Latent Representation of Topographic Classes in Remote Sensing Image Data using Autoencoders

Hannes Stärk

August 21, 2019

Contents

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.

5.1 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.

5.2 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. 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).

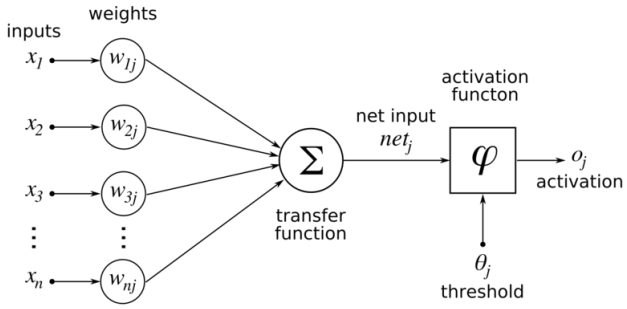


Figure 1: An artificial neuron. Image taken from (**2005-chrislb-artificial-neuron**)

5.3 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 (**2017-geron-homl**). 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 **??**.

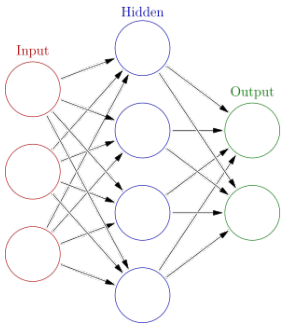


Figure 2: An artificial neural network with three fully connected layers. Image taken from (**2013-glosser-ann**)

The process of learning an artificial neural networks $n(w, x) = \hat{y}$, 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 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 (**2016-goodfellow-deep**) and information about the commonly used optimizers that improve gradient descent can be found in Hands-On Machine Learning (**2017-geron-homl**).

5.4 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 **??**.
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 halve that of 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 (**2016-goodfellow-deep**) which is most of the time desired since the network should learn features instead of exact pixel locations. Many popular CNN architectures like AlexNet (**2012-krizhevsky-imagenet**) use pooling layers between the convolutional layers. In these CNNs the pooling layers are also often used to reduce the dimension of the layers.

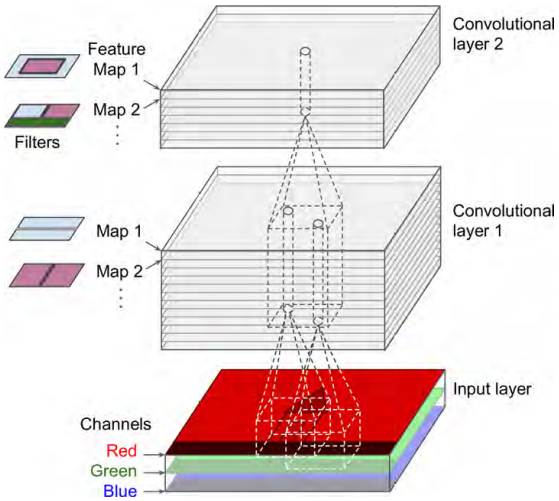


Figure 3: A convolutional network with 12 filters in the first convolutional layer and 7 filters in the second convolutional layer. Each filters can learn different features and an RGB image is depicted as the input here. Image taken from (**2017-geron-homl**)

5.5 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 **??**.
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.

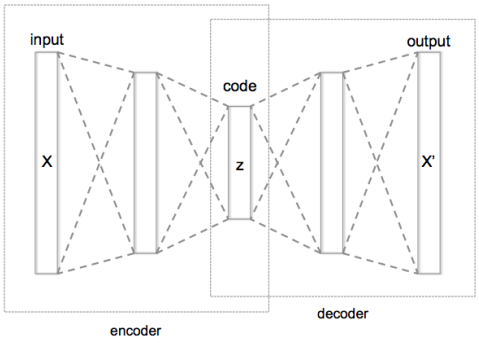


Figure 4: An undercomplete autoencoder has a latent code z whose dimension is smaller than the dimension of the input x. Image taken from (**2015-Chervinskii-autoencoder**)

5.6 Kullback-Leibler Divergence

5.6.1 Entropy

A part of the loss function used in the variational autoencoder is based on Kullback-Leibler divergence. 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 $S1$ 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 $S2$ 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 $S2$ 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 $S2$.
In general, when the encoding is optimal, $f(x)$ is the same as $\log_2 \frac{1}{P(x)}$. For $S1$ $P(x)$ 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 $S1$.
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 $P(a) = 1$.

5.6.2 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 $H(P) = 0$. With the inaccurate distribution Q that the model estimates $H(P,Q)$ will be greater than zero unless $P = Q$ where $H(P,Q) = H(P,P) = H(P) = 0$. So the learning algorithm can try to minimize $H(P,Q)$.

5.6.3 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 $D_{KL}(P \parallel Q)$ is zero if Q and P are the same since that means $H(P,Q) = H(P)$. On the opposite if Q is different from P then $H(P,Q)$ is greater than $H(P)$ and therefore the KL divergence is greater than zero 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.

5.7 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.
As discussed in **??** Kullback-Leibler divergence can be seen as a measure for the difference of two probability distributions. This means that it can be used as the second loss function for the variational autoencoder that should now also minimize the sum of all the Kullback-Leibler divergences 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 (**2016-doersch-tutorial**).

5.8 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 (**2008-vanDerMaaten-visualizing**) 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 **??** 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.

6 Related Work

This section presents concrete insights from related literature that influenced the used methodology. It is divided into two subsections. The first one, being convolutional network architecture, covers common practice about hyper parameters and other choices for convolutional models while the second subsection focuses on generative aspects and autoencoders.

6.1 General and Convolutional Network Architecture

In their paper *Striving for simplicity: The All Convolutional Net* (**2015-springenberg-striving**) the authors find that including pooling layers such as max pooling into a convolutional architecture does not necessarily mean 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.

The paper *Systematic evaluation of CNN advances on the ImageNet* (**2016-mishkin-systematic**) analyzes the impact of multiple learning parameter and architecture choices on the performance of convolutional neural networks. One conclusion it reaches is the suggestion to use batch sizes around 128 or 256.

In the work *Self-Normalizing Neural Networks* (**2017-klambauer-selu**) the authors 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 *Comparison of non-linear activation functions for deep neural networks on MNIST classification task* (**2018-Pedamonti-comparison**). This paper reaches the conclusion that SELU leads to faster learning rates than other activation functions like ReLU or Leaky ReLU.

6.2 Variational Autoencoders and Generative Networks

In *A Note on the Evaluation of Generative Models* (**2015-theis-generative**) it is made clear that there are many difficulties in measuring the optimization and performance of generative models. A clear conclusion is 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.

7 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.

7.1 Environment

7.1.1 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

7.1.2 Software

The programming language used for the work is Python (**1995-rossum-python**) 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 (**2015-martin-tensorflow**). 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 (**2011-pedregosa-scikit**).

7.2 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 (**2019-bosch-semantic**). 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 $2,783$ 1024×1024 RGB images available. Since the autoencoder should be able to distinguish between categories like shadows and vegetation each image is split into 64 128×128 pictures 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. 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.

7.3 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 Hans-On Machine Learning (**2017-geron-homl**). 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 **??**. The described experiments regarding the architecture and their evaluation can be found in section **??**.
The results were that a purely convolutional network without pooling performed the 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 results that are only a little worse than the ones from the purely 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 way there are only floats between 0 and 1. The last deconvolutional layer in the decoder network (Figure **??**) 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 (**2017-klambauer-selu**). This choice is made based on the insights from related work **??**.

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

7.3.1 Pure Convolutional Architecture

Figure ?? 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. $32 * 16 * 16 = 8,192$. 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.

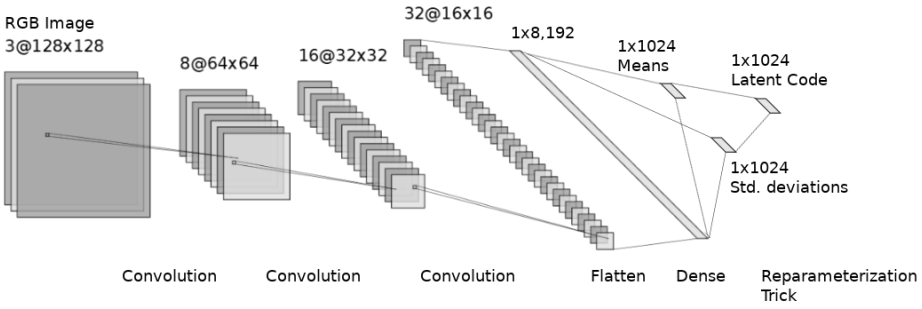


Figure 5: Encoder network architecture with an encoding size of 1024. The kernel size is 3×3 in every layer. The reparameterization trick in the last layer refers to the sampling as described in section ??. Dimensions are specified as $depth@height \times width$.

The decoder, as visualized in Figure ??, takes the latent code produced by the encoder and first passes it through a dense layer with a $32 * 16 * 16 = 8,192$ 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.

7.3.2 Max Pooling Architecture

The used encoder with max pooling is depicted in Figure ??. 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 and output of size $32 \times 16 \times 16$ that is processed further in the same way as in the pure convolutional encoder ??. The decoder is also equivalent to the one in the pure convolutional architecture.

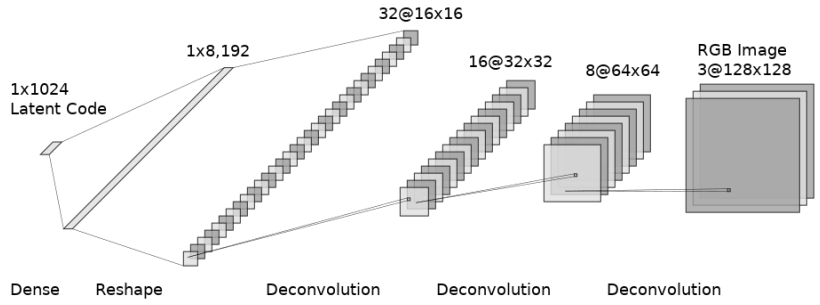


Figure 6: Decoder network architecture with an encoding size of 1024. The kernel size is 3×3 in every layer. The last deconvolutional layer has a sigmoid activation function. Dimensions are specified as $depth@height \times width$.

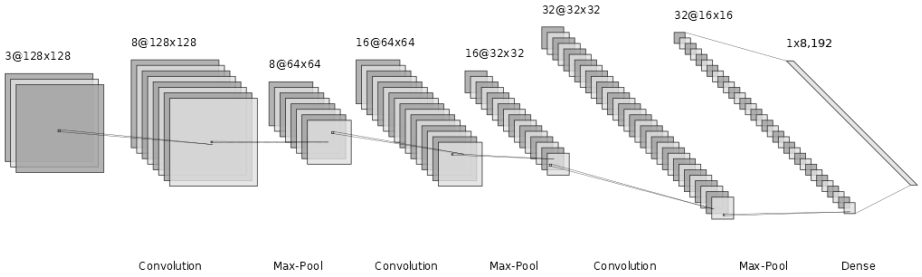


Figure 7: Part of the encoder network architecture with pooling and an encoding size of 1024. The convolutional kernel always has size 3×3. The pooling window always has size 2×2. The rest of the encoder that generates the latent code is the same as in the fully convolutional encoder **??**. Dimensions are specified as $depth@height \times width$.

7.4 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 **??**, is the technique used 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 (**2008-vanDerMaaten-visualizing**). That behavior is caused by the asymmetry of the used Kullback-Leibler divergence **??**. 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.
After 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. That 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 **??** 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 these two types of plots the points are squares if the related image is from Jacksonville or circles if it is from Omaha. 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. This method is repeated for different coding sizes for the two architectures presented in **??**. That process gives insight into what coding sizes yield the best unsupervised clustering. The specific experiments and results are found in the experiments section **??**.

8 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 **??**.

8.1 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 a batch size of 128, 3000 images over 50 epochs. This is done on the personal computer described in section **??**. The quality 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 section **??**. 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. Additionally the contents of each figure are only explained in the first architecture because there are the same types of figures for every test. 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 $1,024$.

Architecture 1

Encoder

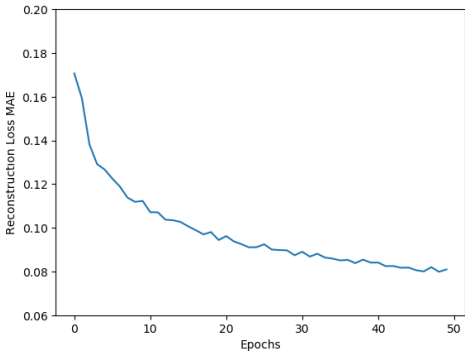
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 64	$8 \times 8 \times 64$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

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

Decoder

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

Table 2: The layers of the decoder.



- MAEs last epoch: 0.08655346
- Mean loss last epoch: 0.08093627
- Total weights: $3,935,651$
- Training time: 12 min 58 sec

Figure 8: The graph depicts the mean of all the mean average errors (MAEs) which were produced in the same epoch for every epoch. *"MAEs last epoch"* is the mean of the MAEs of the batches in the last epoch, i.e. the last value of the graph. *"Mean loss last epoch"* is the mean of the losses that the batches in the last epoch produced. Notice that this is not the same as the MAE since loss is the sum of the MAE and the Kullback-Leibler divergence. *"Total weights"* is the number of weights in the network that are trained.

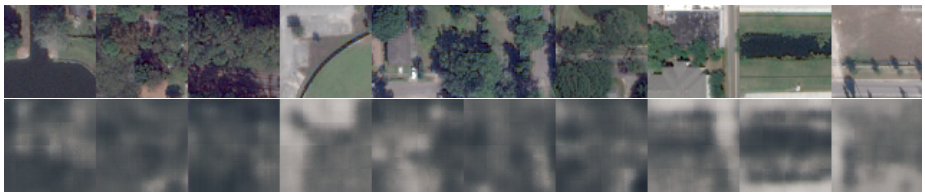


Figure 9: The original images in the top row are taken from a validation set that the VAE has not seen in training. The images in the bottom row are reconstructions of the original produced by the VAE after training.

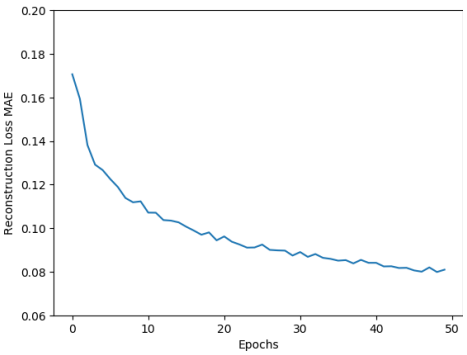
Architecture 2

Encoder

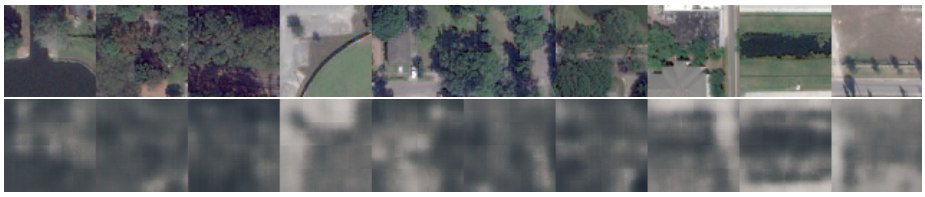
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
Convolution: Kernel 3×3, Stride 2×2, Filters 128	$8 \times 8 \times 128$
Convolution: Kernel 3×3, Stride 2×2, Filters 256	$4 \times 4 \times 256$
Convolution: Kernel 3×3, Stride 2×2, Filters 512	$2 \times 2 \times 512$
Flatten	$2,048$

Decoder

Layer	Output
Dense	$2,048$
Reshape	$2 \times 2 \times 512$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 256	$4 \times 4 \times 256$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 128	$8 \times 8 \times 128$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Total weights: $9,440,579$
- Training time:



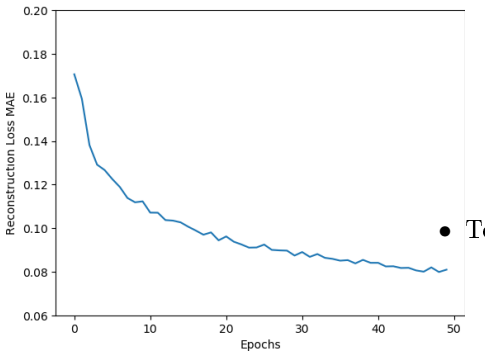
Architecture 3

Encoder

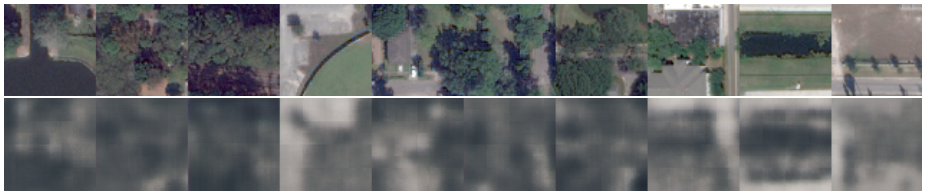
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 64	$8 \times 8 \times 64$
Convolution: Kernel 3×3, Stride 2×2, Filters 128	$4 \times 4 \times 128$
Flatten	$2,048$

Decoder

Layer	Output
Dense	$2,048$
Reshape	$4 \times 4 \times 128$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 64	$8 \times 8 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Total weights: $6,492, Trainingtime$:



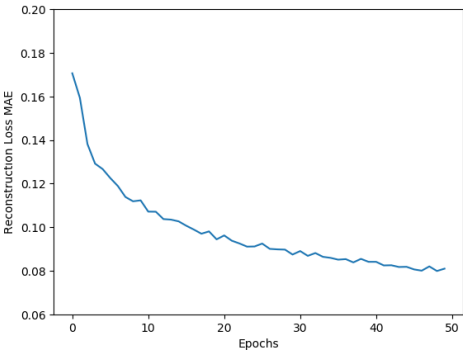
Architecture 4

Encoder

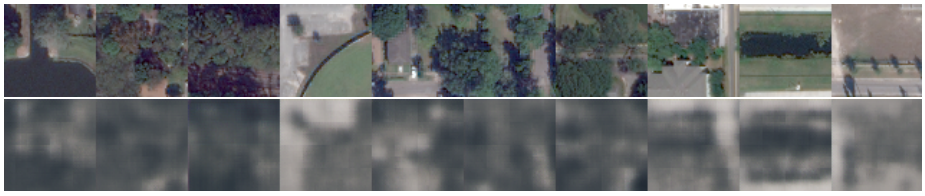
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
Convolution: Kernel 3×3, Stride 2×2, Filters 128	$8 \times 8 \times 128$
Convolution: Kernel 3×3, Stride 2×2, Filters 256	$4 \times 4 \times 256$
Flatten	$4,096$

Decoder

Layer	Output
Dense	$4,096$
Reshape	$4 \times 4 \times 256$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 128	$8 \times 8 \times 128$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



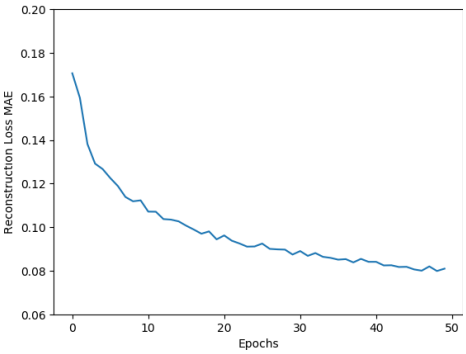
Architecture 5

Encoder

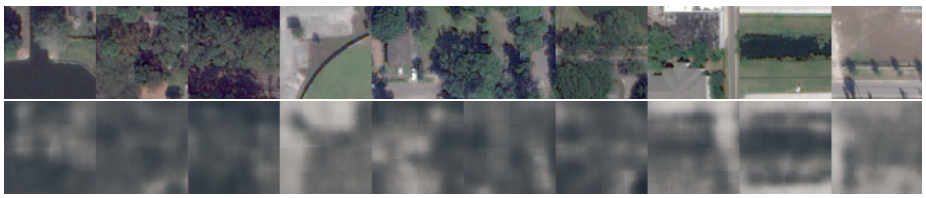
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 64	$8 \times 8 \times 64$
Flatten	$4,096$

Decoder

Layer	Output
Dense	$4,096$
Reshape	$8 \times 8 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



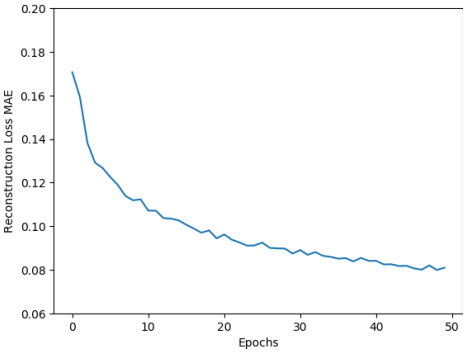
Architecture 6

Encoder

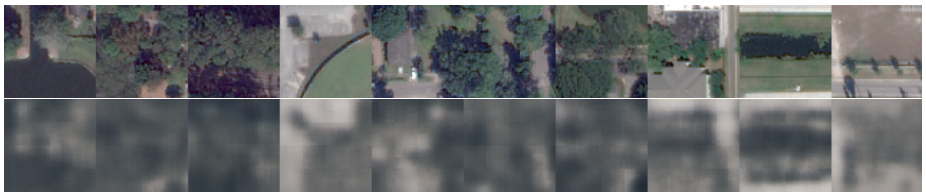
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
Convolution: Kernel 3×3, Stride 2×2, Filters 128	$8 \times 8 \times 128$
Flatten	$8,192$

Decoder

Layer	Output
Dense	$8,192$
Reshape	$8 \times 8 \times 128$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



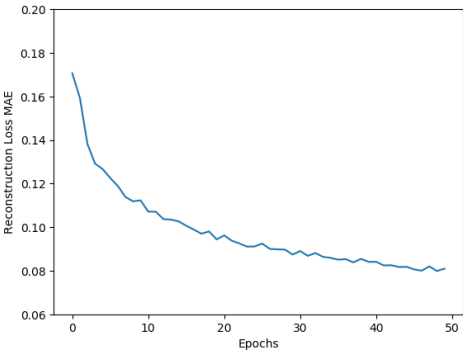
Architecture 7

Encoder

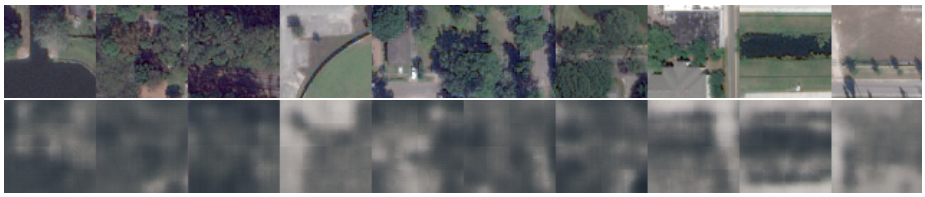
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 4$
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 8$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 16$
Flatten	$4,096$

Decoder

Layer	Output
Dense	$4,096$
Reshape	$16 \times 16 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$32 \times 32 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 4	$64 \times 64 \times 4$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



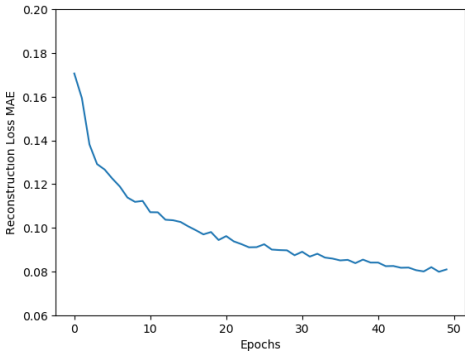
Architecture 8

Encoder

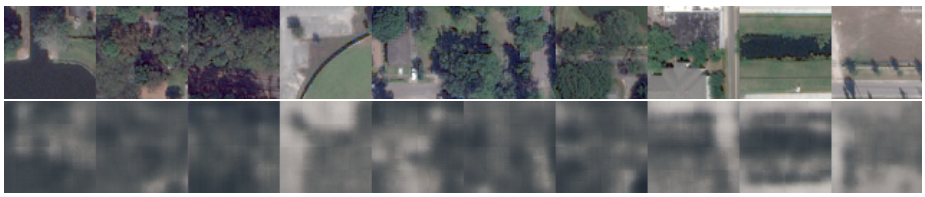
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Flatten	$8,192$

Decoder

Layer	Output
Dense	$8,192$
Reshape	$16 \times 16 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



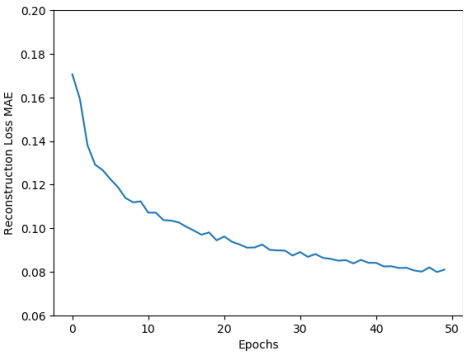
Architecture 9

Encoder

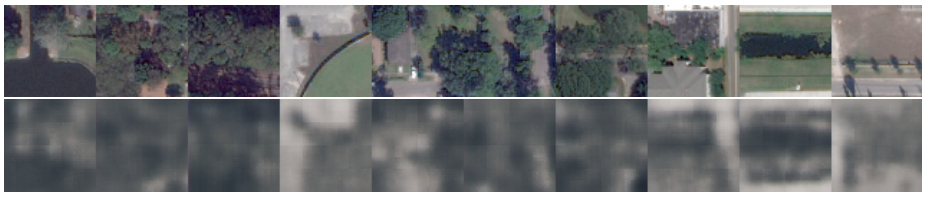
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Flatten	$8,192$

Decoder

Layer	Output
Dense	$8,192$
Reshape	$16 \times 16 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 1×1, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 1×1, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 1×1, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



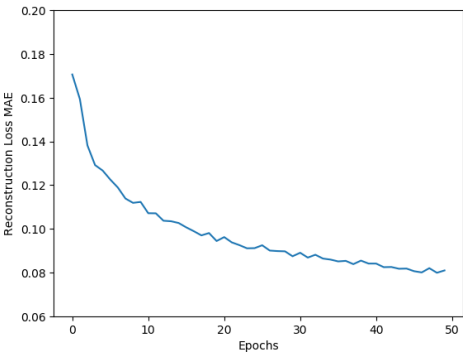
Architecture 10

Encoder

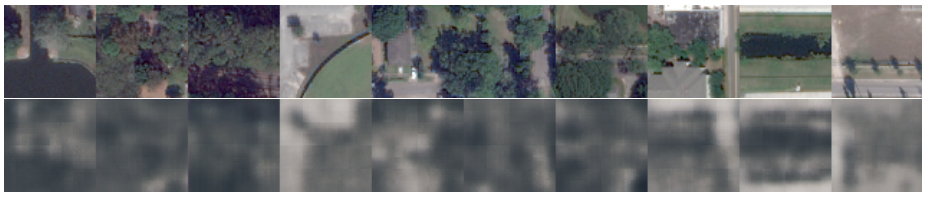
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 64	$16 \times 16 \times 64$
Flatten	$16,384$

Decoder

Layer	Output
Dense	$16,384$
Reshape	$16 \times 16 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



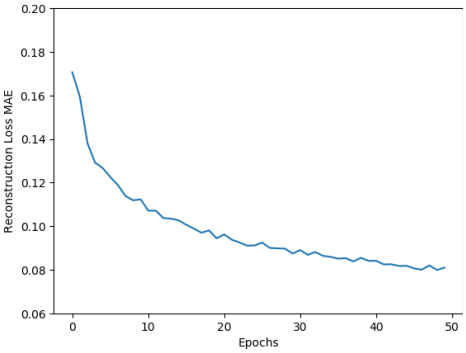
Architecture 11

Encoder

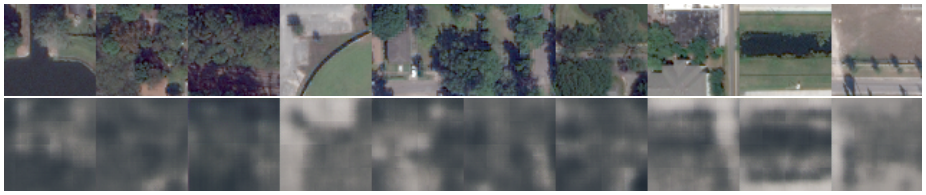
Layer	Output
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Flatten	$32,768$

Decoder

Layer	Output
Dense	$32,768$
Reshape	$32 \times 32 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



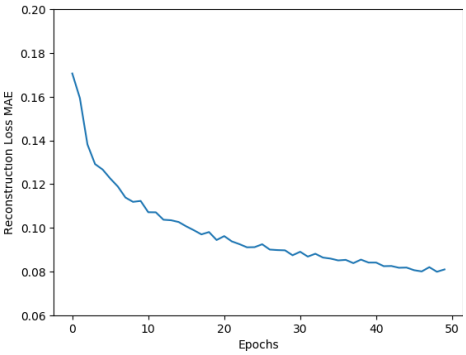
Architecture 12

Encoder

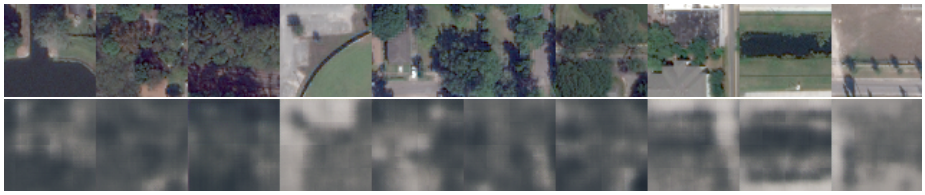
Layer	Output
Convolution: Kernel 3×3, Stride 1×1, Filters 8	$128 \times 128 \times 8$
Avg Pool: Pool size 2×2	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$64 \times 64 \times 16$
Avg Pool: Pool size 2×2	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 32	$32 \times 32 \times 32$
Avg Pool: Pool size 2×2	$16 \times 16 \times 32$
Flatten	$8,192$

Decoder

Layer	Output
Dense	$8,192$
Reshape	$16 \times 16 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



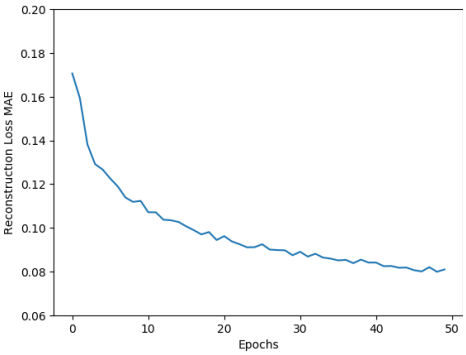
Architecture 13

Encoder

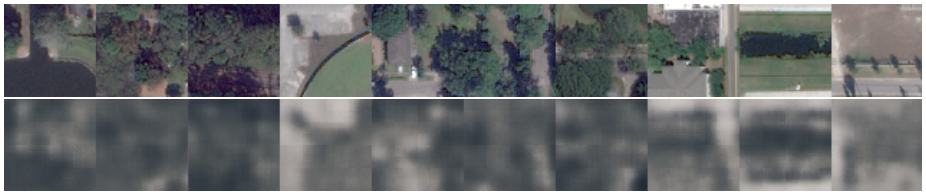
Layer	Output
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$128 \times 128 \times 16$
Avg Pooling: Pool size 2×2	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 32	$64 \times 64 \times 32$
Avg Pooling: Pool size 2×2	$32 \times 32 \times 32$
Convolution: Kernel 3×3, Stride 1×1, Filters 64	$32 \times 32 \times 64$
Avg Pooling: Pool size 2×2	$16 \times 16 \times 64$
Flatten	$16,384$

Decoder

Layer	Output
Dense	$16,384$
Reshape	$16 \times 16 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Total weights: $50,397,187$
- Training time:



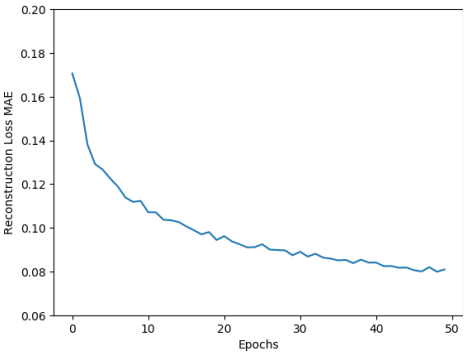
Architecture 14

Encoder

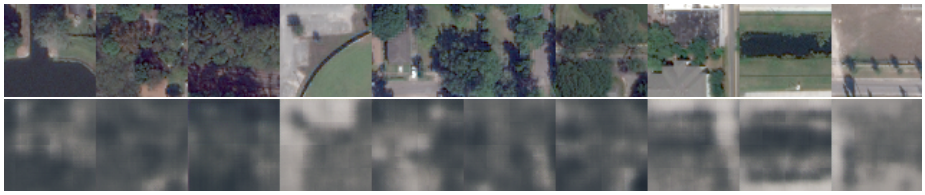
Layer	Output
Convolution: Kernel 3×3, Stride 1×1, Filters 8	$128 \times 128 \times 8$
Max Pooling: Pool size 2×2	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$64 \times 64 \times 16$
Max Pooling: Pool size 2×2	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 32	$32 \times 32 \times 32$
Max Pooling: Pool size 2×2	$16 \times 16 \times 32$
Flatten	$8,192$

Decoder

Layer	Output
Dense	$8,192$
Reshape	$16 \times 16 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



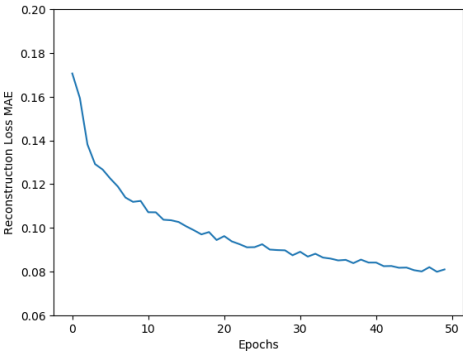
Architecture 15

Encoder

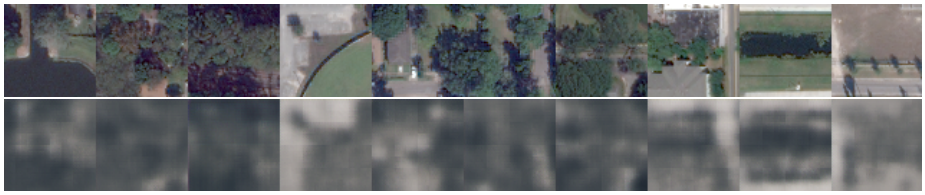
Layer	Output
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$128 \times 128 \times 16$
Max Pooling: Pool size 2×2	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 32	$64 \times 64 \times 32$
Max Pooling: Pool size 2×2	$32 \times 32 \times 32$
Convolution: Kernel 3×3, Stride 1×1, Filters 64	$32 \times 32 \times 64$
Max Pooling: Pool size 2×2	$16 \times 16 \times 64$
Flatten	$16,384$

Decoder

Layer	Output
Dense	$16,384$
Reshape	$16 \times 16 \times 64$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 32	$32 \times 32 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$64 \times 64 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



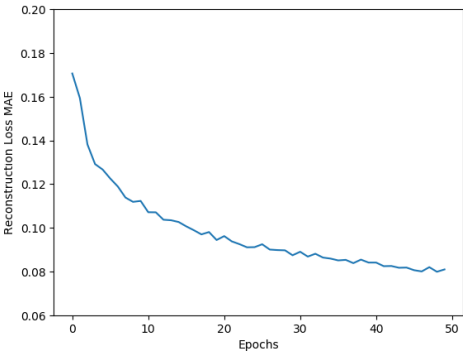
Architecture 16

Encoder

Layer	Output
Convolution: Kernel 5×5, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 5×5, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 5×5, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Convolution: Kernel 5×5, Stride 2×2, Filters 64	$8 \times 8 \times 64$
Convolution: Kernel 5×5, Stride 2×2, Filters 128	$4 \times 4 \times 128$
Flatten	$2,048$

Decoder

Layer	Output
Dense	$2,048$
Reshape	$4 \times 4 \times 128$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 64	$8 \times 8 \times 64$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:

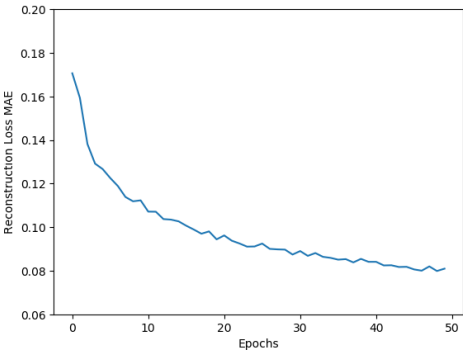
Architecture 17

Encoder

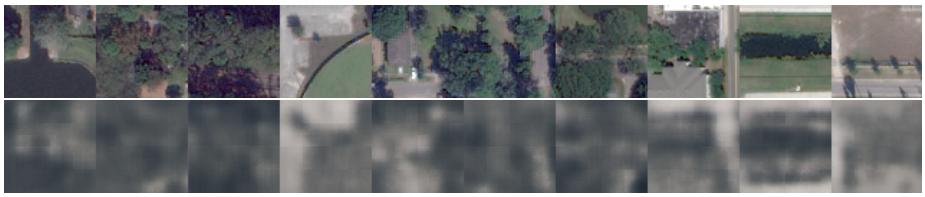
Layer	Output
Convolution: Kernel 5×5, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 5×5, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 5×5, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Flatten	$8,192$

Decoder

Layer	Output
Dense	$8,192$
Reshape	$16 \times 16 \times 32$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 5×5, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



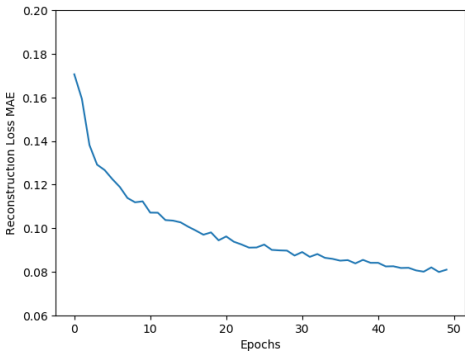
Architecture 18

Encoder

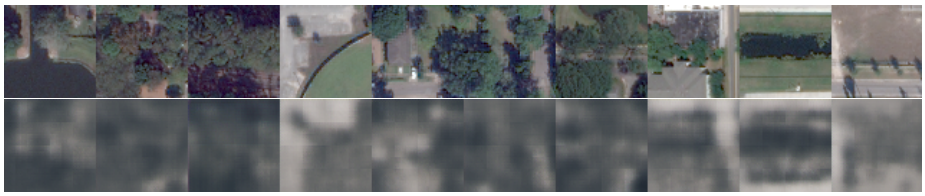
Layer	Output
Convolution: Kernel 3×3, Stride 1×1, Filters 8	$128 \times 128 \times 8$
Convolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Convolution: Kernel 3×3, Stride 1×1, Filters 16	$64 \times 64 \times 16$
Convolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Convolution: Kernel 3×3, Stride 1×1, Filters 32	$32 \times 32 \times 32$
Convolution: Kernel 3×3, Stride 2×2, Filters 32	$16 \times 16 \times 32$
Flatten	$8,192$

Decoder

Layer	Output
Dense	$8,192$
Reshape	$16 \times 16 \times 32$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 16	$32 \times 32 \times 16$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 8	$64 \times 64 \times 8$
Deconvolution: Kernel 3×3, Stride 2×2, Filters 3	$128 \times 128 \times 3$



- MAEs last epoch: 0.16682471
- Mean loss last epoch: 0.16682471
- Training time:



8.2 Latent Space

9 Conclusion and Future Work

9.1 Conclusion

9.2 Future Work