Understanding Variational Autoencoders' Latent Representations of Remote Sensing Images

Hannes Stärk

September 5, 2019

Abstract

In computer vision, neural networks have been successfully employed to solve multiple tasks in parallel in one model. The joint exploitation of related objectives can improve the performance of each individual task. The architecture of these multi task models can be more complex since the networks branch into the atomic tasks at a certain depth. For this reason the process of designing such architectures often involves a lot of time consuming trial-and-error. Therefore a more systematic taxonomy is desirable to replace this experimental process. For constructing this taxonomy it is necessary to have an understanding of the latent information learned by specific layers of single task models.

This work uses convolutional variational autoencoders to produce latent representations of aerial images which are analyzed to understand the inner workings of the models. The method relies on testing whether or not learned clusters can be attributed to different high level input features like topographic classes common for the field of remote sensing. Visualizations are produced to gain insight and an understanding of the information captured in the latent space of the variational autoencoders. Moreover, it is observed how different architectural choices affect the reconstructions and the latent space. Code to reproduce the experiments is publicly available here: https://github.com/HannesStaerk/bachelorThesis.

Contents

1 Introduction

Convolutional neural networks (CNNs) have achieved impressive results in solving the single task of semantic segmentation, i.e. pixelwise classification. Different approaches for designing these CNNs have been published such as adapting known image classification architectures and fine-tuning them for semantic segmentation (Shelhamer et al. 2017) or directly constructing architectures trained for semantic segmentation only (Jégou et al. 2016).

Other authors have researched multi task learning (Kendall et al. 2018) where a single CNN is trained for multiple tasks in parallel. This should improve generalization and increase processing efficiency since multiple problems are solved simultaneously and the CNNs attention can be steered into a correct direction. The tasks for which this has been tested include semantic segmentation and depth estimation. Multi task learning has also been the topic in further research observing its impact on semantic segmentation of aerial images (Schmitz et al. 2019). The authors found no significant improvement in the results neither by using the height information as additional input nor by defining an additional task which leads to the assumption that the used CNN already learns the height and its meaning with only images as input.

While multiple researchers, like Schmitz et al. (2019), often relied on testing many different educated guesses for how to design multi-task models, there have already been proposals where a branched network is constructed automatically based on a relatedness measure between tasks (Vandenhende et al. 2019).

It is expected that a more systematic approach is helpful to eliminate the time consuming process of testing the many different possibilities of multi-task architectures. A multi-task taxonomy, with information about the relatedness of tasks and which tasks can be processed in parallel, will serve that purpose. This issue is addressed in this work since the construction of such a taxonomy requires an understanding of the latent information learned in the different layers of single task models. Knowledge about the features learned by each layer opens possibilities to determine at which depth a multi-task model optimally branches into the atomic tasks. For this purpose a variational autoencoder is trained to generate continuous latent representations of aerial images. Then the latent space of the variational autoencoder is analyzed regarding its clustering of different features such as topographic classes which are common for remote sensing like vegetation or buildings. The underlying expectation is that the used methods can also be employed to decipher the latent information learned in the hidden layers of different single task models. This is a necessary step towards a multi-task taxonomy that can decide at which layer a multi-task model should split into the atomic tasks.

2 Background

This section contains basic knowledge that is necessary for the understanding of the rest of the thesis. It briefly touches on artificial neural networks and convolutional neural networks in the beginning. The subsection after that goes on to describe autoencoders in general. Next, Kullback-Leibler divergence is explained in detail as it is an essential part for the variational autoencoders as they are used in this work. Kullback-Leibler divergence also plays an important role in t-distributed stochastic neighbor embedding. The difference between variational autoencoders and the standard undercomplete autoencoder is covered in the following subsection. Lastly, the dimensionality reduction method t-distributed stochastic neighbor embedding is discussed since it is used in this work to gain an understanding of the high dimensional latent space of the autoencoders.

2.1 Remote Sensing

In remote sensing there is on the one hand the collection of data that has come a long way and on the other hand the processing of said data. In the beginning remote sensing imagery was captured with film cameras from high places or air balloons which was improved a lot by the invention of airplanes and advances in camera technology. Another milestone was hit when high resolution satellites started continuously capturing high amounts of data of the whole world.
The processing of these vast amounts of data requires fast algorithms and a lot of computational resources. When these computational resources are available, the abundance of data makes the field of remote sensing predestined for the use of machine learning techniques. Therefore it is no surprise that deep learning algorithms find ever more applications in the field of remote sensing and enable improved solutions.

2.2 Artificial Neuron

The concept of neurons in computer science is inspired by biological neurons. An artificial neuron has one or more weighted inputs that are summed together and after that passed through a non-linear function called activation function. The output can then be used as the input of another neuron similarly to how stronger or weaker connections can be formed in the biological world. If a neurons activation function has a binary value range it is called a perceptron. By adjusting the separate weights of the inputs a single neuron is able to model simple functions like the logic *and* or the logic *or* but it is not abled to solve other trivial problems like the exclusive or (XOR).

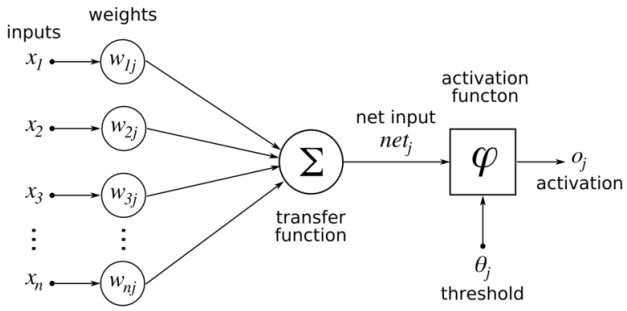


Figure 2.1: An artificial neuron. Image taken from (Chrislb 2005)

2.3 Artificial Neural Network

An artificial neural network (ANN) can model functions $f(x) = y$ with input and output vectors of any size. This is done by assigning each value of the input vector x to a single neuron. All those neurons together make up the input layer of the ANN. The output layer of the ANN also has a single neuron for every value of the output vector y. Between these input and output layers there can be multiple additional layers which are called hidden layers. If there is at least one hidden layer, i.e. at least three layers in total, the ANN is abled to model arbitrarily complex functions given that the hidden layer can contain as many neurons as necessary. However, in practice most of the time deep neural networks with two or more hidden layers are used (Géron 2017). If all the neurons of a previous layer are connected to each neuron in the following layer the layers are called fully connected which is the case for every layer in the artificial neural network depicted in Figure 2.2.

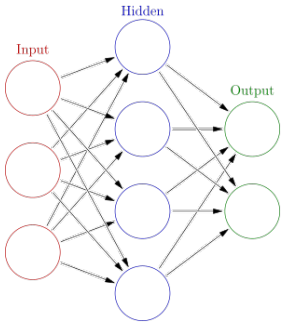


Figure 2.2: An artificial neural network with three fully connected layers. Image taken from (Glosser.ca 2013)

An ANN with at least three layers, consisting of perceptrons only, is called a multilayer perceptron (MLP). A lot of the theory about ANNs was built on the foundation of MLPs, including the learning process. The process of learning

an artificial neural network $n(w, x) = \hat{y}$, with weights w, refers to the process of iteratively adjusting w in such a way that $n(w, x)$ more closely resembles the desired mapping $f(x) = y$ that should be learned. This can be achieved by taking possible inputs x_i for which the desired outputs y_i, called ground truth, are known and optimizing a so called loss function $l(n(w, x_i), y_i)$ for w. This loss function measures a difference between the desired output and the predictions for that output produced by the ANN. This process is then repeated with each pair of inputs x_i and desired outputs y_i. The set of pairs of those inputs and desired outputs is called the training set. Intuitively, the optimized loss function can be seen as the objective of how the output of the artificial neural network should resemble the known desired outputs.
The optimization is most commonly done via gradient descent or adjusted versions of gradient descent. This involves calculating the partial derivatives for each component of w. For this reason the activation functions of the artificial neurons and the loss function have to be differentiable. More detailed information about gradient descent can be read in Deep Learning (Goodfellow et al. 2016) and information about the commonly used optimizers that improve gradient descent can be found in Hands-On Machine Learning (Géron 2017).

2.4 Convolutional Neural Networks

CNNs are especially good where the input has values that have to be interpreted in context to the other inputs. This is for example the case in computer vision where the inputs are images whose single pixel values have little meaning if they are not in the context of their position relative to the other pixels. An example for such a task, where CNNs perform well, is the classification of images of cats and dogs. The difference of CNNs and fully connected networks is the use of convolutional layers. A convolutional layer is not connected to every neuron of the previous layer but instead only to small rectangle of neurons of the previous layer. For images this means that in the first convolutional layer the neurons do not take every pixel of the image as input but instead only a small piece of the image. This way a hierarchical structure is created where the first hidden layer learns lower level features and the following layers learn ever higher level features. Also by only connecting a neuron to a few neurons of the previous layer the number of weights is far lower which means there is much less memory and time needed for training the network. That way computational expenditure is reduced in comparison to fully connected layers.
The kernel of a layer is intuitively speaking the field of vision where a neuron takes its inputs from the previous layer. If the kernel of a neuron laps over the edge of the input space different kinds of padding can be used for the missing input values. These kernels, also called filters, are basically matrices of weights that are adjusted while training which way they learn features of the input. In practice multiple neurons have the same input space leading to the ability to extract multiple features that can be processed further in the following layer. The amount of neurons with the same input space is referred to as the number of filters since each one of those neurons has its own filter with its respective

weights and all those filters are applied to the same piece of the input. This is visualized in Figure 2.3.

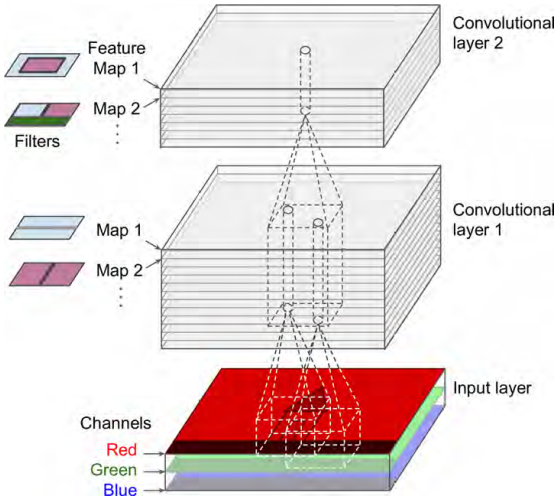


Figure 2.3: A convolutional network with 12 filters in the first convolutional layer and 7 filters in the second convolutional layer. Each filters can learn different features and an RGB image is depicted as the input here. Image taken from (Géron 2017)

Instead of having a kernel in each possible area of the input there can also be a distance between each kernel which is called stride. This can be used to decrease the dimensionality of the output of a layer. For example with strides of 2 the output dimensions are half as as long as the input dimensions.

Pooling A pooling layer replaces the value of an output with a summary of the nearby outputs. For instance max pooling only takes the maximum output of all outputs in a surrounding rectangle. Note that the pooling layers only perform an operation on multiple outputs of a previous layer and therefore do not have weights. The use of pooling layers increases the networks tolerance to small changes in the inputs (Goodfellow et al. 2016) which is most of the time desired since the network should learn features instead of exact pixel locations. Many popular CNN architectures like AlexNet (Krizhevsky et al. 2012) use pooling layers between the convolutional layers. In these CNNs the pooling layers are also often used to reduce the dimension of the layers.

2.5 Autoencoders

An autoencoder is an artificial neural network that aims to reproduce its input. The first part of an autoencoder is the Encoder $f(x) = z$ that takes an input x and maps it to a latent code z. The second part is the decoder $g(z) = x'$

that tries to generate a reproduction x' similar to the input x that the latent code z was produced from. Since the output of the neural network should be the same as the input there is no need for labeled data and the network can be learned unsupervised. Just having a network that is the identity function would be useless but by constraining the autoencoder in specific ways it can be forced to learn useful traits leading to possibilities in pretraining networks or randomly generating content similar to the content the autoencoder was trained with.
One common constraint for autoencoders is choosing a dimension for the latent code that is smaller than the dimension of the input. This means that the encoder is forced to learn to extract only the most important features from the input and the latent code is a representation of those most important features in a lower dimensionality than the input. Such an autoencoder, called undercomplete autoencoder, is visualized in Figure 2.4.

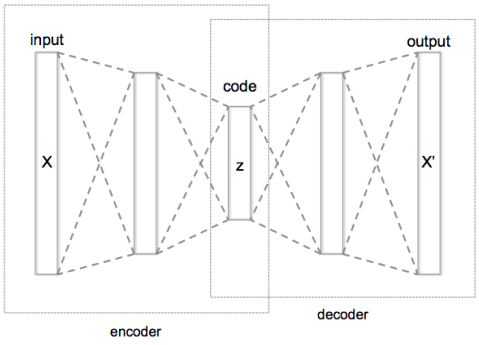


Figure 2.4: An undercomplete autoencoder has a latent code z whose dimension is smaller than the dimension of the input x. Image taken from (Chervinskii 2015)

Undercomplete autoencoders can be used for pretraining by taking the trained encoder that has learned the most important features of the input and using it for the actual desired task like classification. Another obvious application would be dimensionality reduction.

2.6 Variational Autoencoders

Standard autoencoders are fairly limited in their generative capabilities or in their capabilities to interpolate in the latent space and the problem lies in their latent code. For generation tasks a random latent code z is sampled from the space of possible latent codes. But with a standard autoencoder this space of possible latent codes is most likely not continuous and chances are high that the sampled z is intuitively speaking from a gap in the latent space that the decoder has not been trained for. Therefore it will not produce realistic outputs. This

discontinuous latent space also means that interpolation in it does not work well. To solve this in a variational autoencoder the encoder does not produce discrete values but rather probability distributions p_i by generating two vectors of the same size as z where one vector μ contains means and the other vector σ contains standard deviations. Then each z_i is determined by sampling it from $p_i = N(\mu_i, \sigma_i)$. As a result the decoder is forced to not only just learn how to decode single, discrete latent codes but also the possible variations introduced by the random sampling. This leads to the desired continuous decodable latent space. However, the encoder could still produce σ sigma that are very small and μ that are far apart. To prevent this the encoder is forced to produce distributions that are similar to a normal distribution with standard deviation 1 and mean 0 by adding a second loss function next to the loss $L(x, x')$ between the input and output.
For this purpose a measure for the difference between two probability distributions is necessary. This measure can be used as the second loss function for the variational autoencoder that should now also minimize the difference between the output distribution and $N(0, 1)$.
To achieve the goal of generating new content that is similar to the input that the variational autoencoder was trained with, one can sample each component for a latent code from $N(0, 1)$ and use it as the input for the decoder. Also smooth interpolation in the latent space is possible leading to possibilities for combining content in interesting ways.
Further details about variational autoencoders can be found here (Doersch 2016).

2.7 Kullback-Leibler Divergence

2.7.1 Entropy

The previous subsection 2.6 mentioned the necessity of a measure for the difference between two probability distributions that can be used as additional loss function in variational autoencoders. Kullback-Leibler divergence serves this purpose and ist used in this context. To understand Kullback-Leibler divergence it seems necessary to explain entropy from the field of information theory. In short, entropy is a measure for the minimum average size an encoding for a piece of information can possibly have.
Suppose there is a system S_1 that can have four different states a, b, c, d and every one of those states is equally likely to occur, that means the probability $P(x)$ of each state x is $1/4$. Now the goal is to losslessly transmit all information about that system with the minimum average amount of bits. That can be done with only two bits for example like this

$$a : 00 \quad b : 01 \quad c : 10 \quad d : 11$$

However, if $P(a) = 1$ and $P(b) = P(c) = P(d) = 0$, zero bits will suffice to encode the information since it is always certain that the system is in state a. So the entropy of the system clearly depends on the probabilities of each state. To see in which way, one can consider the system S_2 with $P(a) = 1/2,\ P(b) =$

1/4, $P(c) = 1/8$ and $P(d) = 1/8$. In that case it would be best to encode the state with the highest probability with as few bits as possible since it has to be transmitted the most often. That means a is encoded with one bit as 0. When decoding the information there must be no ambiguities so while the encoding for b has to start with a 1 it cannot be 1 since we need to encode two more states so $b : 10$. Additionally if $c : 11$ there would be no space left for d: say $d : 111$ then if the transmitted information is $111111\ldots$ it could either be decoded to $ccc\ldots$ or $dd\ldots$. So c should rather be encoded as 110 which way $d : 111$ works. In the end a valid encoding that can transmit all information with the minimum average amount of bits is

$$a : 0 \quad b : 10 \quad c : 110 \quad d : 111$$

Here the states c, d are encoded with three bits instead of the two bits in the first example. But c and d are transmitted far less often than a which now only needs one bit. To be more precise half of all transmissions have one bit. Additionally a quarter of all transmissions have two bits. The sum of those probabilities multiplied with the respective amount of bits is the average amount of bits needed to transfer the information in a given encoding. So in the example, with $f(x)$ as the number of bits that encode a state x, that turns out to be $P(a)f(a)+P(b)f(b)+P(c)f(c)+P(d)f(d) = 1.75$. That means for S_2 on average you only need 1.75 bits to encode a state and since that is also the minimum 1.75 is the entropy of S_2.
In general, when the encoding is optimal, $f(x)$ is the same as $\log_2 \frac{1}{P(x)}$. For S_1 $P(x)$ is 1/4 so the number of bits for x is $\log_2(4) = 2$ which matches the two bits the first encoding uses for the states of S_1.
The entropy H of a system with a set of discrete events X and the probability distribution $P(x)$ for each $x \in X$ is

$$H(P) = \sum_{x \in X} P(x) \log_2 \frac{1}{P(x)} = -\sum_{x \in X} P(x) \log_2 P(x) \tag{1}$$

This is often written as the expectation for a given state x under the distribution P.

$$H(P) = E_{x \sim P}[-log_2 P(x)]$$

Intuitively if a system has high entropy, the size of the encodings are high on average and many states have small probabilities. This means it is hard to predict what state the system will be in at a given time since there is no state that can be guessed with high confidence. If entropy is low, zero for example, one can be confident that the system is in a certain state as in the previous example with $P(a) = 1$.

2.7.2 Cross Entropy

If the real distribution P of a system is unknown an estimate distribution Q could be guessed and encoding sizes $-log_2 Q(x)$ can be produced which will not

be optimal for the true distribution P. Now with some data gathered and P known the used encoding sizes can be cross-checked with the expectation under the actual distribution resulting in the cross entropy $H(P, Q)$

$$H(P, Q) = E_{x \sim P}[-log_2 Q(x)] = -\sum_{x \in X} P(x) \log_2(Q(x)) \tag{2}$$

In machine learning tasks regarding classification this is often used as a loss function since the label of a piece of data gives us a distribution P with absolute certainty and $H(P) = 0$. The inaccurate distribution Q that the model estimates leads to $H(P, Q) > 0$ unless $P = Q$ where $H(P, Q) = H(P, P) = H(P) = 0$. So the learning algorithm can try to minimize $H(P, Q)$.

2.7.3 Kullback-Leibler Divergence

Having computed the entropy $H(P)$ and cross-entropy $H(P, Q)$ of two distributions $P,\ Q$ it is possible to compare those distributions by comparing $H(P)$ and $H(P, Q)$ through subtraction

$$\begin{aligned} D_{KL}(P \parallel Q) &= H(P, Q) - H(P) \\ &= E_{x \sim P}[-log_2 Q(x)] - E_{x \sim P}[-log_2 P(x)] \\ &= E_{x \sim P}[-log_2 Q(x) + log_2 P(x)] \\ &= E_{x \sim P}[log_2 \frac{P(x)}{Q(x)}] \\ &= \sum_{x \in X} P(x) log_2 \frac{P(x)}{Q(x)} \end{aligned} \tag{3}$$

where $D_{KL}(P \parallel Q)$ is called the Kullback-Leibler divergence of P and Q. This works because if $Q = P$ then $H(P, Q) = H(P)$ and therefore $D_{KL}(P \parallel Q) = 0$. On the opposite if $Q \neq P$ then $H(P, Q) > H(P)$ and therefore $D_{KL}(P \parallel Q) > 0$ proportional to how different Q and P are. In summary Kullback-Leibler divergence is a measure of how different two probability distributions are. The divergence is zero if the distributions are the same and greater than zero if not.

2.8 t-Distributed Stochastic Neighbor Embedding

The method t-distributed stochastic neighbor embedding (t-SNE) is a machine learning technique for reducing dimensionality. It was first introduced by Laurens van der Maaten and Geoffrey Hinton (Maaten et al. 2008) who claim that t-SNE would be especially well suited for creating single visualizations that reveal the structure of the high dimensional data. Their paper is also where most of the information for this subsection is drawn from.
The method generates an initial embedding in the low dimensional space by sampling from a normal distribution centered around the origin. The learning process improves this embedding by first calculating a representation of the similarities between all points in the high dimensional space. For the similarity

of point x_i to x_j this representation is a conditional probability $p_{ij} = \frac{r_{i|j}+r_{j|i}}{2N}$, so the conditionals are symmetric, with $r_{i|j}$ given by the following.

$$r_{i|j} = \frac{exp(-|x_i - x_j|^2/2\sigma_i^2)}{\sum_k \sum_{l \neq k} exp(-|x_k - x_l|^2/2\sigma_i^2)} \tag{4}$$

For the similarity of points y_i and y_j from the low dimensional space the representation is a conditional probability q_{ij} given by the following.

$$q_{ij} = \frac{(1 + -|x_i - x_j|^2)^{-1}}{\sum_k \sum_{l \neq k} (1 + -|x_i - x_j|^2)^{-1}} \tag{5}$$

Then the mistake between the representation in high dimensional space and in low dimensional space is given by the Kullback-Leibler divergence, subsection 2.7, of high dimensional and low dimensional conditionals which is $\sum_i \sum_{j \neq i} p_{ij} log \frac{p_{ij}}{q_{ij}}$. The Kullback-Leibler divergence is then minimized with gradient descent, moving the points in the embedding to a more optimal position.

3 Methodology

In this section the subsection environment will explain which tools were used to program the autoencoders and which hardware the developed models were trained and written on. The subsection Datasets contains information about the remote sensing image data that was analyzed using the autoencoders. The architecture subsection lays out the design of the different models regarding the layers and loss functions. The last subsection, Latent Space, explains the technique used to visualize and understand the latent code between the encoder and decoder with comparisons of the latent space between different architectures and varying sizes of the latent code.

3.1 Environment

3.1.1 Hardware

Training autoencoders with larger images of multiple channels takes a lot of computation especially if the features that should be learned are as complicated and abstract as the topography of remote sensing data. As this is very time consuming the training of the models was mainly done on a remote machine from the Leibniz University in Hannover that could run 24/7. Only the programming of the models and tests with small datasets took place on a local personal computer that had no GPU that was capable of training models in the given scenario. The following are the specifications of the used systems:

Personal Computer

AMD Ryzen 7 1700, 16 GB RAM, NVIDIA GeForce GTX 760 2GB

Remote Machine

Lenovo Legion Y520T-25IKL, Intel i7-7700, 8GB RAM, NVIDIA GeForce GTX 1060 3GB

3.1.2 Software

The programming language used for the work is Python (Rossum 1995) with the development environment PyCharm locally and and Jupyter Notebooks on a remote machine. The machine learning framework Tensorflow was used which is developed by Google and offers cross-platform support, running on most CPUs and GPUs (Martín Abadi et al. 2015). Whenever possible Tensorflow's implementation of the Keras API specification was used which is a high level API for training machine learning models. For dimensionality reduction of the latent space, Scikit-Learn was used (Pedregosa et al. 2011).

3.2 Datasets

The available data are 1024x1024 images of the two United States cities Jacksonville in Florida and Omaha in Nebraska taken from the US3D Dataset that was partially published to provide research data for the problem of 3D reconstruction (Bosch et al. 2019). The images for each recorded area cover one square kilometer and can be divided into four categories with the first one being multispectral satellite images with eight channels (MSI). From the MSI data three channels were extracted and used as red, green and blue intensities (RGB). Thirdly there are digital surface models (DSM) and Lastly semantic labeling with five different categories.

The MSI data was collected by the WorldView-3 satellite of Digital Globe from 2014 to 2016. The images were taken in different seasons and times of day leading to great differences in their appearance regarding for instance shadows, reflections, overall brightness or clouds. This is an advantage for training models that are capable of processing data with similar differences in appearance. In the MSI dataset a single picture consists of eight channels for eight different bands of the spectrum with a ground sample distance of 1.3 meters. The eight channels of the imagery correspond to the following wavelengths:

- Coastal: 400 - 450 nm
- Blue: 450 - 510 nm
- Green: 510 - 580 nm
- Yellow: 585 - 625 nm
1. Red: 630 - 690 nm
2. Red Edge: 705 - 745 nm
3. Near-IR1: 770 - 895 nm
4. Near-IR2: 860 - 1040 nm

Three of those channels were extracted and used as RGB data. Each pixel of an image is described by three bytes representing the intensity of the wavelengths

of the reflected light corresponding to either red, green or blue.
The DSM data was collected using light detection and ranging technology (Lidar) which measures the distance to points of the earth's surface. This distance is proportional to the value of the single channel that each pixel of the DSM has. Lastly, there are semantically labeled pictures with one channel of a single byte that encodes one of five different topographic classes. Those classes are vegetation, water, ground, building and clutter. The semantic labeling was done automatically from lidar data but manually checked and corrected afterwards. Those four categories of data all cover one square kilometer in each image. Additionally they contain a lot of oblique view on buildings and other valuable features like clear shadows making the data ideal for training models that should detect those features.
In the dataset there are $2,783$ RGB images of the size 1024×1024 available. Since the autoencoder should be able to distinguish between categories like shadows and vegetation each image is split into 64 pictures of size 128×128 resulting in $178,112$ total training images. Experimentation with larger image dimensions never produced reconstructions of acceptable quality regarding pure visual inspection. Another option would be to resize larger sections of the original images to 128×128 pixels. However, smaller picture sections are more likely to have a dominant feature like containing mainly shadows, buildings or vegetation. This is desirable since the variational autoencoder should learn to distinguish those features and to cluster them in an unsupervised manner.

3.3 Architecture

The first general structure of the vanilla autoencoder and the variational autoencoder is a combination of the convolutional autoencoder and the variational autoencoder presented in the second edition of Hands-On Machine Learning (Géron 2017). That model is adjusted to work with 128×128 RGB images and various different architectural changes are tested. These experiments are first conducted with only 3000 training images and 200 validation images since the training should be fast to allow testing many different architectures. The best performing models are then evaluated and compared after training with all $178,122$ pictures. Here the performance is measured regarding the quality of the reconstruction and not the quality of the latent space. In all these test the dimension of the latent code is 1024 while the architectures allow to alter the coding size for the experiments regarding the latent space in subsection 4.2. The described experiments regarding the architecture and their evaluation can be found in subsection 4.1.
The results were that a fully convolutional network without pooling performed best. However, to inspect the effects of pooling on the latent code an architecture with pooling is used as the second model in the latent space experiments. This model yields reconstructions that are only a little worse in terms of visual fidelity than the ones from the fully convolutional network but the training times were almost twice as long.

The following are choices that apply to both architectures. The RGB input values between 0 and 255 are first normalized. Each value is divided by 255 which means that there are only floats between 0 and 1. The last deconvolutional layer in the decoder network, Figure 3.2), uses the sigmoid activation function $S(x) = \frac{e^x}{e^x+1}$. This guarantees that the values in the output image are all between 0 and 1 and can be interpreted as RGB values. Every other layer uses a scaled exponential linear unit (SELU) as activation function with the standard scale $\lambda = 1.05070098$ and $\alpha = 1.67326324$ as they were calculated in (Klambauer et al. 2017).

$$selu(x) = \lambda \begin{cases} x & if x > 0 \\ \alpha e^x - \alpha & if x \leq 0 \end{cases} \tag{6}$$

Klambauer et al. (2017) show that using a scaled exponential linear unit (SELU) as activation function leads to self normalizing networks. These networks eliminate the problem of vanishing or exploding gradients. Additionally the authors prove superiority of SELU over other normalization techniques in their benchmarks.
This is in line with the results of Pedamonti (2018) who reaches the conclusion that SELU leads to faster learning rates than other activation functions such as ReLU or Leaky ReLU.

3.3.1 Fully Convolutional Architecture

This experiment is based on the insights from Springenberg et al. (2015) who find that including pooling layers such as max pooling into a convolutional architecture does not necessarily lead to an improvement. When using a large enough network the invariances that should be introduced by pooling layers can also be learned in a network with convolutional layers only. Thus, in certain cases, convolutional layers with strides of two can be used to reduce dimensionality in a CNN instead of pooling layers which is why both options are explored in this work. Figure 3.1 visualizes the used encoder network. It has an example size for the latent code of 1024 which is not fixed. There are three convolutional layers with kernels of the dimension 3×3 and strides of two. The output of the last convolutional layer is flattened to a single vector which means that it has the size of the depth times the height times the width of the last convolutional layer, i. e. $32 \times 16 \times 16 = 8,192$. This flattened vector is the input for two fully connected dense layers that produce the means and standard deviations with the chosen size of the latent code. The final step is to sample the latent code from the means and deviations.

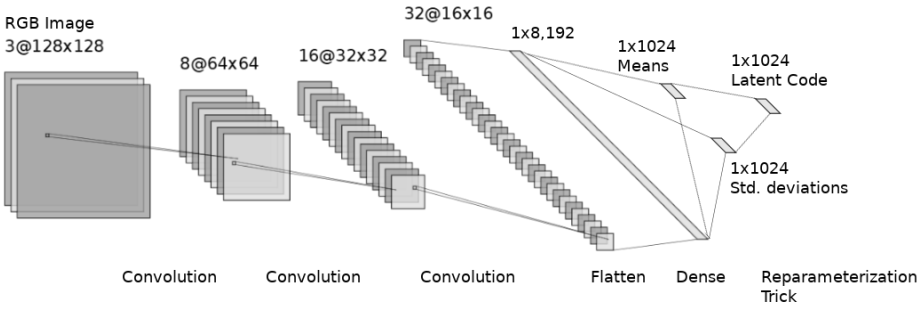


Figure 3.1: Encoder network architecture with an encoding size of 1024. The kernel size is 3×3 in every layer. The reparameterization trick in the last layer refers to the sampling as described in subsection 2.6. The dimensions of each layers output are specified with depth first, the height second and the width third.

The decoder, as visualized in Figure 3.2, takes the latent code produced by the encoder and first passes it through a dense layer with a $32 \times 16 \times 16 = 8,192$ output. This way it is possible to reshape that output vector to $32 \times 16 \times 16$ as the next step. After that three deconvolutions are applied. These deconvolutional layers are symmetrical to the convolutional layers in the first network resulting in an output with the same dimensions as the input of the encoder.

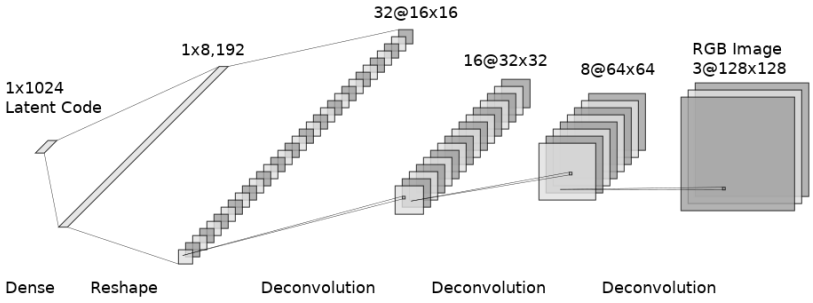


Figure 3.2: Decoder network architecture with an encoding size of 1024. The kernel size is 3×3 in every layer. The last deconvolutional layer has a sigmoid activation function. The dimensions of each layers output are specified with depth first, the height second and the width third.

3.3.2 Max Pooling Architecture

The used encoder with max pooling is depicted in Figure 3.3. It contains three convolutional and three pooling layers with a pooling layer following every convolutional layer. The convolutional kernels have size 3×3 and the convolution is done with strides of one which means that they do not reduce height and width of their input. Instead, each pooling layer quarters the dimension with

a pooling window of size 2×2. This produces an output of size $32 \times 16 \times 16$ that is processed further in the same way as in the fully convolutional encoder in subsubsection 3.3.1. The decoder is also equivalent to the one in the fully convolutional architecture.

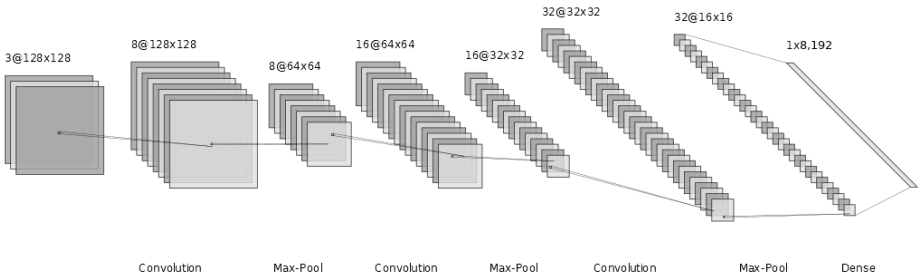


Figure 3.3: Part of the encoder network architecture with pooling and an encoding size of 1024. The convolutional kernel always has size 3×3. The pooling window always has size 2×2. The rest of the encoder that generates the latent code is the same as in the fully convolutional encoder in subsubsection 3.3.1. The dimensions of each layers output are specified with depth first, the height second and the width third.

3.4 Understanding Latent Space

To understand and work with the latent space learned by the variational autoencoder is problematic and difficult since it is a very high dimensional space and it is not known which information of the input the autoencoder learns. For the understanding and visualization of the latent space its dimensionality needs to be reduced while preserving the underlying structure of the latent space. Then the low dimensional codes can be visualized in a way that allows humans to comprehend what the autoencoder learned. One dimensionality reduction method that could be used here is principal component analysis (PCA).
However, t-distributed stochastic neighbor embedding (t-SNE), as presented in subsection 2.8, is used in addition to PCA in this work. The reason for this choice is that PCA focuses on preserving large distances, so dissimilar points in the high dimensional space appear far apart in a low dimensionality visualization. For visualization though it may be more interesting to keep more of the underlying structure which is accomplished by t-SNE as its focus is on preserving the context of points to their neighbors (Maaten et al. 2008). That behavior is caused by the asymmetry of the used Kullback-Leibler divergence, subsection 2.7. During the training process it penalizes a lot if large probabilities, i. e. far apart points, in high dimensional space are represented by small probabilities, i. e. close together points, in low dimensional space. Meanwhile, the penalty is negligible if small probabilities in high dimensional space are represented by large probabilities in low dimensional space, leading to the claimed focus on local similarities.
In the experiments subsection 4.2 this lower dimensional space is visualized for

different coding sizes for the two architectures presented in subsection 3.3. This gives insight into what coding sizes yield the best unsupervised clustering.

4 Experiments

This sections contains experiments regarding the architecture of the variational autoencoder (VAE) and the quality of the latent space. The experiments use the test and validation data of the RGB imagery as well as the ground truth semantic labeling and the digital surface models as described in subsection 3.2.

4.1 Variational Autoencoder Architecture

The experiments presented in this subsection determine the architecture that is used for the further experiments regarding the latent space in the next subsection. All the tested VAE architectures have latent code of the size 1024 and are trained with 3000 images over 50 epochs. The chosen batch size is 128 which is base on the results of Mishkin et al. (2016) who have analyzed the impact of multiple learning parameter and architecture choices on the performance of convolutional neural networks. One conclusion they reach is the suggestion to use batch sizes around 128 or 256. The training here is done on the personal computer described in subsubsection 3.1.1.
Regarding the evaluation of the experiments it is interesting how Theis et al. (2016) conclude that there are many difficulties in measuring the optimization and performance of generative models. They find that there is no "one-fits-all" metric for the quality of models like variational autoencoders. While the intuitive evaluation of visual results favors overfitted models, pure reliance on numerical measures is not appropriate when the goal is image synthesis. Overall the quality of evaluation measure is largely dependent on the application and goal. Therefore the performance of the VAEs is on the one hand measured by the mean absolute error (MAE) between the reconstruction and the input image. On the other hand it is judged by the subjective opinion of how realistic the reconstructions look and how well the semantic contents were recreated. This is the case since the MAE cannot capture the visual fidelity of a recreation. For instance if a reconstruction was an image that has the mean color values of the input as every pixel it is just one color in all of the image and only one value was captured from the input. However, this result has a lower MAE than an image with the inverted values of the input. Clearly the inverted image is a much better reconstruction of the input than the unicolored image and therefore additionally judging the results via a visual inspection is necessary.
The VAE architecture descriptions all cover an encoder up to a one dimensional tensor that is fed into two dense layers which create the means and standard deviations as explained in subsection 3.3. Since these two dense layers, the resampling and the dimension of the latent code is the same for every architecture the following specifications of encoders do not include these features for brevity.

The inputs for every encoder are 128×128 RGB images, i.e. $128 \times 128 \times 3$ images. The inputs for every decoder are the latent code vectors of size $1,024$.

4.1.1 Number of Convolutions

This subsection contains 5 tests for which the encoder and decoder architectures are specified in Tables below. The layers contained by the architecture in a test is specified by the number in the *Test* column as described in the tables caption. This means that the test architecture 4 has 6 convolutional and 6 deconvolutional layers while test 1 has only 3 convolutional and 3 deconvolutional layers.

Experiment Architectures

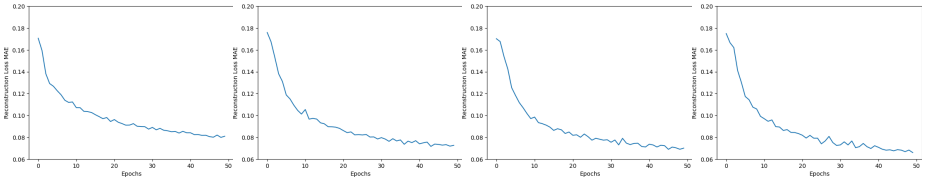
Test	Layer	Output
all	Conv: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
all	Conv: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
1	Conv: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
2	Conv: Kernel 3×3, Stride 2×2, Filters 64	$8 \times 8 \times 64$
3	Conv: Kernel 3×3, Stride 2×2, Filters 128	$4 \times 4 \times 128$
4	Conv: Kernel 3×3, Stride 2×2, Filters 256	$2 \times 2 \times 256$
all	Flatten	$1,024$

Table 1: The layers of the encoder of test 4 up to the vector that the two dense layers use to produce the means and standard deviations for the latent code. Test x only contains the layers up to the layer with x in the *Test* column and the layers with *all*. The output of the *Flatten* layer varies depending on the test.

Test	Layer	Output
all	Dense	$1,024$
all	Reshape	$2 \times 2 \times 256$
4	Deconv: Kernel 3×3, Stride 2×2, Filters 128	$4 \times 4 \times 128$
3	Deconv: Kernel 3×3, Stride 2×2, Filters 64	$8 \times 8 \times 64$
2	Deconv: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
1	Deconv: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
all	Deconv: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
all	Deconv: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$

Table 2: The layers of the decoder of test 4. Test x only contains the layers after the layer with the number x in the *Test* column and the layers with *all*. The outputs of the *Dense* and *Reshape* layers vary depending on the test.

Results

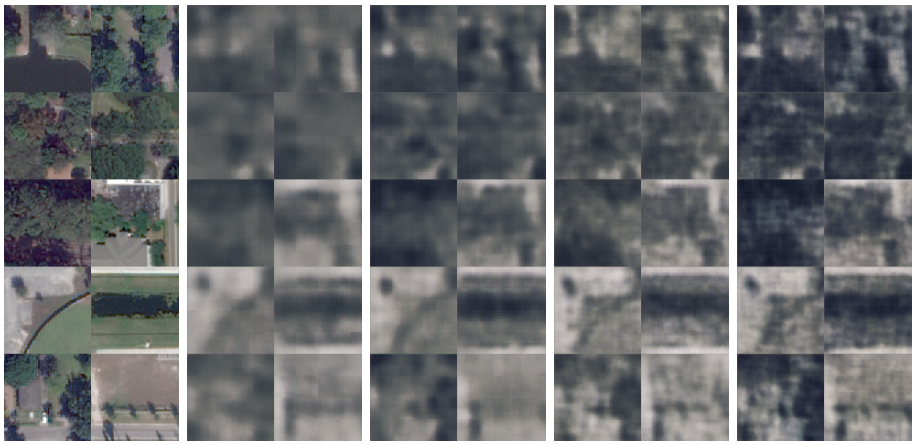


(1) (2) (3) (4)

Figure 4.1: There is a graph for each test architecture. Each graph depicts the mean MAEs between input and reconstructions across all iterations in an epoch of training.

Test	MAEs last Epoch	Loss last Epoch	Training time
4	0.08093627	0.08655346	12 min 58 sec
3	0.07256235	0.07953142	13 min 17 sec
2	0.07004701	0.07820769	14 min 8 sec
1	0.06594399	0.0751977	15 min 27 sec

Table 3: For each test architecture the table shows the mean of the MAEs across all iterations of the last epoch, the mean loss across all iterations of the last epoch and the time it took to train the model on the personal computer as specified in subsubsection 3.1.1. Note that the loss is not the same as MAE since it additionally takes the Kullback-Leibler divergence into account.



(1) (2) (3) (4)

Figure 4.2: The left column shows original images taken from a validation set that the VAE has not seen during training. The other columns show the reconstructions of the VAE test architectures.

Discussion

The learning curves in Figure 4.1 show a correlation between the MAE of the reconstructions and the number of convolutional and deconvolutional layers. The less layers the are the better the learning curve is which is in line with the visual fidelity of the reconstructions in Figure 4.2. The reconstructions of the VAE with the least layers are the sharpest and the more layers the VAE has the blurrier the reconstructions are.
A possible reason for this behavior is that the number of weights in the two dense layers between the last convolutional layer and the means and standard deviations increases drastically if the number of convolutional layers decreases. This is the case because for each additional value in the last convolutional layer there are $1,024$ additional weights in the three dense layers of the VAE since $1,024$ is the size of the latent code. For example architecture 4 only has $3,935,651$ weights while architecture 3, with one convolutional and deconvolutional layer less, has $6,492,192$ total weights. The weights in the dense layers make up most of the total weights and the higher amount of weights also leads to longer training times as it can be seen in Table 3. However, this increase in training time is negligible compared to the increase of the quality of the reconstructions. This leads to the idea that increasing the number of filters, and therefore the size of the dense layers, could yield better results which is tested in the next section.

4.1.2 Number of Filters

The tables below describe test architectures in a similar way as the previous subsubsection 4.1.1. The differences here are that the number of filters is doubled and the last convolutional layer and the first deconvolutional layer is missing.

Experiment Architectures

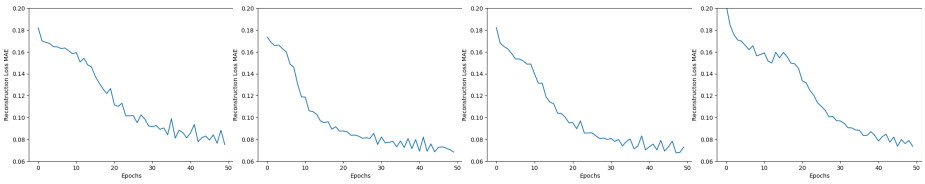
Test	Layer	Output
all	Conv: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
1	Conv: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
2	Conv: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
3	Conv: Kernel 3×3, Stride 2×2, Filters 128	$8 \times 8 \times 128$
4	Conv: Kernel 3×3, Stride 2×2, Filters 256	$4 \times 4 \times 256$
all	Flatten	$2,048$

Table 4: The layers of the encoder of test 4 up to the vector that the two dense layers use to produce the means and standard deviations for the latent code. Test x only contains the layers up to the layer with x in the *Test* column and the layers with *all*. The output of the *Flatten* layer varies depending on the test.

Test	Layer	Output
all	Dense	$2,048$
all	Reshape	$4 \times 4 \times 256$
4	Deconv: Kernel 3×3, Stride 2×2, Filters 128	$8 \times 8 \times 128$
3	Deconv: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
2	Deconv: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
1	Deconv: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
all	Deconv: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$

Table 5: The layers of the decoder of test 4. Test x only contains the layers after the layer with the number x in the *Test* column and the layers with *all*. The outputs of the *Dense* and *Reshape* layers vary depending on the test.

Results

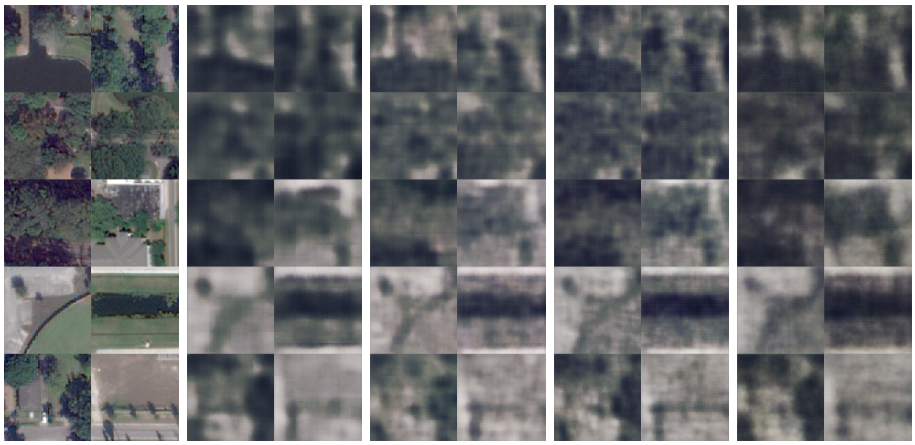


(1) (2) (3) (4)

Figure 4.3: There is a graph for each test architecture. Each graph depicts the mean MAEs between input and reconstructions across all iterations in an epoch of training.

Test	MAEs last Epoch	Loss last Epoch	Training time
4	0.07518449	0.08227883	21 min 31 sec
3	0.06828707	0.07637741	22 min 31 sec
2	0.07282908	0.08208682	26 min 17 sec
1	0.07343805	0.08214357	34 min 31 sec

Table 6: For each test architecture the table shows the mean of the MAEs across all iterations of the last epoch, the mean loss across all iterations of the last epoch and the time it took to train the model on the personal computer as described in subsubsection 3.1.1. Note that the loss is not the same as MAE since it additionally takes the Kullback-Leibler divergence into account.



(1) (2) (3) (4)

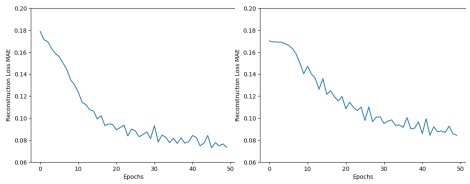
Figure 4.4: The left column shows original images taken from a validation set that the VAE has not seen during training. The other columns show the reconstructions of the VAE test architectures.

Discussion

The trend of less layers correlating with better results does not hold up for the experiments in this subsection when observing the MAEs in Table 6. However, the MAEs of the last epoch are not as crucial as in the previous subsection since the learning curves in Figure 4.3 show a lot of variance in comparison to the previous subsection. Still the best MAE of the previous architectures beats all results of these tests while also taking less than half as long to train and having more realistic looking reconstructions.

4.1.3 Kernel Size

In this subsection the two architectures 1 and 3, now called 1 and 2, of the subsubsection 4.1.1 are tested with a kernel size of 5×5 instead of 3×3. These architectures are chosen since architecture 1 performed the best so far and a second one is chosen for comparison.



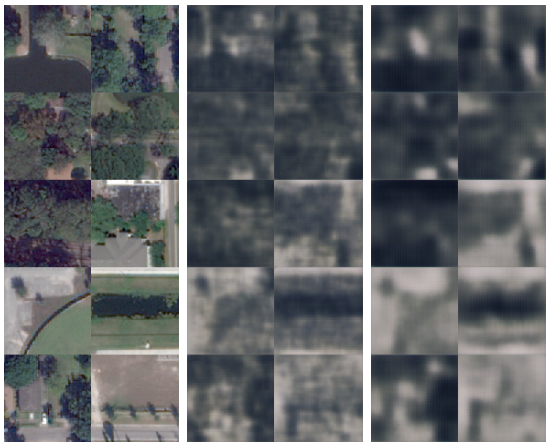
(1) (2)

Figure 4.5: There is a graph for each test architecture. Each graph depicts the mean MAEs between input and reconstructions across all iterations in an epoch of training.

Results

Test	MAEs last Epoch	Loss last Epoch	Training time
2	0.08430038	0.08984258	19 min 19 sec
1	0.07351803	0.08184456	19 min 40 sec

Table 7: For each test architecture the table shows the mean of the MAEs across all iterations of the last epoch, the mean loss across all iterations of the last epoch and the time it took to train the model on the personal computer as described in subsubsection 3.1.1. Note that the loss is not the same as MAE since it additionally takes the Kullback-Leibler divergence into account.



(1) (2)

Figure 4.6: The left column shows original images taken from a validation set that the VAE has not seen during training. The other columns show the reconstructions of the VAE test architectures.

Discussion

The architecture, which had more realistic reconstructions previously, still performs best with the increased kernel size. However, the reconstructions are all less realistic looking than the ones produced by the test architectures with a lower kernel size.

4.1.4 Pooling

This subsection again tests the so far best performing architecture, number 1 from subsubsection 4.1.1, but the convolutional layers now have a stride of 1×1 and the dimensionality reduction is done by pooling layers instead as visualized in the tables below. Additionally the architecture 2 from subsubsection 4.1.2 is tested with similar modifications for comparison. The tests 1 and 2 refer to these architectures with max pooling while 3 and 4 refer to the architectures with average pooling.

Experiment Architectures

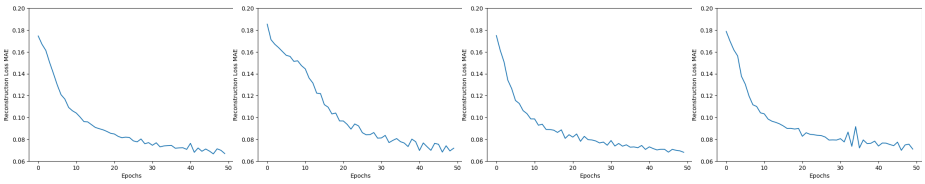
Layer	Output
Conv: Kernel 3×3, Stride 1×1, Filters 8	$128 \times 128 \times 8$
Pool: Pool size 2×2	$64 \times 64 \times 8$
Conv: Kernel 3×3, Stride 1×1, Filters 16	$64 \times 64 \times 16$
Pool: Pool size 2×2	$32 \times 32 \times 16$
Conv: Kernel 3×3, Stride 1×1, Filters 32	$16 \times 16 \times 32$
Pool: Pool size 2×2	$16 \times 16 \times 32$
Flatten	$1,024$

Table 8: The layers of the encoder of test 1 up to the vector that the two dense layers use to produce the means and standard deviations for the latent code.

Layer	Output
Dense	$1,024$
Reshape	$16 \times 16 \times 32$
Deconv: Kernel 3×3, Stride 1×1, Filters 16	$32 \times 32 \times 16$
Deconv: Kernel 3×3, Stride 1×1, Filters 8	$64 \times 64 \times 8$
Deconv: Kernel 3×3, Stride 1×1, Filters 3	$128 \times 128 \times 3$

Table 9: The layers of the decoder of test 1.

Results

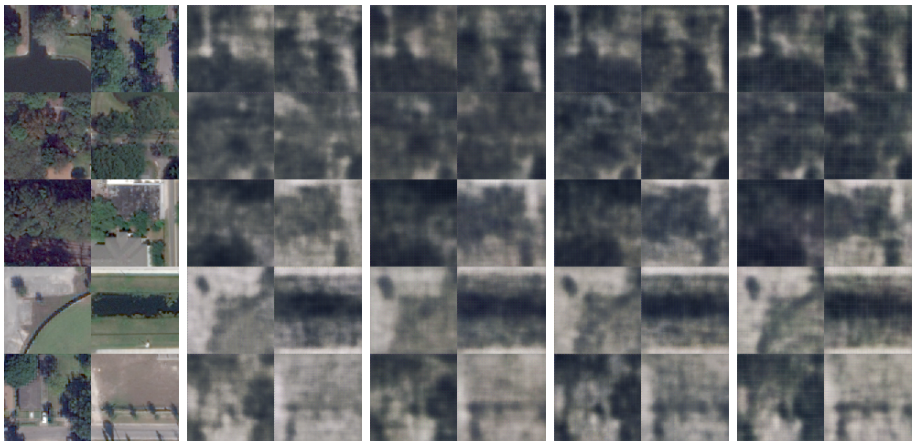


(1) (2) (3) (4)

Figure 4.7: There is a graph for each test architecture. Each graph depicts the mean MAEs between input and reconstructions across all iterations in an epoch of training.

Test	MAEs last Epoch	Loss last Epoch	Training time
4	0.07099171	0.07931245	53 min 11 sec
3	0.06803931	0.07641139	30 min 48 sec
2	0.07182192	0.08001233	54 min 9 sec
1	0.06689975	0.07499783	31 min 39 sec

Table 10: For each test architecture the table shows the mean of the MAEs across all iterations of the last epoch, the mean loss across all iterations of the last epoch and the time it took to train the model on the personal computer as described in subsubsection 3.1.1. Note that the loss is not the same as MAE since it additionally takes the Kullback-Leibler divergence into account.



(1) (2) (3) (4)

Figure 4.8: The left column shows original images taken from a validation set that the VAE has not seen during training. The other columns show the reconstructions of the VAE test architectures.

Discussion

Regarding the results in Table Table 10 it could be argued that pooling just led to worse results with MAEs that are a bit worse and training times that are far longer. However, the visual fidelity of the recreations is judged as the best for architecture 1 of this subsection.

4.1.5 Best Architectures

The two architectures that came out as the best regarding visual fidelity and the MAE between inputs and reconstructions are number 1 from subsubsection 4.1.1 and the architecture with max pooling. These are also the architectures presented in subsection 3.3 that are used for the further experiments regarding the latent space. The following Figure 4.9 shows the reconstructions of these architectures trained on the full training set of $178,112$ images instead of just 3000 images. The size of the latent code is $1,024$ again.
As expected the reconstructions are a lot better and apart from a bit of blur they look the same as the originals while the pooling architecture creates more blur than the fully convolutional architecture.

(a) (b)

Figure 4.9: The left column shows original images taken from a validation set that the VAE has not seen during training. Column (a) shows the reconstructions of the fully convolutional VAE while column (b) depicts the reconstructions of the max pooling architecture.

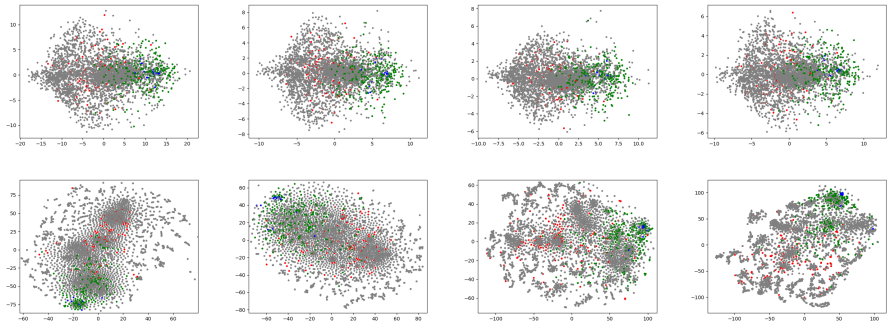
4.2 Latent Space

The results below visualize two dimensional data points that represent a latent code which then again corresponds to an input image. The dimensionality

reduction from the high dimensional latent space to two dimensional space is done via t-SNE as explained in subsection 3.4.
After PCA and t-SNE is used to reduce the dimensions of the latent code to 2, the low dimensional representation can be plotted in a scatter plot. This way it becomes evident if the autoencoder has learned any distinct clusters. To recap, every point in those plots is a low dimensional representation of a latent code that corresponds to a 128×128 section of an RGB satellite image. Now the goal is to find out what features the learned clusters in the plots correspond to. For that purpose the semantic labeling and digital surface models, subsection 3.2, are used. In the first type of plot the points are colored according to the most dominant class in the corresponding image. Those classes are building, vegetation, water, ground and clutter. In the second kind of plot the points have a gray value equal to the average pixel values in the digital surface models meaning that points for images that have on average more heights are brighter than points representing images with less heights. In a third type of plot the points are represented by the corresponding images themselves. In those plots it can be observed if the clusters correlate to one of the visualized features which would mean that the autoencoder has learned to cluster that feature.
All the plots in the following subsections display points for 4000 images.

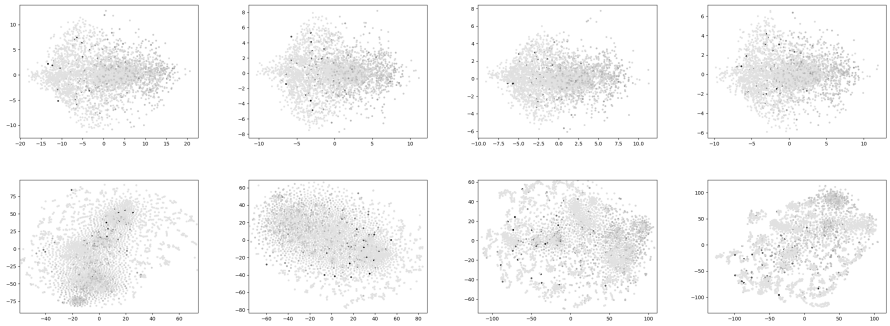
4.2.1 Fully convolutional Architecture

When observing the following two figures it first of all becomes evident that the VAEs have generated some clusters which is more clear in the visualizations produced with t-SNE than in those produced with PCA.
Regarding the question of what features the learned clusters correspond to, the plots colored by dominant class reveal that the VAEs have learned to distinguish water (colored blue) and vegetation (colored green) from the rest to a reasonable degree. Especially the t-SNE visualization for coding size 50 shows a very distinct cluster with all water images and two close clusters for most of the vegetation images. However, the same can not be said for height information. The overall high and low images are almost all evenly distributed across all clusters with only a single cluster appearing slightly darker, i.e. higher, than the rest. This suggests the VAEs have learned the topographic classes vegetation and water as features while not distinguishing by height up to the latent layer.



(a) Code size: $1,024$ (b) Code size: 512 (c) Code size: 128 (d) Code size: 50

Figure 4.10: PCA results in the top row. T-SNE results in the bottom row. The coloring corresponds to the predominant classes in the input images with green for vegetation, grey for ground, blue for water, red for buildings and yellow for clutter.



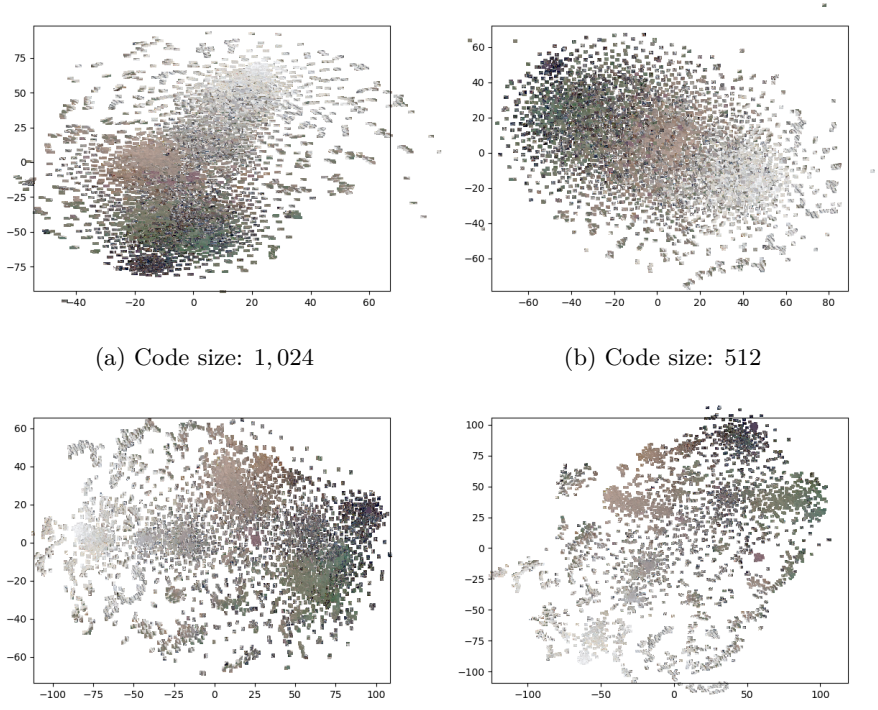
(a) Code size: $1,024$ (b) Code size: 512 (c) Code size: 128 (d) Code size: 50

Figure 4.11: PCA results in the top row. T-SNE results in the bottom row. The coloring corresponds to the average of the height values in the digital surface model for the input image. Darker points represent a higher average height.

The figure below only features the results of t-SNE since the PCA visualizations offer no additional insights. The most obvious of those insights is that the learned clusters correspond to the average color of the images. While this is rather easy to learn a closer inspection of the clusters show that the VAE has also learned far more complicated features which can be seen in Figure 4.14. The VAEs also seem to cluster by the direction of edges and higher level features like types of roads with images of different sections of the same road always being in the same cluster.

The t-SNE visualizations also show much more distinct clusters for different high level features if the coding size is lower with the coding size 50 yielding

the most obvious clusters. This suggests that the smaller latent space forces the VAE to learn and cluster high level features in an increased fashion. However, the reason for the better clustering might also simply be that the smaller latent representations are easier to interpret and therefore the t-SNE algorithm is abled to retain more of the structure in the high dimensional space. If the improved visualization is the result of the learning behavior of the VAE it means that there is a tradeoff between a more interpretable latent space with good clustering and the reconstruction quality. This is the case because a smaller latent dimension means worse reconstructions since the VAE has to fit the same information into a smaller vector. This can also be seen in Figure 4.13.



(a) Code size: $1,024$

(b) Code size: 512

(c) Code size: 128

(d) Code size: 50

Figure 4.12: Two dimensional output points of t-SNE of 4000 latent codes. The points are represented by the input images that produced the latent code corresponding to the two dimensional representation.

(a) 1024 (b) 512 (c) 128 (d) 50

Figure 4.13: The left column shows original images taken from a validation set that the VAE has not seen during training. The other columns show the reconstructions produced by the VAEs with the different coding sizes.

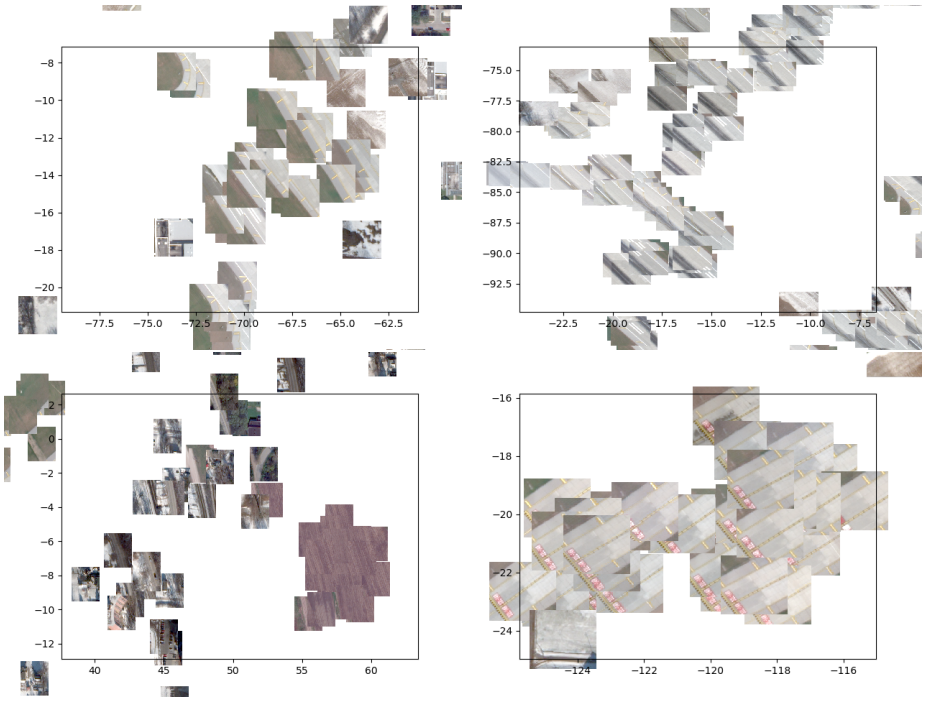
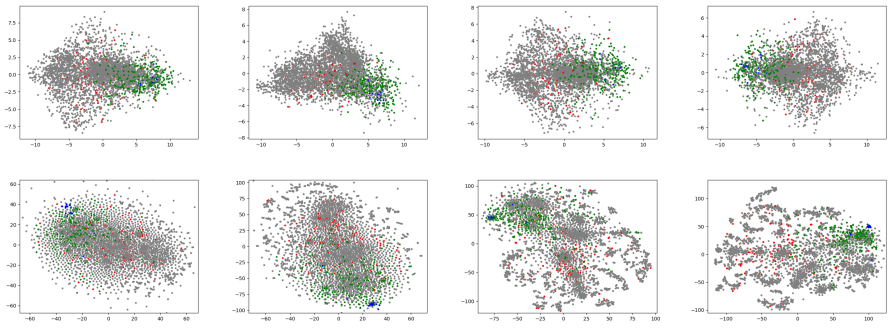


Figure 4.14: Clusters of VAE with coding size 50

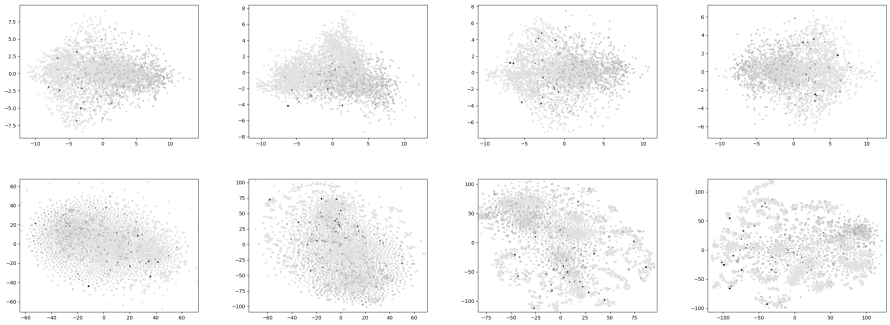
4.2.2 Pooling Architecture

The hypothesis is that the invariances introduced by the pooling layers force the VAE to learn more higher level features than the fully convolutional architecture. However, this does not seem to be the case since the figures below show almost exactly the same results as the fully convolutional VAE in the previous subsection. This means that the tested pooling architecture offers no advantages with the quality of the latent space being equal while producing worse reconstructions as it was seen in subsection 4.1. The convolutional VAE seems to implicitly learn the invariances introduced by the pooling layers.



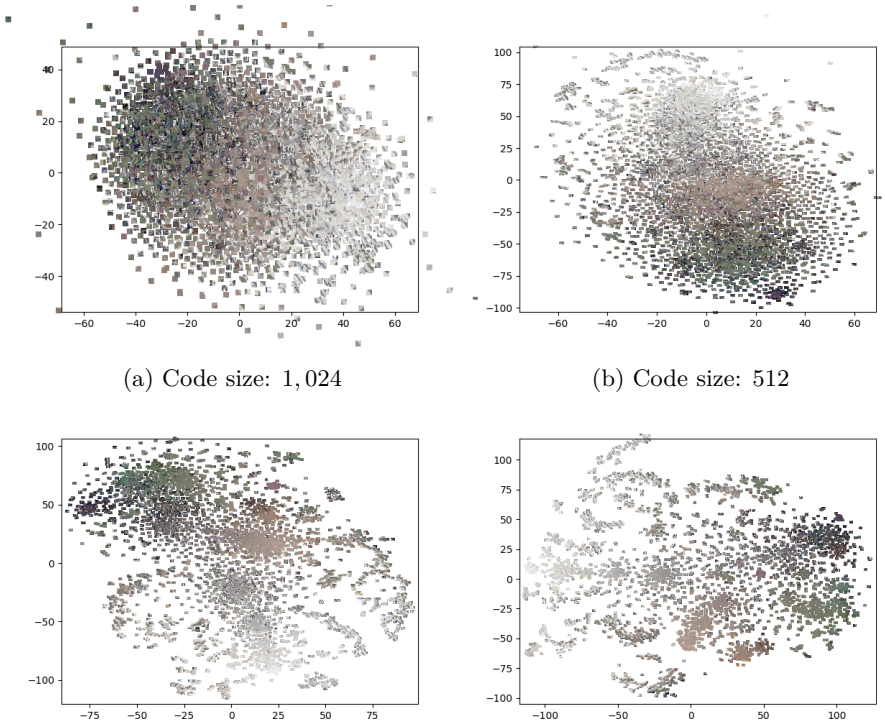
(a) Code size: $1,024$ (b) Code size: 512 (c) Code size: 128 (d) Code size: 50

Figure 4.15: PCA results in the top row. T-SNE results in the bottom row. The coloring represents the predominant class in the input with green for vegetation, grey for ground, blue for water, red for buildings and yellow for clutter.



(a) Code size: $1,024$ (b) Code size: 512 (c) Code size: 128 (d) Code size: 50

Figure 4.16: PCA results in the top row. T-SNE results in the bottom row. The coloring corresponds to the average of the height values in the digital surface model for the input image. Darker points represent a higher average height.



(a) Code size: $1,024$ (b) Code size: 512

(c) Code size: 128 (d) Code size: 50

Figure 4.17: Two dimensional output points of t-SNE of 4000 latent codes. The points are represented by the input images that produced the latent code corresponding to the two dimensional representation.

5 Conclusion and Future Work

5.1 Conclusion

The latent space of variational autoencoders for aerial images is investigated and visualized which allows for an improved understanding of the performed unsupervised clustering and the latent representation in general. For that purpose the thesis first offers a brief introduction into neural networks and convolutional architecture as well as a comprehensive explanation of how VAEs improve standard autoencoders to generate a useful continuous latent space.

Experiments with different VAE architectures are conducted regarding their reconstruction quality where a fully convolutional architecture yielded the most accurate reconstructions and this VAE's latent space is analyzed. Using principle component analysis and t-stochastic neighbor embedding the dimension of the latent representations is reduced and the results are visualized. It becomes clear

that the VAE clusters in an unsupervised manner. This means, up to the latent layer, it learned the features for which it clustered the inputs.

The formed clusters are compared with available height and classification data revealing that the learned clusters correspond to different topographic classes to a reasonable degree which is not the case for the average height of an image. While the VAE additionally groups the images by their average color it definitely also learns very complex features like types or directions of roads.

Thereby it is found that smaller latent dimensions lead to better visualizations of distinct clusters which suggests that the smaller encoding size forces better clustering to be learned. However, it is also possible that the shorter latent representations are just easier to interpret and the visualization techniques just worked better. That being said, a smaller latent space also means worse reconstructions and there is a tradeoff to be made between an interpretable latent space on the one hand and reconstruction quality on the other hand.

5.2 Future Work

From the viewpoint of this thesis the next step, in the regard of constructing a multi task taxonomy, is to apply the techniques used for understanding the latent space of VAEs to different layers in single task models. This insight about the latent information learned in the different layers of the single task model then opens possibilities to systematically determine optimal multi task architectures instead of relying on ad hoc testing.

To more directly establish on this thesis it would be interesting to analyze the unsupervised clustering performed by the VAEs for additional topographic classes if semantic labeling data for those classes was available. Moreover, a normalization method for the classes should be employed since in the used dataset an aerial image is more likely to predominantly have pixels labeled as ground instead of pixels labeled as building. This normalization could take the overall occurrence of each class in the whole dataset into account and reveal additional learned features in the latent layer.
Additionally, the attribution of the learned clusters to high level input features should be done more precisely than by merely testing whether or not the clusters correspond to available data. This can likely be done using layer-wise relevance propagation as proposed by Bach et al. (2015).

6 Erklärung der Urheberschaft

Hiermit versichere ich, die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, die Zitate ordnungsgemäß gekennzeichnet und keine anderen, als die im Literatur/Schriftenverzeichnis und Bilderverzeichnis angegebenen Quellen und Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Literature

Bach, Sebastian et al. (July 2015). "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PLoS ONE* 10.7, e0130140. DOI: 10.1371/journal.pone.0130140.

Bosch, Marc et al. (2019). "Semantic Stereo for Incidental Satellite Images". In: *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, pp. 1524–1532.

Doersch, Carl (2016). "Tutorial on variational autoencoders". In: *arXiv preprint arXiv:1606.05908*.

Géron, Aurélien (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media. ISBN: 978-1491962299.

Goodfellow, Ian et al. (2016). *Deep learning*. MIT press.

Jégou, Simon et al. (2016). "The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation". In: *CoRR* abs/1611.09326.

Kendall, Alex et al. (June 2018). "Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Klambauer, Günter et al. (2017). "Self-Normalizing Neural Networks". In: *CoRR* abs/1706.02515.

Krizhevsky, Alex et al. (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Maaten, Laurens van der et al. (2008). "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9, pp. 2579–2605.

Martín Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org.

Mishkin, Dmytro et al. (2016). "Systematic evaluation of CNN advances on the ImageNet". In: *CoRR* abs/1606.02228.

Pedamonti, Dabal (2018). "Comparison of non-linear activation functions for deep neural networks on MNIST classification task". In: *CoRR* abs/1804.02763.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Rossum, Guido (1995). *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands.

Schmitz, Matthias et al. (2019). "Semantic Segmentation of Airbourne Images and Corresponding Digital Surface Models - Additional Input Data or Additional Task?"

Shelhamer, Evan et al. (Apr. 2017). "Fully Convolutional Networks for Semantic Segmentation". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 39.4, pp. 640–651. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2016.2572683.

Springenberg, Jost Tobias et al. (2015). "Striving for Simplicity: The All Convolutional Net". In: *ICLR (Workshop)*.

Theis, L. et al. (2016). "A note on the evaluation of generative models". In: *International Conference on Learning Representations*. arXiv:1511.01844.

Vandenhende, Simon et al. (2019). “Branched Multi-Task Networks: Deciding What Layers To Share”. In: *CoRR* abs/1904.02920.

Images

Chervinskii (2015). *Schematic picture of an autoencoder architecture.* [CC BY-SA 3.0 (`http://creativecommons.org/licenses/by-sa/3.0/`)].

Chrislb (2005). *Diagram of an artificial neuron.* [CC BY-SA 3.0 (`http://creativecommons.org/licenses/by-sa/3.0/`)].

Glosser.ca (2013). *Artificial neural network with layer coloring.* [CC BY-SA 3.0 (`http://creativecommons.org/licenses/by-sa/3.0/`)].