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

Hannes Stärk

August 13, 2019

Contents

1 Vorwort

2 Kurzfassung

3 Abstract

4 Introduction

4.1 Topic Overview

4.2 Research Questions

5 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 up, 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 next subsection. At last, the dimensionality reduction method t-distributed stochastic neighbor embedding is brought up since it is used in this work to gain an understanding of the high dimensional latent space of the autoencoders.

5.1 Remote Sensing Imagery

5.2 Artificial Neuron

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 models the axon of a biological neuron to which other biological neurons can connect via their dendrites which is similar to an artificial neuron using the output of another artificial neuron as its input. 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).

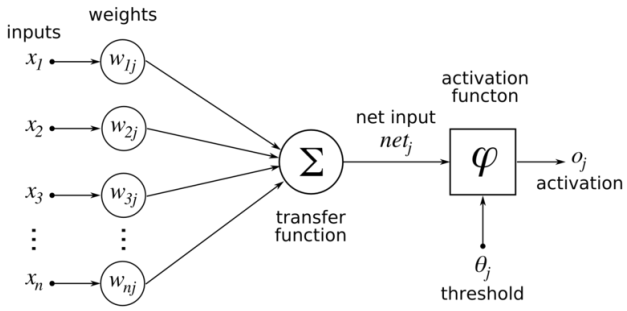

Figure 1: An artificial neuron. Image taken from (Chrislb 2005)

5.3 Artificial Neural Network

By taking layers of multiple artificial neurons and connecting the outputs of the layers to the inputs of the next layer, functions $f(x) = y$ with input and output vectors of any size can be modeled. The layers between the input and output layer are called hidden layers. Arbitrarily complex functions could be modeled by three layers if they may contain arbitrarily many neurons but in practice most of the time deep neural networks with two or more hidden layers are used (Geron n.d.). If all the neurons of a previous layers 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.

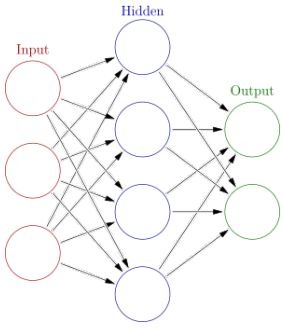

Figure 2: An artificial neural network with three fully connected layers. Image taken from (Glosser.ca 2013)

The process of learning an artificial neural networks $n(w, x) = z$, with weights w, refers to the process of iteratively adjusting w in such a way that $n(w, x)$ more closely resembles the desired function that should be learned $f(x)$. This

can be achieved by taking possible inputs x_i for which the desired outputs $\hat{z}_i$ are known and optimizing a loss function $l(n(w, x_i), \hat{z}_i)$ for w and repeating this process with each input x_i and desired output $\hat{z}_i$. 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 goal 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 which 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 (Geron n.d.).

5.4 Convolutional Neural Networks

When working with neural networks that should model a function with data like images as inputs, most of the time convolutional neural networks (CNNs) are used that contain 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.
The rectangle a neuron in a convolutional layer takes its inputs from is called kernel and has a kernel size specifying the height and width of the kernel. If the kernel of a neuron laps over the edge of the input space different kinds of padding can be used for the values of the kernel where there are no input values. These kernels, also called filters, are basically maps 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 3.
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

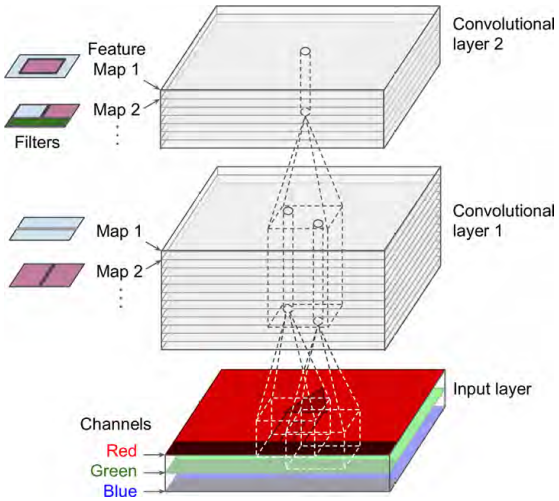

Figure 3: A convolutional network with 12 filters in the first convolutional layer and 7 filters in the second convolutional layer. Each filters can learn different features and an RGB image is depicted as the input here. Image taken from (Geron n.d.)

all outputs in a surrounding rectangle. Note that the pooling layers just 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 also often are used to reduce the dimension of the layers.

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 4.

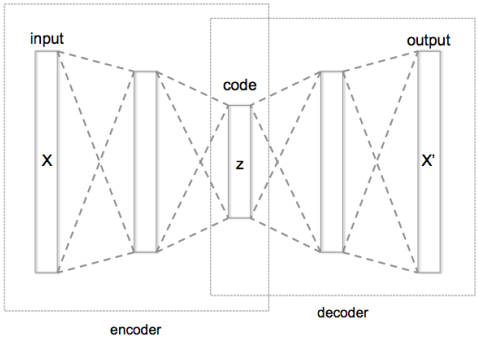

Figure 4: An undercomplete autoencoder has a latent code z whose dimension is smaller than the dimension of the input x. Image taken from (Chervinskii 2015)

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.

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 in an optimal encoding $f(x)$ 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 $\log_2(4) = 2$ what 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 like 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 that is zero if they 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 5.6 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. The total resulting loss function is the following equation.

$$L(x, x') + \sum_i D_{KL}(p_i \parallel N(0, 1)) \tag{4}$$

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).

5.8 t-Distributed Stochastic Neighbor Embedding

6 Related Work

7 Methodology

In this section the subsection environment will explain what tools were used to program the autoencoders and what 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. In the last subsection latent space there are the techniques used to visualize and understand the latent code between the encoder and decoder with comparisons of the latent space between normal autoencoders and variational autoencoders.

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 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 (Google n.d.). Whenever possible Tensorflows implementation of the Keras API specification was used which is a high level API for training machine learning models.

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 (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. Thereby 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 earths surface. This distance is proportional to the value of the single channel that each pixel of the DSM has. Lastly there are semantic 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. Those smaller picture sections are more likely to have a dominant feature like containing mainly shadows or only vegetation.

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 Hans-On Machine Learning (Geron n.d.). That model is adjusted to work with 128×128 RGB images

7.3.1 The Loss Function

Intuitively the loss function defines a goal that the model should reach when training by minimizing the loss function. In the variational autoencoder one of those goals is that the distribution of the latent space is similar to a normal distribution since that makes it possible to sample latent variables from a normal distribution. From those sampled variables the decoder can generate an output that resembles the real distribution.
To define that goal in a loss function Kullback-Leibler divergence is used to force the distribution of the latent space to resemble a normal distribution.
The second goal is of course that the output of the variational autoencoder is similar to the input. The loss function used for this purpose is the sum of the absolute differences between each input and each output pixel.
The final loss function is the sum of both, the Kullback-Leibler divergence and the sum of the absolute errors.

7.4 Latent Space

8 Conclusion and Future Work

9 Conclusion

10 Future Work

References

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.

Chervinskii (2015). *Schematic picture of an autoencoder architecture*. [Online; accessed August 09, 2019][CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)].

Chrislb (2005). *Diagram of an artificial neuron*. [Online; accessed August 08, 2019][CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)].

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

Geron, Aurelien (n.d.). *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly UK Ltd. ISBN: 9781492032649.

Glosser.ca (2013). *Artificial neural network with layer coloring*. [Online; accessed August 08, 2019][CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)].

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

Google (n.d.). *Tensorflow*.

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