 & 0.09 & 0.01 & \cdots \end{bmatrix}_{4x148} \tag{5.7}$$

$$TM_1 = \begin{bmatrix} 0.00 & 0.00 & \mathbf{1.00} & 0.00 & \cdots \\ 0.00 & \mathbf{1.00} & 0.00 & 0.00 & \cdots \\ 0.00 & 0.00 & 0.00 & \mathbf{1.00} & \cdots \\ \mathbf{1.00} & 0.00 & 0.00 & 0.00 & \cdots \end{bmatrix}_{4x148} \tag{5.8}$$

In the example above n is 1. These matrices describe the probability of occurrence for each n-grams of size 1 at

each position in the sentence. n-grams of size 1 is simply words (or the end token), so PM_1 is the same as matrix 5.1. As in example 5.1, these matrices would describe the sentence "rover on regolith <end>". Notice the TM_1 matrix has only 1s and 0s as we know the target caption with 100% certainty. To calculate the probability matrices for n of a larger size, we will assume random independent selection of words (rows) based on the above matrix. Fox example, the probability of the "rover on" n-gram at the beginning of the sentence is $0.9 \cdot 0.95 = 0.855$. The probability of the "rover on" n-gram to occur as the second bi-gram is $0.00 \cdot 0.02 = 0$. The probabilities of the occurrence of these n-grams (n parameter greater than 1) will be dependent on each other. For example, if the first bi-gram does not contain the word "on", it is impossible for the next bi-gram to be "on regoilth". We will however assume independence as this simplify calculations. If we calculate these probabilities for all possible n-grams at all possible positions, we can create TM_n and PM_n matrices for all values of n. These matrices will quickly become very large. For example, n parameter of 2, the PM_2 matrix will have a number of rows of one less because the last word (the end token) does not have any other words to form a bi-gram with. However the number of columns will be the size of the vocabulary squared. Every word can be combined with every other word to form a bi-gram. n-grams of larger size will quickly be very large. Practically this is not a problem since the T_n matrix will be sparse, and we will use tricks so we don't have to construct the matrices.

Let us take a closer look at the numerator of equation 5.6.

$$E(|\mathbf{T_n} \cap \mathbf{P_n}|) = \sum_{i=0}^{N_{Pn}-1} prob(TM_{n*i}) prob(PM_{n*i}) \tag{5.9}$$

TM_{n*i} means column number i of matrix TM_n. The $prob(\cdot)$ function takes a column of a probability matrix and gives the probability of occurrence of this n-gram one or more times over the sentence. Given the probability distribution given by our captioning model, $prob(TM_{n*i})$ and $prob(PM_{n*i})$ are independent. The probability of occurrence of an n-gram in both target and prediction is simply the product of the two. As mentioned before, independence between occurrences of n-grams across the sentence will be assumed independent, so the multiplication rule can be used implementing the $prob(\cdot)$ function.

$$prob(PM_{n*i}) = 1 - \prod_{j=0}^{N_{PM_{n*i}}-1} (1 - PM_{nji} \cdot f_{nji}) \tag{5.10}$$

There is an equivalent equation for TM_{n*i}. PM_{nji} is the element in the PM_n matrix at row i column j. $N_{PM_{n*i}}$ is the size of the prediction sentence and number of rows in the PM_n matrix. f_{nji} is a probability factor describing the probability of n-gram size n, index i number j is reachable or if the sentence has ended yet. $(1-PM_{nji} \cdot f_{nji})$ is the probability of an n-gram not occurring at index j, $\prod_{j=0}^{N_{PM_{n*i}}-1}(1 - PM_{nji} \cdot f_{nji})$ is the probability of the n-gram not occurring at all in the whole sentence, and equation 5.10 is the probability of it occurring at least once.

f_{nji} is calculated as:
For $n = 1$:

$$f_{1ji} = 1 - \prod_{k=0}^{j}(1 - PM_{1ki}) \tag{5.11}$$

For $n > 1$

$$f_{nji} = 1 - \prod_{k=j-n+1}^{j} (1 - f_{1ki}) \quad (5.12)$$

Since $prob(TM_{n*i})$ from equation 5.9 will be mostly zero, we can implement this as a lookup of all n-grams in the target and sum together $prob(PM_{n*i})$ for all indices i. Now that we know how to calculate the numerator of the right hand side of equation 5.6, let us look at the denominator.

$$E(|\mathbf{T_n} \cup \mathbf{P_n}|) = E(|\mathbf{T_n}| + |\mathbf{P_n}| - |\mathbf{T_n} \cap \mathbf{P_n}|) \quad (5.13)$$

$$= E(|\mathbf{T_n}|) + E(|\mathbf{P_n}|) - E(|\mathbf{T_n} \cap \mathbf{P_n}|) \quad (5.14)$$

In equation 5.14 we now have three additives to calculate. $E(|\mathbf{T_n} \cap \mathbf{P_n}|)$ we already know what is. $E(|\mathbf{T_n}|)$ is the expected size of the target n-gram set which is simply $|\{TM_{n*i}\}|$. $|\{TM_{n*i}\}|$ is the size of the unique target n-gram set. The last additive, $E(|\mathbf{P_n}|)$, is trickier. This is the expected number of unique n-grams found in the predictions. Due to the large number of possible n-grams when n is large, and the large number of combinations of occurrences of these n-grams, we make a simplification. Instead of calculating the expected number of unique n-grams in the prediction, we calculate the expected number of n-grams in the prediction.

$$E(|\mathbf{P_n}|) \approx \sum_{k=0} (k+1) \cdot PM_{nkl} \cdot f_{nkl} \quad (5.15)$$

l is the index of the end token. $PM_{nkl} \cdot f_{nkl}$ is the probability of the size of the size of n-grams in the prediction being $k + 1$. The approximation we do where we count the number of n-grams instead of the number of unique n-grams will probably not harm the loss function. Our approximation will make the loss larger as the denominator will be larger (small score makes loss larger). When the model is performing well, it will predict captions with very few repeating n-grams, so when the model performs well, the approximation will be good.

Results

Teacher enforcing When applying the Jaccard similarity loss function together with the use of teacher enforcing, there is a leakage of target information. The following is an example output from the model:

Target word	Predicted word
2	5
5	4
4	9
9	25
9	10
10	13
10	7
7	8
8	11
11	3
3	0
3	0
3	0
3	0
3	0
3	0

The numbers represent words. The prediction seem good, just shifted a little. The number 3 represent the sentence end token. The test and train score of the model is good, and is improving over time. See figure 5.49. The problem becomes apparent when testing the model. See figure 5.50. The red captions describe the predictions. As we can see, they are way of. Looking at the table, we see the problem. The model has simply learned to predict whatever the teacher enforcer is giving it. This creates a good test score,

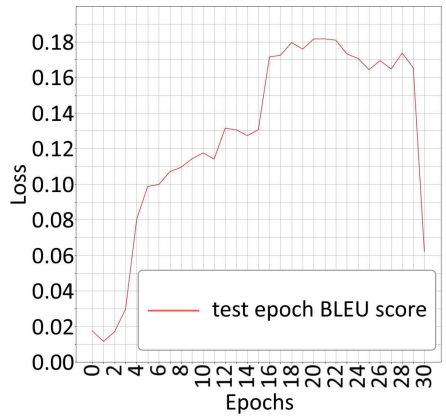


Figure 5.48: The loss during training using a combination of cross categorical entropy loss and Jaccard loss.

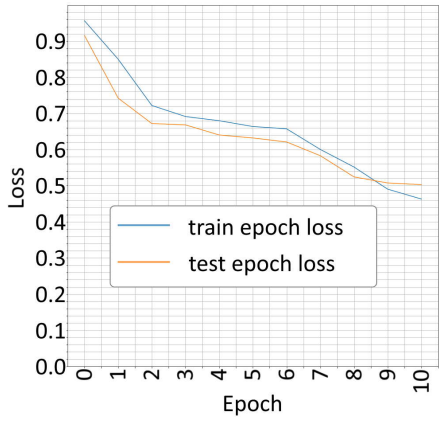


Figure 5.49: The loss during training using Jaccard loss function and teacher enforcing.

since Jaccard score support shifting of the sentence. When teacher enforcing is turned off when generating captions for the validation data, the model will be its own teacher. As can be seen in the table, the first prediction is wrong. After this first wrong prediction, the model relay of the teacher. In figure 5.50 the first wrong prediction happened to be "mudstone", after that it repeat after the teacher (itself). This result was predictable as it was encountered by the differentiable BLEU paper[CFC18]. Turning of teacher enforcing fixes the problem, but slows training substantially.

Average vector problem We hoped we would not encounter the same problem as in the differentiable BLEU paper[CFC18] regarding the model simply learning to predict the average of the most common words and n-grams. See figure 5.51 for example output. The model fits the distribution of data, not the dependencies. This problem was also expected as we didn't do anything fundamentally different from the implementation in the differentiable BLEU paper[CFC18].

Combined losses The forementioned problems were predicted. What we wanted to test, was to see if we could combine the cross categorical entropy loss with the Jaccard loss to improve the model. The resulting BLEU test loss can be seen in figure 5.48. The first 30 generations, the model is trained on the cross categorical entropy loss function. Then the model is "fine tuned" using the Jaccard loss function. As we can see, the performance of the model drops rather than improving. The reason might be due to some unexpected consequences of some of the simplifications we did during the derivation of the loss function, alternatively a way the model

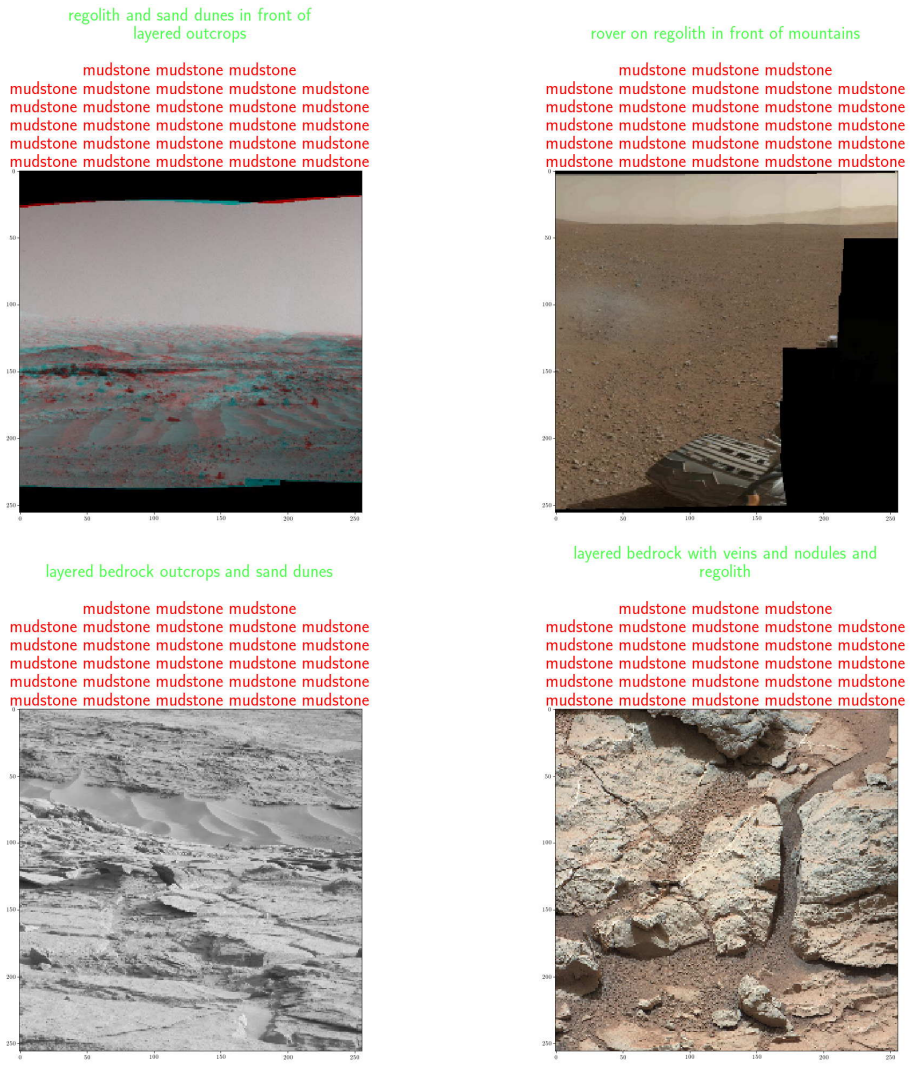


Figure 5.50: Predicted captions on the validation data after training using Jaccard loss function and teacher enforcing. Green captions are the target, red are the predictions.

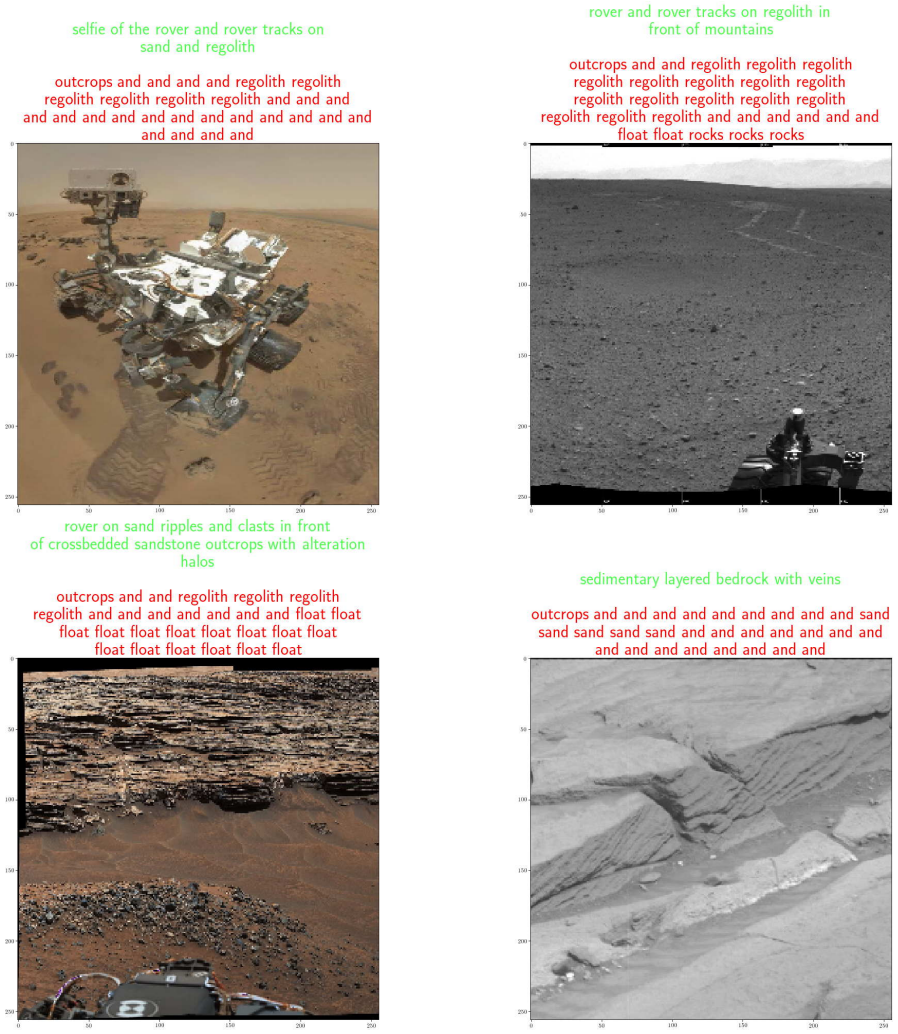


Figure 5.51: Predicted captions on the validation data after training using Jaccard loss function without teacher enforcing. Green captions are the target, red are the predictions.

tricks the loss function, as seen when using the teacher enforcer. Do to the tainted test data we have been working with, further fine tuning of the captioning model is difficult. The validation data is not tainted, but lacks captions, making fine tuning difficult as we have no numerical test value to improve, and must instead manually compare results which is inaccurate and time consuming.

CHAPTER 6

Conclusion

We set ourselves some research goals we explored in this writing. Some attempts we more successful than others, but learned something regarding each goal. Due to the discovery of test data tainted by train data, it was difficult to validate the success of some approaches. The validation data we acquired to avoid the taintedness, is not labeled. To further work on the approaches in this writing, labeled untainted data is needed.

Regarding our first goal, we did see find the splitting of the full captioning model into multiple parts beneficial. Due to the tainted test data, it is difficult to compare the results in a meaningful way. We did see the advantage of reduced over-fitting. The main split of our model was into a segmenter and a captioner. The captioner suffered more from over-fitting that the segmentation model. Had we not split the model, it would have been difficult to locate this problem, which brings us to our second research problem. Splitting the model allowed us to detect where problems were located more accurately in the model. See figure 5.47 row 6 (of images). As mentioned before, we see the segmentation model detecting a bit of rover in the image (there is no rover in the image), and the captioner mentions rover in the caption. The intermediate output allows us to know this is a problem in the segmentation model.

Regarding our third research goal, during the testing of our models, we used various testing metrics. Amongst them is calculating the variation or standard deviation of performance between testing images. This metric showed how even though the test score might not be changing much, the over-fitting of the model might still degrade due to over-training/over-fitting by increasing the variance in the test error. The scramble error helped us validate segmentations and captions for images, reducing our human bias. Seeing images of the Martian surface that has no vegetation, taken by a rover in similar areas, the similarity of images might make similar features be found in several images. This may lead to general predictions fitting to multiple images, and we as humans having confirmation bias might over-estimate our model's performance. Scrambling the test target and predicted test target, we can get a sense of our model's ability to not just fit the distribution of data, but to fit the dependencies. To what degree did our model predict "regolith" by chance, or did it actually see this in this particular image? This metric turned out to be very useful when tainting of the test data made us have to rely more on manual inspection of results.

Our fourth goal was exploring ways of using unsupervised ways of turning information in the training data into target information. We did not have

much success in this. The problem was mostly related to the design of loss functions rewarding the behavior we desired of the segmenter. The unsupervised image segmenter (section 5.5) kept exploiting the loss function. The auto-encoder (section 5.6) struggled to learn features rather than just colors. One method we did not explore, was using clustering based on depth slices of the filters from pre-trained convolutional architectures we explored. But the results would probably be similar to the clustering done using the unsupervised image segmenter as the concept is similar.

Our fifth goal was not explored throughout. The reason was the information added by a 3d renderer might be more related to perspective projection and spacial information like occlusion and orientation, then features for classification. We instead went for the manual segmentation approach, as it is more direct. Possible future work includes exploring synthetic rendering for detecting more volumetric features like mountains and mounds.

As mentioned before, untainted test data with labels are needed to continue working with the techniques mentioned in this writing. The over-fitting problems with the captioner can be mitigated using regularization techniques, however, we found it difficult to find a balance between over-fitting and a complex enough model for good performance. Currently, the segmentation model is limited when it comes to what features it detects, all features described in the captioner not detected by the segmentator is noise, leading to over-fitting. Future work might include improving the segmentation model so it can detect more features, which most likely would need more manually segmented data. Alternatively, filter away the mentioned features in the captions that are not yet detectable. An end to end training of the model could allow for more general detection of important features. We were not able to do this, due to over-fitting problems. We believe more data would be needed to do this.

Bibliography

[20] *Neural machine translation with attention*. https://www.tensorflow.org/tutorials/text/nmt_with_attention. Accessed: 2020-07-10. 12th June 2020.

[Ast09] Astley, R. *Rick Astley - Never Gonna Give You Up (Video)*. Youtube. 2009. URL: https://www.youtube.com/watch?v=dQw4w9WgXcQ.

[BCB14] Bahdanau, D., Cho, K. and Bengio, Y. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014. arXiv: 1409.0473 [cs.CL].

[Ble19] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Blender Institute, Amsterdam, 2019.

[CFC18] Casas, N., Fonollosa, J. A. and Costa-jussà, M. R. *A differentiable BLEU loss. Analysis and first results*. 2018.

[Den+09] Deng, J. et al. 'ImageNet: A Large-Scale Hierarchical Image Database'. In: *CVPR09*. IEEE Conference on Computer Vision and Pattern Recognition, 2009.

[Dr 19] Dr Michel Valstar, S. R. *Active (Machine) Learning - Computerphile*. 2019. URL: https://youtu.be/ANlw1Mz1SRI?t=315.

[DZ19] Dutta, A. and Zisserman, A. 'The VIA Annotation Software for Images, Audio and Video'. In: *Proceedings of the 27th ACM International Conference on Multimedia*. MM '19. Nice, France: ACM, 2019.

[G18] G, A. *A review of Dropout as applied to RNNs*. https://medium.com/@bingobee01/a-review-of-dropout-as-applied-to-rnns-72e79ecd5b7b. Accessed: 2020-07-09. 22nd June 2018.

[Gho+20] Ghosh, S. et al. 'MAARS: Machine learning-based Analytics for Rover Systems'. In: *2020 IEEE* vol. 1, no. 1 (2020).

[He+15] He, K. et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[Hoh+13] Hohenwarter, M. et al. *GeoGebra 4.4*. http://www.geogebra.org. Dec. 2013.

[How+17] Howard, A. G. et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[How19] Howard, J. *Using LaTeX for Your Thesis*. fastai. 2019.

[Kur18] Kurita, K. *An Intuitive Explanation of Why Batch Normalization Really Works*. https://mlexplained.com/2018/01/10/an-intuitive-explanation-of-why-batch-normalization-really-works-normalization-in-deep-learning-part-1/. Accessed: 2020-07-05. 10th Jan. 2018.

[KUR20] KURAMA, V. *A Review of Popular Deep Learning Architectures: ResNet, InceptionV3, and SqueezeNet*. https://blog.paperspace.com/popular-deep-learning-architectures-resnet-inceptionv3-squeezenet/. Accessed: 2020-07-03. 2020.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[Ono+19] Ono, M. et al. 'Make Planetary Images Searchable: Content-based search for PDS and On-Board Datasets'. In: *50th Lunar and Planetary Science Conference 2019* vol. 1, no. 1 (2019).

[Ono20] Ono, M. personal communication. 2020.

[Pap+02] Papineni, K. et al. 'BLEU: a Method for Automatic Evaluation of Machine Translation'. In: (Oct. 2002).

[PK16] Park, S. and Kwak, N. 'Analysis on the Dropout Effect in Convolutional Neural Networks'. In: *ACCV*. 2016.

[Qi15] Qi, R. (*Learning 3D Object Orientations From Synthetic Images*. 2015.

[RFB15] Ronneberger, O., Fischer, P. and Brox, T. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[Rot+16] Rothrock, B. et al. 'SPOC: Deep Learning-based Terrain Classification for Mars Rover Missions'. In: *AIAA SPACE 2016* vol. 1, no. 1 (2016).

[Sze+15] Szegedy, C. et al. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV].

[Wik20] Wikipedia contributors. *Regression toward the mean — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-July-2020]. 2020.

[Won19] Wong, W. *What is Teacher Forcing*. https://towardsdatascience.com/what-is-teacher-forcing-3da6217fed1c. Accessed: 2020-07-10. 15th Oct. 2019.