Stockholms universitet

Toward Decoding The Abstract Image Representations in Neural Networks

Martin Björklund

Masteruppsats i matematisk statistik
Master Thesis in Mathematical Statistics

Masteruppsats 2024:7
Matematisk statistik
Juni 2024

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Stockholm University

Mathematical Statistics
Stockholm University
Master Thesis **2024:7**
http://www.math.su.se

Toward Decoding The Abstract Image Representations in Neural Networks

Martin Björklund*

June 2024

Abstract

In this thesis, we present a decoder that reconstructs images from high-dimensional abstract representations inside a neural network. As our model of interest we use a ResNet-18 model trained on the CIFAR-10 image data and investigate 6 checkpoints in the model, which we use as input to the decoder. Drawing inspiration from autoencoders, our decoder mirrors the architecture of ResNet-18, aiming to reverse the sequence of operations that it has applied to the original images in the dataset.

We find that the decoder works well for checkpoints placed early in the network but that the quality of the reconstructed images deteriorates as one moves toward the output layer in the neural network, resulting in a higher mean squared error for the reconstructions. Moreover, we examine how the decoder reconstructs images based on artificially generated abstract representations.

As a further assessment of the decoder's performance, we let ResNet-18 classify each reconstructed image. Based on its accuracy on the reconstructions, we find that the decoder does not generally preserve the features that are necessary for accurate classification. While this could be due to a loss of information in each checkpoint, we hypothesize that it is because of the loss function used for the decoder. We propose introducing suitable regularization terms to the loss function to ensure that the decoder preserves features in the representations that are relevant for classification.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden. E-mail: kmartinbjorklund@gmail.com. Supervisor: Chun-Biu Li.

Acknowledgements

I would like to extend my gratitude to my advisor Chun-Biu Li for his help with this thesis. His ambition, ideas and honest feedback has been truly invaluable throughout the project. Furthermore, I would like to thank Nik Tavakolian for his very helpful feedback and ideas regarding both the project as a whole and the code, as well as for providing the trained ResNet-18 network along with data from the checkpoints.

In order to speed up the writing process, figures 1 and 2 were created using AI tools and then modified.

Contents

1 Introduction

Deep neural networks have become a widespread and powerful tool in many areas of machine learning, with applications ranging from time series analysis to self-driving cars. Despite their widespread use and versatility, they are often considered as "black boxes", concealing their inner workings from the user. Interpretation of a neural network is far from straightforward and in contrast to statistical models such as linear or logistic regression, one cannot directly interpret the estimated parameter values. This is problematic, as interpretation of models is often desired and may lead to better implementation.

The topic of explainable artificial intelligence is an active field of research, and has been investigated in numerous studies. A widespread hypothesis is that neural networks collapse the features of the input that are unnecessary for classification, resulting in representations within the network that only preserve the most important elements of the input. Mahendran & Vedaldi [1] examined this by reconstructing the input images to a convolutional neural network, measuring the loss between the representation of the original image within the network and the representation of the reconstructed image. Köhler [2] opted for a different approach, using techniques from unsupervised learning to examine the inner workings of a convolutional neural network.

In this thesis, we create a *decoder*, a model that aims to reconstruct the input to a neural network from the abstract representations of the data at the hidden layers. Inspired by the structure of autoencoders [3], our decoder mirrors the architecture of the encoding network that we wish to interpret. With this in mind, using a convolutional neural network with image data is a natural choice, since the use of images provides a reconstruction that humans can easily understand. Due to its widespread popularity and success, we use a ResNet-18 model [4] as the model of interest. In it, we place 6 checkpoints from which we input the abstract representations to our decoder, which reconstructs images by minimizing the mean squared error between the original image and the reconstruction.

Aside from simply reconstructing the input images and examining how the decoder's performance changes depending on its input, we examine how well the decoder preserves features that are important for classification by letting ResNet-18 classify the reconstructions. Moreover, we investigate the decoder's ability to generate images artificially and examine how these generated images change as we alter the input.

We find that the reconstructions from early checkpoints are similar to the original images but that their quality deteriorates, as we see blurry reconstructions with a higher loss for later checkpoints. Furthermore, ResNet-18 performs poorly when classifying the reconstructions from all but the first two checkpoints, suggesting that the decoder may not be able to preserve the relevant

features for image classification. While this could mean that the hypothesis we mentioned is wrong, we propose that usage of a different loss function with suitable regularization terms in the decoder could produce better results and suggest such a function.

The rest of the thesis is structured as follows. Section 2 describes the theory and methods necessary for understanding the study, with Subsection 2.5.2 outlining the architecture of the decoder. Next, Section 3 presents the results, followed by the main conclusions in Section 4. The code used to produce the results for this study is available at `https://github.com/martin-bjorklund/decoder_thesis`.

2 Theory and Methods

2.1 Neural Networks

A *neural network* is a type of machine learning model used for supervised, unsupervised as well as reinforcement learning. Neural networks learn to approximate some target function $f^*(\boldsymbol{x})$ by learning parameters $\boldsymbol{\theta}$ for its approximation $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$, where $\boldsymbol{x}$ is the input to the network and $\boldsymbol{y}$ is the output, both of which can be multi-dimensional.

The most basic type of neural network is a *feedforward neural network*, also known as a multilayer perceptron. As explained in Goodfellow *et al* [5], they are so named since information is only propagated forward through the network, in contrast to for example recurrent neural networks, which we do not cover in this thesis.

Every neural network consists of several chained functions. For example, we could have a neural network with the structure $f(\boldsymbol{x}) = f_3(f_2(f_1(\boldsymbol{x})))$. In this case, f_1 is the first *layer* of the model, while f_2 makes up the second layer, and so on. The *depth* of this model is 3, since the network consists of 3 nested functions. Layer 2 is a *hidden layer*, while the final layer is known as the *output layer* and layer 1 is the *input layer*. Typically, neural networks have a depth much greater than 3 and thus have several hidden layers.

Each layer contains a number of *neurons*, the maximum number of which defines the *width* of the network. In Figure 1 we see an example of a simple feedforward neural network whose depth and width are both equal to 3. In this example, each neuron is connected to every neuron in the subsequent layer, meaning that each layer is *fully connected*.

But what does each layer consist of? Typically, in a feedforward neural network, if we let $\boldsymbol{x}$ be a vector-valued input, the layer calculates $\phi(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$, where $\boldsymbol{W}$

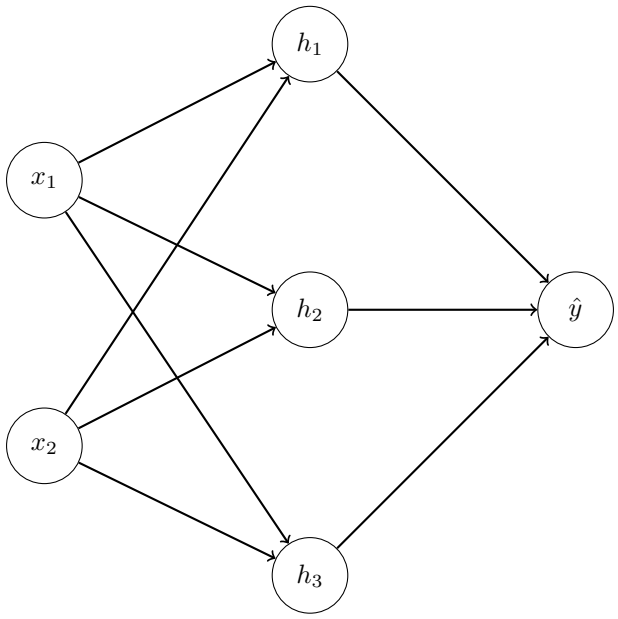


Figure 1: Diagram of a simple neural network. The network has 2 input units, 1 hidden layer with 3 units and one output unit. Each circle represents a neuron.

and $\boldsymbol{b}$ are trainable parameters known as the *weight matrix* and the *bias*. Note that the usage of the word bias in deep learning differs from its usage within statistics. Here, ϕ is some non-linear *activation function* acting element-wise on its input. Many such activation functions exist, one of the most popular choices being the rectified linear unit (ReLU) [6], defined as

$$h(x) = \max(0, x) \tag{1}$$

for some scalar x.

In classification settings, we typically want the output of the model to sum to 1, enabling interpretation of the output as conditional probabilities. For binary classification, one may use the *sigmoid* function, which for some scalar x is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

and may also be used as an activation function in the hidden layers of the model. For multi-class classification, we typically use the *softmax* function, whose ith output element for a vector-valued input $\boldsymbol{x}$ is defined as

$$\text{softmax}(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}}. \tag{3}$$

Usage of either of these functions allows us to interpret their output as the estimated conditional probability $P(\boldsymbol{x}$ belongs to class $i|\boldsymbol{x})$.

The fact that the activation functions are non-linear is important. A network consisting only of linear layers can only model linear functions or perform linear classification. For example, a network for binary classification with no non-linear layers, sigmoidal output and a fixed decision threshold (such as 1/2) can only produce linear decision boundaries, which is equivalent to logistic regression.

In a deep neural network, the dimensionality of the data is transformed in each layer. We call these transformed representations of the data *abstract representations* (also known as *deep* representations). A common strategy is to first increase the dimensionality of the abstract representations and then decrease their size in subsequent layers. It is theorized that such a structure allows the network to first find the relevant features of the data using the high-dimensional abstract representations and then compress and discard features that are irrelevant for the task at hand, compressing the representations to a form that is suitable as output of the network.

2.2 How a Neural Network Learns

When training a neural network, our goal is to find the values of the parameters $\boldsymbol{\theta}$ that minimize the *loss function* (also known as the cost function). This function quantifies the difference between the actual target value and the predicted value of the model, where higher values represent worse predictions. In classification settings, one often uses the *cross-entropy* loss function. For an observation with k classes, we have a length k dummy variable vector $\boldsymbol{y}$ containing the label and a length k vector $\hat{\boldsymbol{y}}$ containing the predicted conditional probabilities. Then, the cross-entropy loss for that observation is defined as

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{j=1}^{k} \boldsymbol{y}_j \log \hat{\boldsymbol{y}}_j, \tag{4}$$

where the index j denotes the jth element of the vectors. This function is bounded by 0 from below and, as we would expect from a loss function, takes on higher values when the true class is assigned a low probability. To calculate the loss for the whole dataset, we take the average loss per observation.

Now, let $\boldsymbol{o}$ be the vector of unnormalized conditional probabilities for an observation, i.e. the output of the final layer of a neural network. A useful feature of the cross-entropy loss is the simplicity of its derivative when used with the softmax function [7]. Substitution of $\hat{\boldsymbol{y}}_j$ with $\text{softmax}(\boldsymbol{o})_j$ followed by standard mathematical operations gives us that

$$\frac{\partial L(\boldsymbol{o}, \boldsymbol{y})}{\partial \boldsymbol{o}_j} = \text{softmax}(\boldsymbol{o})_j - \boldsymbol{y}_j, \tag{5}$$

which is just the difference between the estimated conditional probability of class j and the jth element of the dummy variable vector containing the label.

To understand how we find the minimum of a loss function in deep learning, one must first know the basics of the *gradient descent* algorithm, which works iteratively in order to find the argument that minimizes the function. Let $l(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}; \boldsymbol{\theta})$ be some arbitrary loss function for an observation i, where we have explicitly expressed the loss function's dependence on the parameters. At each step, gradient descent updates the parameters according to the rule

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \epsilon \cdot \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}; \boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t}, \tag{6}$$

where the index t denotes the current step, N is the number of observations and ϵ is a small positive parameter known as the *learning rate*. According to this update rule, we see that if the gradient is positive we decrease the value of $\boldsymbol{\theta}$ and increase it if the gradient is negative. Intuitively, we can think of gradient descent as letting the parameter $\boldsymbol{\theta}$ roll down the surface formed by the loss function until it reaches the bottom.

In practice, the learning rate decreases in each step. If the learning rate was constant, the algorithm could "step over" the minimum several times, going back and forth between different sides of it. Reducing the learning rate in each step reduces the risk of this, allowing the algorithm to make finer adjustments to the parameters in later epochs.

Since the dataset may consist of hundreds of thousands of observations or more and neural networks can have millions of parameters, calculation of the loss for the whole dataset along with the gradients $\partial l(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}; \boldsymbol{\theta})/\partial \boldsymbol{\theta}$ can be very computationally expensive. Therefore, one often uses randomly selected subsets of the original data, known as *mini-batches*. Gradient descent using mini-batches is known as *stochastic graddient descent*, variants of which are used to train neural networks. Since the size of the mini-batch is much smaller than the size of the entire training set, it takes several steps to go through all of the data. One iteration over the whole training set is known as an *epoch*. Letting m represent the number of observations in a mini-batch, each step in stochastic gradient descent updates the parameters according to the rule

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \epsilon \cdot \frac{1}{m} \sum_{i=1}^{m} \frac{\partial l(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}; \boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t}, \tag{7}$$

which is usually done for several epochs.

One of the most widely used variants of stochastic gradient descent is *Adam* [8], which we use when training the models in Section 3. While an in-depth description of the Adam algorithm is outside the scope of this thesis, one should be aware that it introduces several new parameters as well as methods that counteract the risk of getting stuck in local minima.

Of course, in order to use any of these optimization algorithms, we must first calculate the gradients $\partial l(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}; \boldsymbol{\theta})/\partial \boldsymbol{\theta}$. This is done using *forward propagation*

and *backpropagation*.

During forward propagation, we run the data through each layer of the network, from the input layer all the way to the final layer, making sure to store the output from every intermediate layer. Based on this output, the value of the loss function is calculated. Then, during backpropagation, we calculate the gradients of the loss function with respect to the parameters. Using the chain rule of calculus, we first get the gradient of the loss with respect to the parameters of the final layer and then move sequentially backwards through the network until we reach the parameters of the first layer. While this may sound complicated at first, it is merely an application of the chain rule. For a more detailed example of how this process works in practice, see for example Zhang *et al* [7].

2.3 Encoders, Decoders and Autoencoders

An *autoencoder* is a neural network with a symmetric structure, trained to output a representation as similar to its input as possible [3]. While a deep understanding of autoencoders is not necessary for grasping the contents this thesis, this section provides a brief overview of their structure in order for the reader to understand the reasoning behind the methodology we use. For more details on autoencoders, see for example Charte *et al* [3].

An autoencoder can contain an arbitrary number of hidden layers, with a general structure that, as shown in Figure 2, includes an *encoder* f, which maps the input x to an *encoding* y. This encoding is then run through a *decoder* g which aims to create a reconstruction r of the input x [3].

If the encoding y is of lower dimension than the input x, the autoencoder is known as *undercomplete*. This structure allows the encoding to capture only the most important features of the data, enabling us to use it for tasks such as dimensionality reduction. An *overcomplete* autoencoder, on the other hand, has an encoding y that is of higher dimension than the input.

To train an autoencoder one may use the same techniques as for a feedforward neural network, using mini-batches and stochastic gradient descent. Special care needs to be taken using regularization when training an overcomplete autoencoder to make sure that it does not simply learn the identity mapping, which would not be particularly useful. The exact form of the loss function used for an autoencoder depends on the input. A common choice is the mean squared error, but for binary inputs the cross-entropy loss is usually preferred [3].

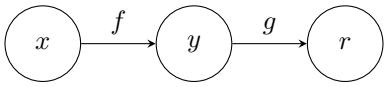


Figure 2: Overview of the general structure of an autoencoder. From Chartre *et al* [3].

2.4 Convolutional Neural Networks

2.4.1 The Convolution Operation

The *convolution operation* plays a fundamental role when processing image data, which are usually represented as tensors or multi-dimensional arrays composed of real values. In this representation, the x and y axes represent the pixel values, while another axis represents the color channels, the number of which depends on the image type. Grayscale images have only one color channel, while colored images usually have three.

So how does one deal with this input? While flattening the image and inputting it as a vector to a feedforward neural network may seem natural, this is highly impractical. Consider a scenario where we aim to classify images as cats or dogs. Connecting each pixel of even a greyscale 1 megapixel image to the nodes of a 1000-node fully connected layer would require $10^6 \cdot 10^3 = 10^9$ weights to be trained [7]. Not only would this approach be computationally infeasible; it would also be unlikely to fully capture the important aspects of the image.

Aside from this, we also want our network to give us the same prediction no matter where in the image the cat or dog is located. Clearly, simply flattening the input image would not yield appropriate results.

To address these problems, we use the convolution operation. For an input image I with two axes and a *kernel* K with the same number of axes as the image, element i, j of the convolution output (also known as the *feature map*) can be written as

$$(I * K)_{i,j} = \sum_m \sum_n I_{i+m,j+n} K_{m,n}. \tag{8}$$

The kernel K is a multidimensional array of trainable parameters, whose values are found by the training algorithm. Usually being of a smaller height and width than the input image but with the same number of channels, common choices are 3×3 and 5×5 kernels [5]. An illustration of a convolution can be found in Figure 3.

This operation solves the potential problems we introduced in the beginning of this subsection. By letting the kernel be of a smaller size than the input, we allow for parameter sharing between the pixels instead of letting each one be

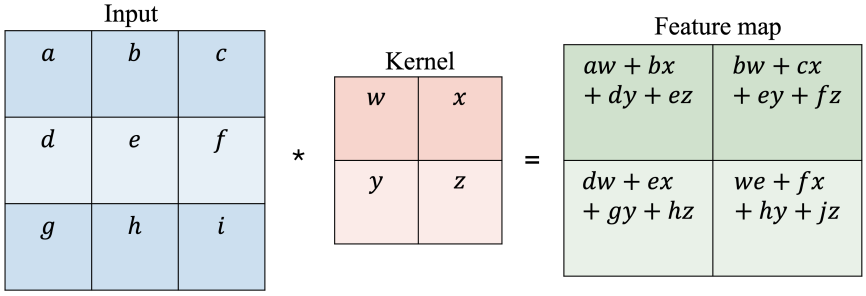


Figure 3: Illustration of a convolution with a 3×3 input and a 2×2 kernel with a stride of 1 and no padding.

associated with its own parameter. This greatly reduces the number of trainable parameters required, which in turn decreases computational costs in the form of memory requirements and training time.

Moreover, the convolution operation's *equivariance to translation* solves the issue of getting the same prediction independently of the location of an object within an image. Formally, the equivariance to translation can be expressed as $f(g(x)) = g(f(x))$ for a convolution operation f and some translation g.

To control the size of the feature map, we may adjust the *zero padding* and *stride*. Introducing a padding of p appends p zero-valued rows and columns on each side of the input. Besides enlarging the feature map, this also allows us to make better use of the corner values of the input, which are otherwise only taken into account once per observation.

The stride, on the other hand, controls the step size of the kernel as it slides over the input. A higher stride lets the kernel skip over some elements of the input and leads to a smaller feature map that does not extract the features of the image as finely as with a low stride.

Sometimes, we may want the feature map to have a different number of channels compared to the input. Assume we have c_{in} input channels and c_{out} output channels. We then need a 4-dimensional kernel K of dimensions $c_{\text{in}} \times c_{\text{out}} \times k_h \times k_w$, where k_h and k_w denote the height and width of the kernel, respectively. Using a 3-dimensional input I, where the first dimension corresponds to the number of input channels, and a stride of s, element i, j, k of the feature map of such a convolution can be described by the formula

$$(I * K)_{i,j,k} = \sum_{l,m,n} I_{l,(j-1)s+m,(k-1)s+m} K_{i,l,m,n}. \tag{9}$$

A typical convolutional neural network does not only use convolutional layers, but also fully connected layers as well as other operations described in the

remainder of this section.

2.4.2 Batch Normalization

Batch normalization, first introduced by Ioffe and Szegedy [9], aims to address the problem they name "internal covariate shift", where the inputs to activation functions in the hidden layers of a neural network can be on widely different scales at different points during the training process and at different points in the network. They theorized that internal covariate shift slows down the training of a neural network by forcing the parameters to constantly adapt to the differing scales of the data which they act upon.

Batch normalization ensures that the input to each layer during training has mean 0 and unit variance along each dimension. While it is standard practice to normalize the input to the network, the novelty in this method lies in doing so for each mini-batch and before each activation function. Intuitively, one might imagine that this helps the training process by letting the inputs to the activation functions be on similar scales, thereby reducing the internal covariate shift. However, Santurkar *et al* [10] suggest that the reason why batch normalization works is not because it reduces the internal covariate shift, but because it makes the optimization landscape smoother. For more details on this, see the original paper [10].

Mathematically, the operation can be described as follows during training. Let $\boldsymbol{x}$ be an element of a mini-batch $\mathcal{B}$, $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ be the mean vector of the mini-batch and $\hat{\boldsymbol{\sigma}}^2_{\mathcal{B}}$ be vector containing the variance of the mini-batch along each dimension. For some learnable parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ as well as some small constant ϵ, the batch normalization operation is defined as

$$\mathrm{BN}(\boldsymbol{x}) = \boldsymbol{\gamma} \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\sqrt{\hat{\boldsymbol{\sigma}}^2_{\mathcal{B}} + \epsilon \mathbf{1}}} + \boldsymbol{\beta}, \tag{10}$$

where the multiplication, division and square root is performed element-wise and $\mathbf{1}$ is a vector with each element equal to 1. The parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ recover the degrees of freedom lost by normalization and ensure that the operation is able to represent the identity transform, while the constant ϵ provides numerical stability and prevents division by 0.

Once the network has been trained, the mini-batch statistics $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ and $\hat{\boldsymbol{\sigma}}^2_{\mathcal{B}}$ are replaced with their counterparts based on the whole training set, meaning that batch normalization behaves differently during training and evaluation.

In convolutional neural networks, batch normalization is usually placed after the convolution operation and before the activation function [7]. With a mini-batch of size m and a convolution with output width w and height h for each channel, the batch normalization is done over all $m \times h \times w$ elements per output channel,

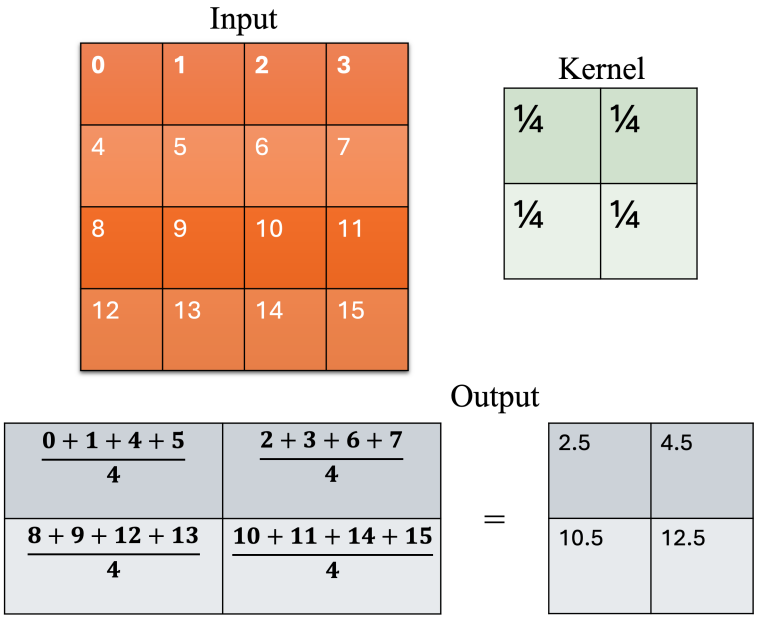


Figure 4: An illustrated example of average pooling with an input of size 4×4, a kernel of size 2×2 and a stride of 2. Note how the output, displayed as a gray matrix, contains the average value of different sections of the input.

independently of the other channels. Each channel is therefore associated with its own mini-batch statistics and elements of the learnable parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$.

2.4.3 Average Pooling

It is common practice to use *pooling operations* in convolutional neural networks. These operations replace the outputs of layers with summary statistics of the nearby elements, which introduces approximate translational invariance, meaning that even if the input to the pooling operation is shifted by some small amount, most of its output remains the same. This is a useful property for image classification tasks, since we as previously mentioned care more about whether or not a feature is present in the data than its exact location [5].

One such pooling operation, which we use in this thesis, is *average pooling*. Consider an input $\boldsymbol{x}$ of size $C \times H \times H$, a kernel K of size $C \times k \times k$ and a stride of s. Performing average pooling with these arguments yields an output of size $C \times H_{\text{out}} \times H_{\text{out}}$, where

$$H_{\text{out}} = \frac{H - k}{s} + 1,$$

and element (c, h, w) of the output of such an operation is defined as [11]

$$\mathrm{AP}(\boldsymbol{x})_{c,h,w} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} \boldsymbol{x}_{c,s\cdot h+m,s\cdot w+n}. \tag{11}$$

Note that in contrast to convolutions, the kernel used for average pooling contains no trainable parameters, as each element of the kernel is equal to $1/k^2$.

A visualization of an average pooling operation with a stride of 2, an input of size 4×4 and a kernel of size 2×2 with 1 channel is provided in Figure 4.

2.4.4 Residual Connections

Residual connections, introduced by He *et al* [4], address the *degradation issue*, where deep networks perform worse than their shallow counterparts. Assume our goal is to model some target function f^* and that our network with l layers is able to model a class of functions $\mathcal{F}_l$ using different parameter settings. If f^* is within $\mathcal{F}_l$ we may be able to reasonably approximate f^*, but this is often not the case. To achieve better performance and bring $\mathcal{F}_l$ closer to f^*, one might be tempted to increase model complexity by adding another layer. However, this could backfire and actually bring us further away from f^*, since the new layer may not be able to represent the identity function and thus $\mathcal{F}_l$ is not guaranteed to be a subset of $\mathcal{F}_{l+1}$ [7].

Residual connections solve this issue by adjusting the target function for each layer. Instead of letting a layer approximate a function $f(\boldsymbol{x})$ for some input $\boldsymbol{x}$, it is made to learn the residual mapping $g(\boldsymbol{x}) = f(\boldsymbol{x}) - \boldsymbol{x}$. This allows each layer to learn the identity mapping should it be necessary, ensuring that $\mathcal{F}_l \subseteq \mathcal{F}_{l+1}$ for all $l \geq 1$. Thus, adding more layers with residual connections guarantees that we are always able to more closely approximate the target function f^*. Of course, getting better results is not always guaranteed, due to the stochastic nature of training neural networks.

This mathematical approach may be useful for understanding why residual connections work, but one can also think of residual connections as preventing the loss of information, "reminding" the network of the original image. An illustration of a layer utilizing a residual connection is provided in Figure 5.

2.4.5 ResNet-18

The network of interest in this thesis, ResNet-18 [4], makes use of many of the deep learning methods we have discussed so far. As is common for this

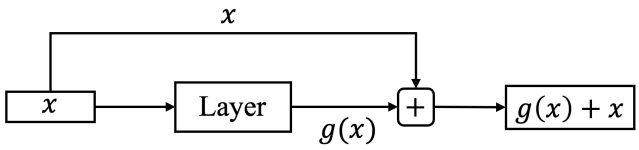


Figure 5: Schematic of a layer utilizing a residual connection.

architecture, we use it for classification of images. The usage of ResNet models has become widespread thanks to their high accuracy in this context.

ResNet-18 is built using a repeating block-type structure. The first layer of the network is a convolutional layer with a 3×3 kernel followed by batch normalization and the ReLU activation function. This is then followed by four *residual blocks*, each of which contains four convolutional layers, all followed by batch normalization and the ReLU activation function. Each residual block also contains two residual connections. After these blocks follows an average pooling operation with a 4×4 kernel and finally a fully connected layer. An overview of this structure is displayed in Figure 6.

Some of the residual connections utilize a convolution operation with a 1×1 kernel followed by batch normalization, in order to account for the fact that the number of channels changes during the first convolution of the blocks. This only applies to the first residual connection of blocks 2, 3 and 4, since this is where the number of channels change.

Most convolutions in the network utilize a padding of 1 and stride of 1. Exceptions are the first convolutions in blocks 2-4, which use a stride of 2, and the 1×1 convolutions, which use a stride of 1 and no padding.

As is the case with most neural networks, the data are transformed as they pass through the network, changing dimensionality by passing through the layers. As input to the model, we use images of size $3 \times 32 \times 32$, described in Section 3. The first convolutional layer of the model increases the number of channels to 64, while the first convolutional layer of residual blocks 2-4 each double the number of channels and halve the height and width of the abstract representations. The average pooling operation performs further downsampling, yielding a representation of dimensions $512 \times 4 \times 4$. Finally, the output of the final layer of the model is a $10 \times 1 \times 1$ array, where element i is the unnormalized conditional class probability of class i, meaning that the output does not sum to 1. To allow interpretation of the output as conditional probabilities, one may use the softmax function as described in (3).

By first increasing the dimensionality, it is theorized that ResNet-18 can more easily identify the features separating each class, as we have already mentioned

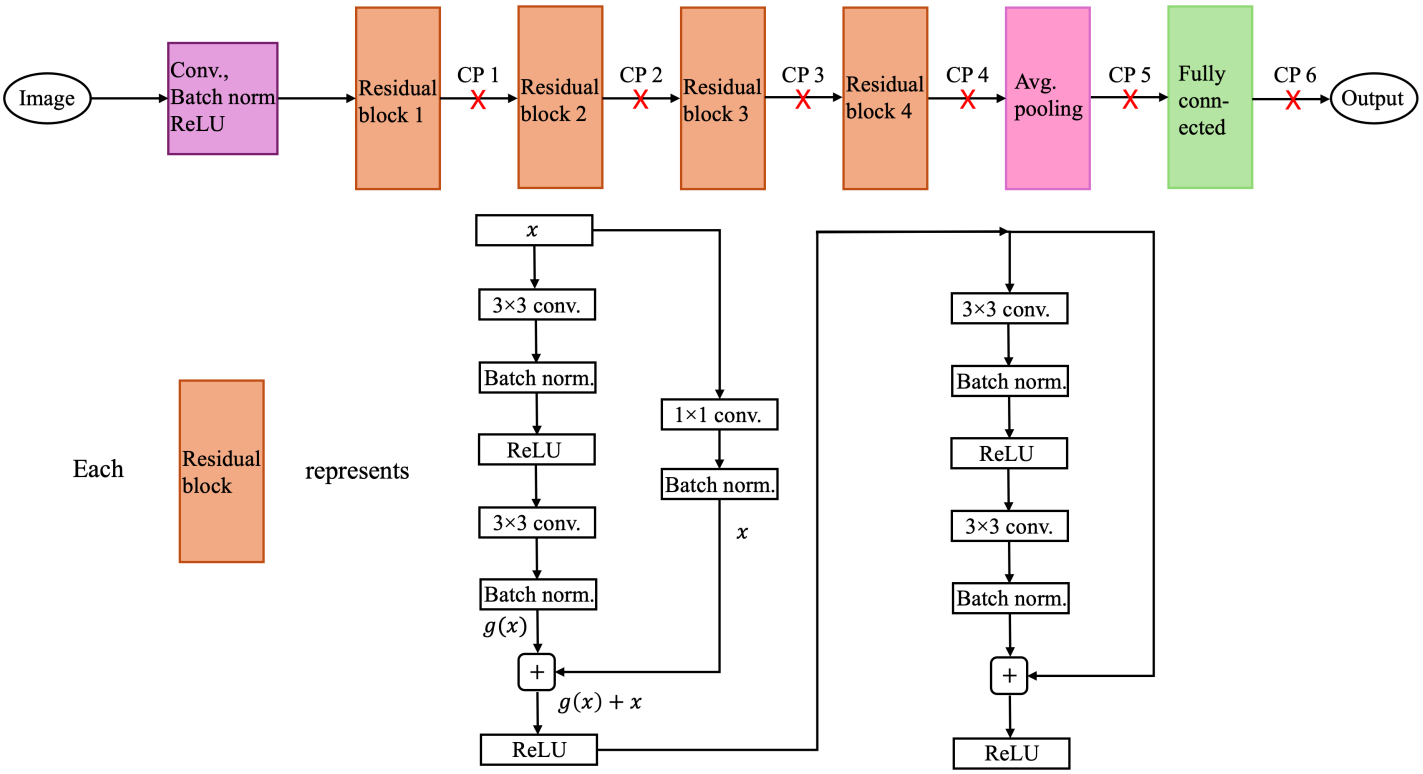


Figure 6: Overview of the ResNet-18 architecture. Adapted from Köhler [2]. Each red X marks the location of a checkpoint (CP). Note that the 1×1 convolution in the residual connection is not used in block 1, since no change in dimensionality occurs.

Table 1: Dimensionality of the representation in each checkpoint, denoted as channels × height × width.

	Dimensionality
Checkpoint 1	$64 \times 32 \times 32$
Checkpoint 2	$128 \times 16 \times 16$
Checkpoint 3	$256 \times 8 \times 8$
Checkpoint 4	$512 \times 4 \times 4$
Checkpoint 5	$512 \times 1 \times 1$
Checkpoint 6	$10 \times 1 \times 1$

in Section 2.1. Then, as we decrease the dimensionality, the model may discard irrelevant features and compress the representation to a form that is suitable as output of the network.

In the network, we place 6 *checkpoints*; one after each residual block, one after the average pooling operation and one after the fully connected layer. Using the abstract representations of the data in these checkpoints, we aim to reconstruct the original images using a decoder, mirroring the architecture of ResNet-18. The dimensionality of the representation in each checkpoint is shown in Table 1.

2.5 Decoding the Abstract Representations

2.5.1 Transposed Convolution

While the convolution operation typically decreases the size of its input, we have so far not discussed ways to increase the size of something, which is a necessary property of the decoder we construct in this thesis. *Transposed convolutions*, sometimes known as *deconvolutions*, meet this need. Although upsampling using a convolution is also possible, it is not as computationally efficient due to the amount of padding required [12].

The transposed convolution is not an exact inverse of the convolution operation. Instead, it can take the output of a convolution operation and recover the *shape* of its original input. For an understanding of how the transposed convolution works in practice it is best to see an illustration of it, as shown in Figure 7. As we can see, the transposed convolution also makes use of a kernel of trainable parameters that slides across the input, performing element-wise multiplication and then addition.

In Algorithm 1 we present the basic algorithm for a transposed convolution using one channel.

For multiple channels, the operation is similar to what we saw for the convolution operation. That is, for c_{in} input channels and c_{out} output channels, we need a kernel K of dimensions $c_{\text{out}} \times c_{\text{in}} \times k_h \times k_w$. We perform the transposed convolution as described in Algorithm 1 a total of c_{in} times for a single output channel and add the output matrices Y. We repeat this process for every output channel and thereafter concatenate the results, which results in an output of dimensions $c_{\text{out}} \times k_h \times k_w$.

Algorithm 1 The transposed convolution algorithm with one channel and a stride of s. Here, $Y_{i:i+h,j:j+w}$ denotes the elements of Y in rows i to $i+h$ and columns j to $j+w$.

Require: Kernel matrix K of dimensions $k_h \times k_w$
Require: Input matrix I of dimensions $h \times w$
Require: Stride s
$Y \leftarrow (s(h-1)+k_h) \times (s(w-1)+k_w)$ matrix of zeros
for $i \leftarrow 1$ **to** h in steps of size s **do**
    for $j \leftarrow 1$ **to** w in steps of size s **do**
        $Y_{i:i+h,j:j+w} = Y_{i:i+h,j:j+w} + X_{i,j} \cdot K$
    end for
end for
return Y

The transposed convolution is named due to its relationship with the convolution operation when viewed as a matrix multiplication. Let us consider a convolution with 2-dimensional input X and output Y. If we vectorize the input and output of the convolution operation by unrolling them from left to right and top to bottom, denoting them as X_{v} and Y_{v}, we can represent the convolution operation as a matrix multiplication of the form

$$Y_{\text{v}} = CX_{\text{v}}, \tag{12}$$

where C is a sparse matrix whose non-zero elements are taken from the kernel of the convolution [12]. With this representation of the convolution operation, we have also defined a transposed convolution by instead using the transpose of the sparse matrix, C^T. With this in mind, one may also think of a transposed convolution where the forward propagation and backpropagation have been swapped, since backpropagation is an application of the chain rule and

$$\frac{\partial Y_{\text{v}}}{X_{\text{v}}} = C^T.$$

Again, note that the transposed convolution is not an inverse of the regular convolution and that there exist more than one transposed convolution able to recover the shape of the input to a convolution operation. Suppose we wish to reverse a convolution operation and find a representation that not only has the same shape as its input, but also the same contents, as is the goal in the

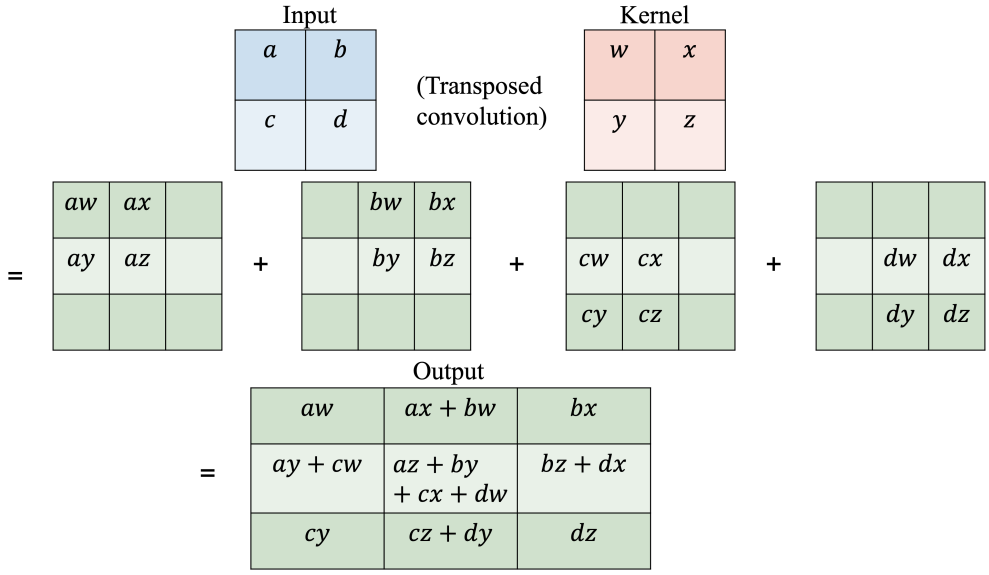


Figure 7: An illustration of a transposed convolution with a 2×2 input, a 2×2 kernel, a stride of 1 and no padding. Transposed convolutions typically result in an output of larger size than its input.

decoder we outline in Section 2.5.2. By using trainable parameters in the kernel of the transposed convolution, as opposed to just using C^T of the corresponding convolution operation, we are able to get a representation that resembles the input to the convolution operation much more closely.

Just like with regular convolutions, transposed convolutions can implement padding and stride. However, the effect they have on the output size of the operation is reversed. While an increased stride in the convolution operation decreases the size of the output, it *increases* the size of the output for transposed convolutions. Just like with regular convolutions, the stride specifies the number of "steps" the kernel takes when moving over the input.

Padding, on the other hand, works quite differently. Using a padding of p in a transposed convolution amounts to removing the p outermost rows and columns on each side of the output [7], thereby decreasing the output size.

2.5.2 Decoder Architecture

Drawing inspiration from autoencoders, we design a decoder that mirrors the architecture of the encoder, ResNet-18. Using this approach, we construct a decoder that aims to reverse the operations made by ResNet-18, thereby re-

constructing the original images from their abstract representations in each checkpoint shown in Figure 6.

To reverse the convolution operation, it is natural to use a transposed convolution with the same kernel size, stride and padding, as this results in a tensor of the appropriate size. However, reversing the average pooling operation is not as straightforward. We therefore introduce an *unpooling* operation. We unpool the elements of each channel independently, expanding the $512 \times 1 \times 1$ representation to a tensor of size $512 \times 4 \times 4$ by replicating the values in the smaller representation four times. After this, we perform an element-wise multiplication of the $512 \times 4 \times 4$ representation with a tensor of learnable parameters of the same size. While this is not an exact inverse of average pooling, the hope is that the multiplication by learnable parameters allows us to more accurately reconstruct the representation by adding degrees of freedom.

Just like ResNet-18, our decoder is built in a block-like structure. Since we for each checkpoint only need to reverse the transformations that ResNet-18 has applied to the data so far, we train a separate decoder for each checkpoint. Each one only applies the transformations necessary to decode the corresponding checkpoint, meaning that the last decoder is far deeper than the first.

This is best understood by looking at Figure 8. As shown there, the full decoder used for the final checkpoint contains a fully connected layer, an unpooling layer, four decoder blocks and finally a transposed convolution. The decoder for the fifth checkpoint removes the fully connected layer, and the decoder for the fourth checkpoint also removes the unpooling layer. In each subsequent decoder for the remaining checkpoints, we remove one block. Thus the decoder for the first checkpoint only contains one decoder block and a transposed convolution layer.

Each block of the decoder mirrors the corresponding residual block of ResNet-18, replacing each convolution with a transposed convolution as shown in Figure 9. The majority of the transposed convolutions use a stride and padding of 1. In order to fully mirror the residual blocks, the exceptions here are in the *final* transposed convolutions of blocks 1-3 instead of the first. Thus, the final transposed convolution in blocks 1-3 use a padding of 1 but a stride of 2. Also note how the 1×1 transposed convolution, using no padding and a stride of 1, occurs in the second residual connection of blocks 1-3 instead of the first.

Since we need to estimate the value of each pixel in every channel based on an abstract image representation, reconstruction of an image amounts to a multivariate regression problem. For this reason, we use the mean squared error (MSE) as our loss function. For an image size of $C \times H \times W$, the MSE for the ith image $Y^{(i)}$ and its reconstruction $\hat{Y}^{(i)}$ is defined as

$$\text{MSE}^{(i)} = \frac{1}{C \cdot H \cdot W} \sum_{c} \sum_{h} \sum_{w} (Y^{(i)}_{c,h,w} - \hat{Y}^{(i)}_{c,h,w})^2, \tag{13}$$

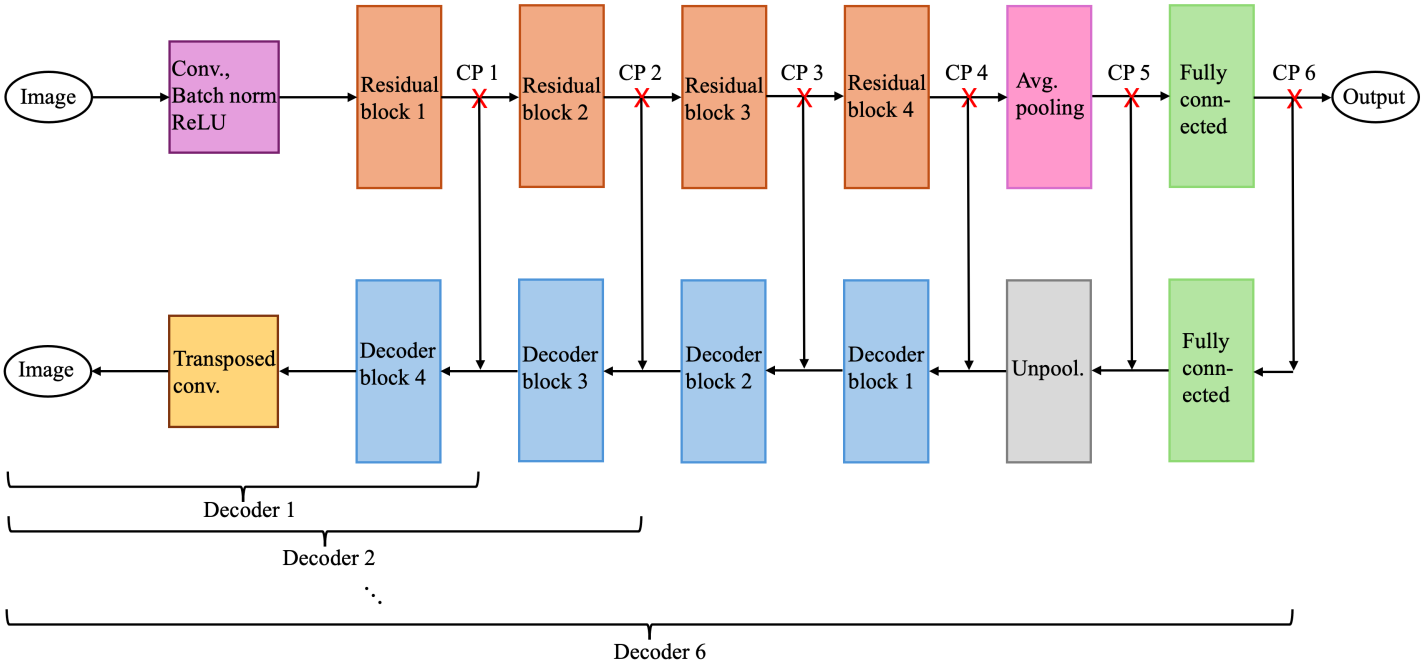


Figure 8: Overview of how data flows from ResNet-18 (top) to the decoders (bottom). Each decoder mirrors the part of ResNet-18 that it aims to reverse.

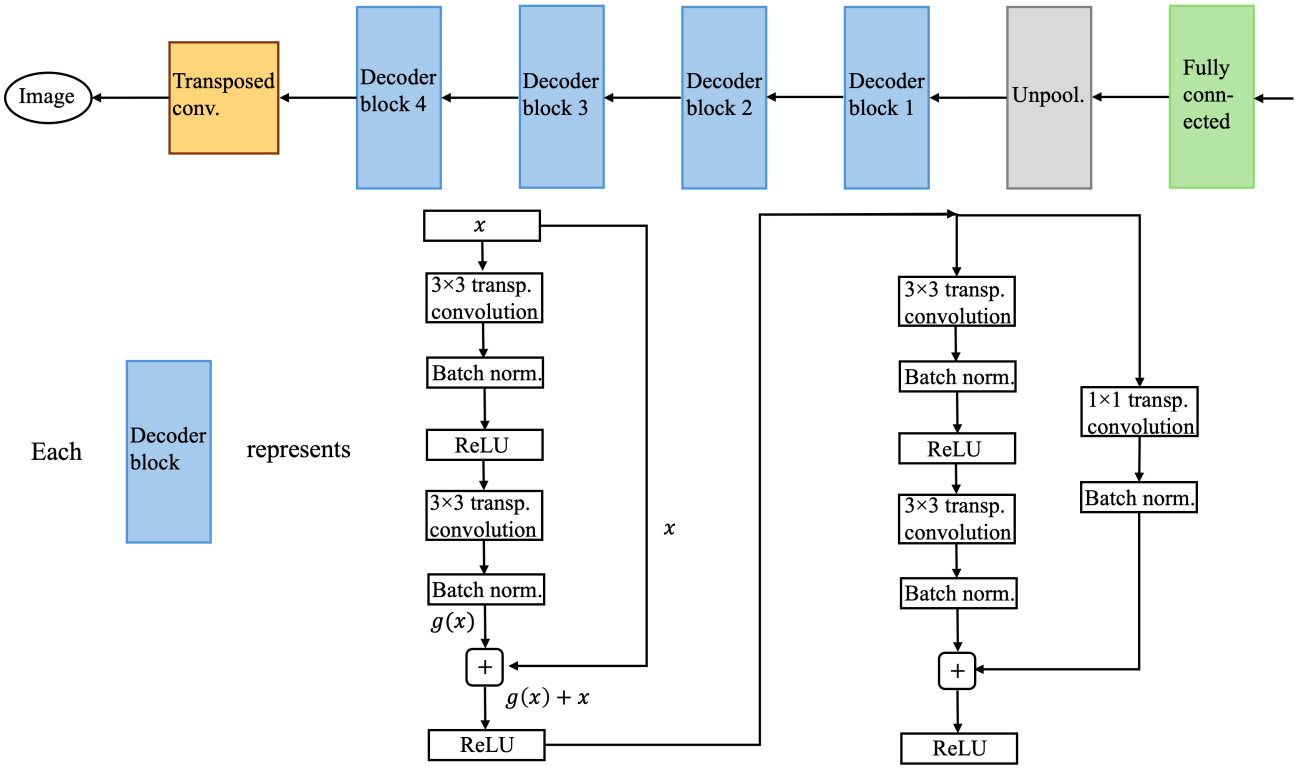


Figure 9: Overview of the full decoder architecture. Note that the 1×1 transposed convolution in the residual connection is not present in decoder block 4.

Figure 10: One example image per class from the training set of CIFAR-10. Due to the low resolution of the images, some may even be difficult for humans to recognize.

where the subscript c, h, w denotes pixel h, w in channel c. The MSE for one mini-batch is then the average MSE of its elements.

3 Results

3.1 Data

As our dataset, we use CIFAR-10 [13]. Widely used in machine learning, it consists of 60000 32×32 pixel color images of different objects and animals, with each image represented as a $3 \times 32 \times 32$ tensor. Every image belongs to one of 10 classes, with there being 6000 observations per class. The dataset is split into a training set of 50000 observations and a testing set of 10000 observations, with an equal number of elements from each class.

In Figure 10, we see one example per class from the training set. The images may seem small and blurry, but the data are useful in part due to this very property, since the small size of each image reduces computational requirements. Moreover, the widespread use of the dataset facilitates reproducibility and recognition.

3.2 Training Process

Prior to training, each image is normalized by subtracting the mean of each channel and dividing by the standard deviation, based on the training set. These normalized images are used as input to our pre-trained ResNet-18, which is trained using Adam and the cross-entropy loss function.

The decoders are trained sequentially, starting from decoder 1 moving up to decoder 6. For decoders 2-6, we use the corresponding parameter values from the previous decoder as initial values, while the remaining parameters are initialized randomly. This saves training time and allows us to use information from the previous decoder. Since all parameters are still kept trainable, the risk that a decoder copies the mistakes of the previous one is low. The randomly initialized parameters are drawn from a $U(-\sqrt{t}, \sqrt{t})$ distribution, where for the fully connected layer $t = 10$ and for the transposed convolutions $t = 1/(c_{\text{out}}k^2)$ where c_{out} is the number of output channels of the operation and k is the kernel size.

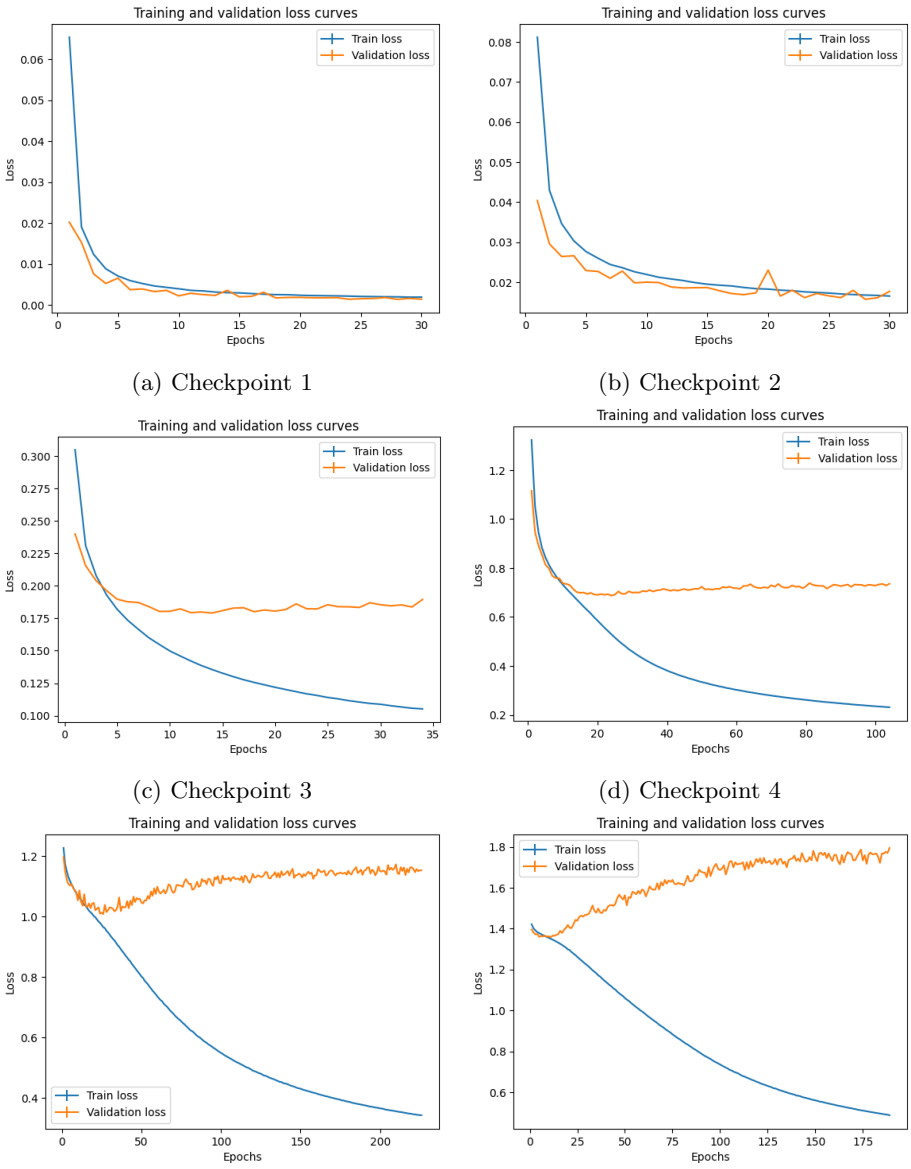
In Figure 11, we see the training and validation loss curves for each decoder. All are trained using Adam until the training loss starts to saturate. This takes more epochs for later checkpoints, presumably since those representations contain less information about the original image. To prevent overfitting, we use the weights that gave the lowest validation loss for each decoder. All training and implementation is done using PyTorch [14].

3.3 Decoder Performance

In Figure 12, we show the mean squared error (MSE) of the decoder for each checkpoint. As we can see, the MSE remains low for early checkpoints, but rapidly increases after checkpoint 3. This pattern of increased MSE for later checkpoints is unsurprising. Not only does the dimensionality decrease as we move to later checkpoints, which causes a collapse of information, but the representations likely also carry less information as features of the image that are irrelevant to classification are discarded in layers of ResNet-18 that are closer to the output. For the sixth checkpoint in particular, the reconstructed images differ strongly from the original images, resulting in a large MSE.

Before visualizing any of the reconstructed images, we need to decide which ones to examine. As we have mentioned, the final checkpoint of ResNet-18 is a vector of length 10 containing the unnormalized probabilities for each class. That is, a high value in element i of the vector indicates that ResNet-18 assigns a high probability to that class.

Figure 13 visualizes the data points from the validation set projected at the final



(a) Checkpoint 1

(b) Checkpoint 2

(c) Checkpoint 3

(d) Checkpoint 4

(e) Checkpoint 5

(f) Checkpoint 6

Figure 11: Loss per epoch when decoding each checkpoint. For the training loss to saturate in later checkpoints, more epochs are required.

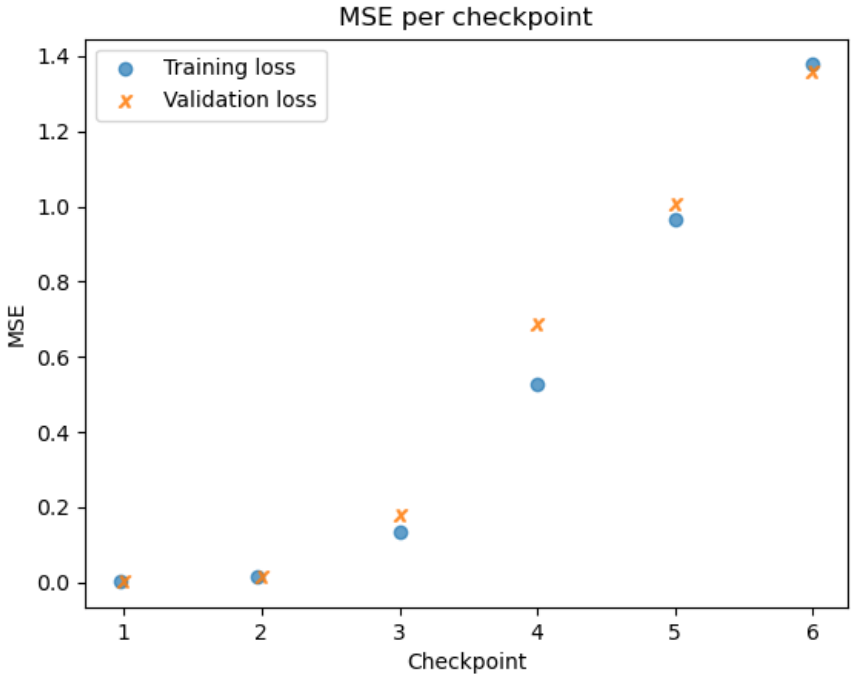


Figure 12: Decoder mean squared error per checkpoint. The error increases abruptly after checkpoint 3.

checkpoint. Each point in the figures represents one image from the validation set, with the axes representing the unnormalized probability for each class. Generally, we see a kind of arrow shape along each axis, where observations at each arrowhead are assigned a high probability of belonging to the corresponding class. For classes that are easily confused by ResNet-18, such as cats and dogs, we see a more triangular shape with the observations being less clustered around a particular axis.

Since the final checkpoint contains information about ResNet-18's classification of the image, it is possible that images that are correctly classified, or are assigned a high probability to the correct class, appear completely differently when reconstructed, as compared to those that are incorrectly classified. Therefore, we reconstruct images belonging to each of these groups. That is, we reconstruct sample observations that are both close to the arrowhead and close to the origin in Figure 13.

Figures 14 and 15 show reconstructed images for observations that ResNet-18 originally classified correctly and incorrectly, respectively. The columns are

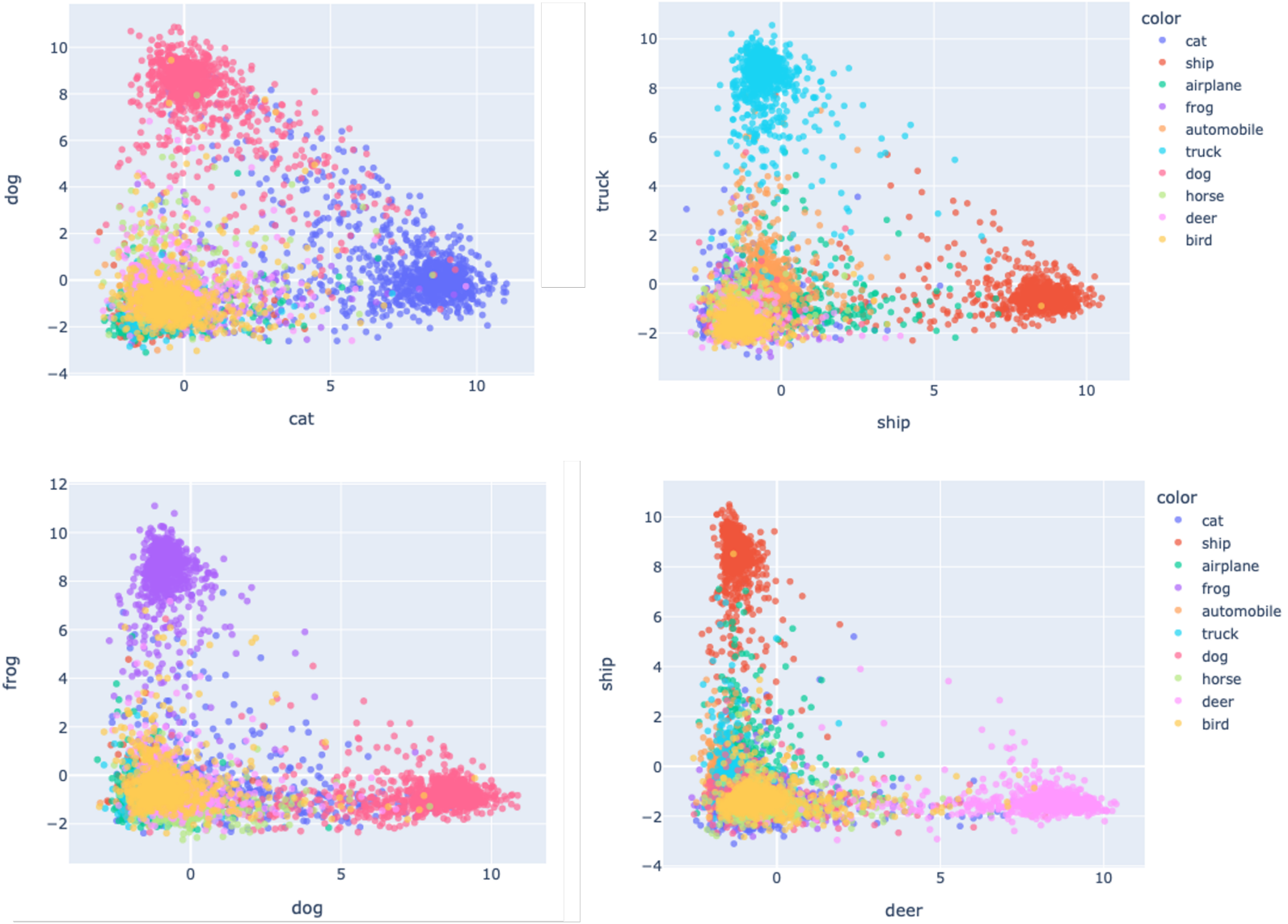


Figure 13: Distribution of elements in the validation set in the final checkpoint for some example classes. Along each axis, the datapoints form an arrow-like shape.

labelled by the checkpoint number, with the first column containing the original image from the validation set. Each row is labelled by the correct class. The correctly classified images shown are all assigned a high probability to the correct class and are therefore located in the arrowhead corresponding to the correct class in Figure 13.

In both figures, we see the same pattern as for the MSE. That is, the reconstructions get blurrier and less similar to the original as we move to later checkpoints. The reconstructions based on the first two checkpoints are nearly indistinguishable from the original image, while the reconstructions from the sixth checkpoint are unrecognizable. We also note a sudden change between checkpoint 4 and 5, with the reconstructions becoming blurry after this in both figures.

Two noteworthy examples illustrating the loss of information in later checkpoints are the automobile and the horse in Figure 14. The reconstructions get blurry for late checkpoints, with a sudden shift in color occurring in checkpoint 5 for the automobile and checkpoint 6 for the horse. Despite starting with a white horse and automobile, we end up with the reconstruction of a red object for the automobile and a brown object for the horse. This clearly shows that information about the specific features of the original image is not preserved by either ResNet-18 or the decoder.

3.4 ResNet-18's Performance on Reconstructed Images

To further evaluate the performance of each decoder, we let ResNet-18 classify the reconstructed images from each checkpoint and examine its accuracy. This way, we plan to determine if the decoder preserves the features necessary for classification.

In Figure 16, we report the accuracy of ResNet-18 on the reconstructed images from each checkpoint based on the validation set. Based on what we saw regarding the MSE in Figure 12, we see an expected pattern, with a high accuracy in the first two to three checkpoints that then drops off beyond checkpoint 3. After this we see an accuracy of around 20%. While better than random guessing, this is not particularly impressive. Looking back to the sample images shown in Figures 14 and 15, we see ResNet-18's classification of the reconstructions underneath each image.

The results from running ResNet-18 on the reconstructed images further show us that the features relevant for classification of the images are not preserved in the reconstructions from checkpoint 4 and onwards. There are two main possible reasons for this. First, as we have already mentioned, we naturally lose more information about the original images, as the dimensionality is lower for the later checkpoints. The second and perhaps most important reason has to do with our choice of loss function for the decoder. Since the mean squared error

Figure 14: Examples of decoded images from each checkpoint, along with the corresponding original image which was correctly classified by ResNet-18. Each row is labelled with the class of the original image. Underneath each image, the prediction of ResNet-18 is displayed.

Figure 15: Examples of decoded images from each checkpoint, along with the corresponding original image which was incorrectly classified by ResNet-18. Each row is labelled with the class of the original image. Underneath each image, the prediction of ResNet-18 is displayed.

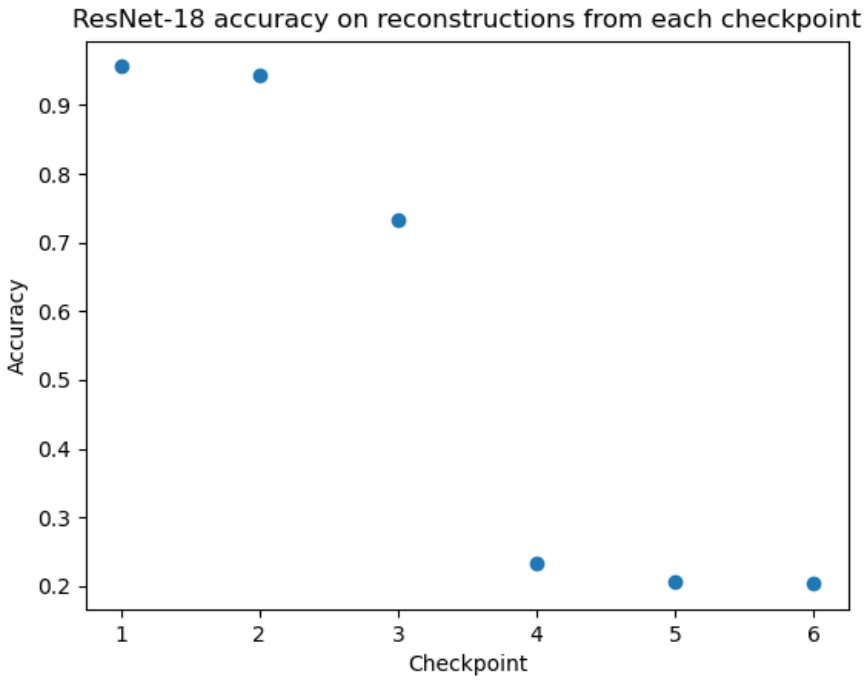


Figure 16: ResNet-18 accuracy on the images reconstructed from each checkpoint, based on the validation set. The reconstructed images based on early checkpoints preserve the features necessary for classification, while reconstructions based the later checkpoints do not.

only measures the difference between the reconstruction and the original image, we argue that the decoder itself introduces further loss of information about the images, as *the loss does not take the class into account*. We elaborate further on this in Section 4.

3.5 Decoding Artificial Data

Since the representation in the final checkpoint is low-dimensional, it is easy to generate artificial data in this representation that could potentially allow us to construct images showing how ResNet-18 views a "typical" element of each class, as a sort of reference image. Recall from Table 1 that in the final checkpoint, each data point is represented as a vector of length 10. To create such artificial data, we set all the elements of this vector to 0 and change the element corresponding to the class we are interested in. Formally, letting $\boldsymbol{z}$ be

such a vector, we then have that $\boldsymbol{z}_j = 0$ for all $j \neq i$, where i is the index of the class we wish to investigate.

As we have discussed relating to the visualizations of the data in the final checkpoint in Figure 13, a high value of $\boldsymbol{z}_i$ for an observation indicates that ResNet-18 assigns a high probability to class i. Setting $\boldsymbol{z}_i$ to a high value in our artificial data and decoding the vector should therefore, in theory, result in an image that ResNet-18 classifies as class i, provided that our decoder preserves the information necessary for classification.

However, this is not the case in practice. In Figure 17, we see images generated based on artificial data. In each row, we vary the element of the vector $\boldsymbol{z}$ that corresponds to the class labelled on the left, while the columns label the value of this element $\boldsymbol{z}_i$. Again, underneath each image we show ResNet-18's classification of it. As we can see, the artificial images are not classified in the way one might expect. In fact, ResNet-18 views all of the artificial images in Figure 17 as either ships or frogs.

When comparing to what we saw in Figures 14 and 15, this is perhaps not so unexpected after all. As we have already discussed, the reconstructions based on later checkpoints do not contain the features necessary for correct classification by ResNet-18.

Despite the unexpected classifications by ResNet-18, we are able to make some interesting observations. First of all, we note that the artificially generated images using high values of $\boldsymbol{z}_i$ look very similar to the pictures from checkpoint 6 in Figure 14. For example, the automobile is red in both Figure 14 and 17.

Secondly, we see a sudden change in the artificially generated images in Figure 17 after $\boldsymbol{z}_i > 7$. Recall that to get the conditional probabilities, we use the softmax function, whose ith element for a vector input $\boldsymbol{z}$ is defined as

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{\boldsymbol{z}_i}}{\sum_j e^{\boldsymbol{z}_j}}.$$

Due to the softmax's usage of the exponential function, it is natural that such a sudden change occurs, since a unit change in $\boldsymbol{z}_i$ results in a large change in $\exp(\boldsymbol{z}_i)$. However, note that with $\boldsymbol{z}_i = 7$ we get an estimated conditional probability of $\text{softmax}(\boldsymbol{z})_i \approx 0.9919$, while using $\boldsymbol{z}_i = 8$ results in $\text{softmax}(\boldsymbol{z})_i \approx 0.9970$, so the location of this change may be surprising. Nevertheless, in Figure 18, we once again visualize the data points projected at the final checkpoint, but this time based on the training set. We find that there is a sudden change in the density around $\boldsymbol{z}_i = 7$ for each of the sample classes. This means that decoding artificial data where $\boldsymbol{z}_i = 7$ (and $\boldsymbol{z}_j = 0$ for $j \neq i$) often amounts to extrapolation, which may be one of the reasons for the location of the sudden change in the images in Figure 17.

Figure 17: Images generated based on artificial data. Each row is labelled with the class whose corresponding element we alter in the vector $\boldsymbol{z}$. Remaining elements of the vector are set to 0. Columns are labelled according to the value of the non-zero element. Underneath each image, ResNet-18's classification of it is displayed.

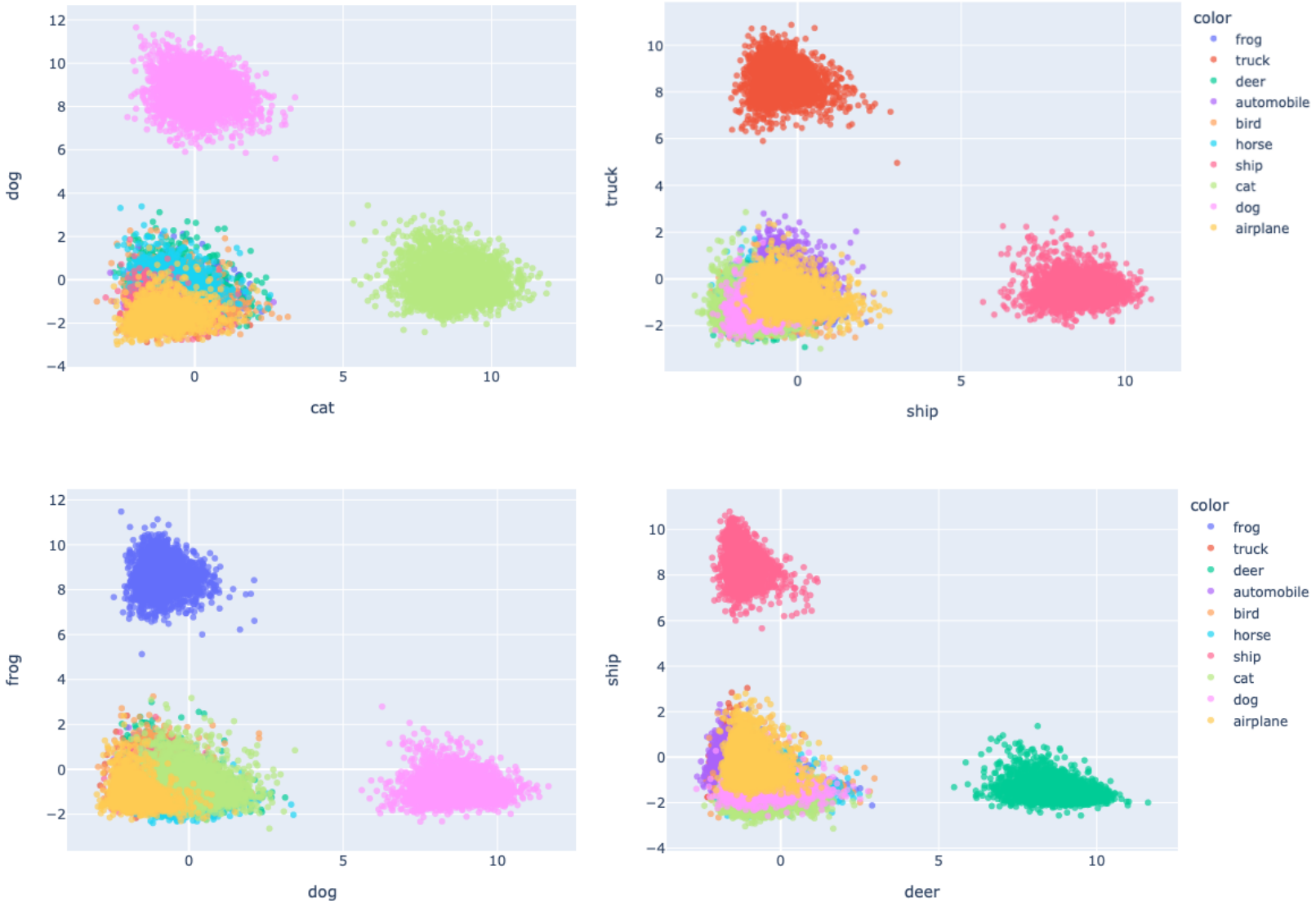


Figure 18: Distribution of elements in the training set in the final checkpoint for some example classes.

4 Concluding Remarks

In this thesis, we have peeked into the black box of a neural network. We investigated the performance and limitations of a decoder mirroring the structure of ResNet-18 using the mean squared error (MSE) loss function. We found that while the decoder performed very well on checkpoint data from early on in ResNet-18, it deteriorated about halfway through the network.

By letting ResNet-18 classify the reconstructed images and examining its accuracy, we got further validation of these results. We conclude that despite its performance on reconstructions based on early checkpoints, the current setup of the decoder needs to be generalized to recreate and preserve the features necessary for correct classification of the images. Still, using ResNet-18's performance on the reconstructed images as validation of the decoder provides a promising quantitative measure of how well the decoder is able to reconstruct relevant features, in contrast to simple visual inspection of the reconstructed images.

The images we generated based on artificial data further highlight the loss of information due to both the low dimensionality of the final checkpoint and the architecture of our decoder, additionally emphasizing the challenge of preserving features relevant for classification. However, the artificial images still provide us with a glimpse of how a typical member of each class might be viewed by ResNet-18.

In comparison to the study by Mahendran & Vedaldi [1], we see a similar visual pattern, as the reconstruction they produced also get blurrier as one moves toward the output layer of the network. As we have mentioned in the introduction, the methods used in this thesis differ from those of Mahendran & Vedaldi [1]. In our study, each abstract representation results in only one reconstructed image, which potentially simplifies interpretation, but could at the same time be misleading as the reconstructions depend on the parameter values of the decoder and are not exact inverses of the transformations applied to the images.

One of the main strengths of the study in this thesis lies in how we measure the decoder's performance. By using ResNet-18 on the reconstructed images to validate the decoders, we are able to accurately judge if the checkpoints and their reconstructions preserve relevant features in the images.

Looking ahead, future studies could incorporate various regularization methods to our loss function for the decoders, making sure that it takes into account ResNet-18's classification of the reconstructions. Letting $L(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)(r)})$ represent the cross-entropy loss as measured between the label of observation i and ResNet-18's classification of the reconstructed image, one could use

$$\text{MSE}(Y^{(i)}, \hat{Y}^{(i)}) + c \cdot L(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)(r)}). \tag{14}$$

Here, $Y^{(i)}$ is the original image, $\hat{Y}^{(i)}$ is the reconstructed image and c is some constant. Furthermore, to ensure that reconstructed images look natural to humans we may introduce a term measuring the norm of the vectorized reconstructed image, as done by Mahendran & Vedaldi [1]. Such a term could prevent extrapolation during image reconstruction, resulting in more natural-looking images.

In conclusion, the thesis further advances the methods used to understand neural networks and provides insight into the challenges and possibilities that come with doing so. By expanding the methods employed here, it may be possible to further shed light into the black box of neural networks in the future.

References

1. Mahendran, A. & Vedaldi, A. *Understanding Deep Image Representations by Inverting Them* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015).
2. Köhler, H. *Unveiling the inner mechanisms of deep convolutional neural networks through the lens of unsupervised learning* MA thesis (Stockholm University, 2022).
3. Charte, D., Charte, F., García, S., del Jesus, M. J. & Herrera, F. A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines. *Information Fusion* **44,** 78–96. ISSN: 1566-2535. http://dx.doi.org/10.1016/j.inffus.2017.12.007 (Nov. 2018).
4. He, K., Zhang, X., Ren, S. & Sun, J. *Deep Residual Learning for Image Recognition* in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 770–778.
5. Goodfellow, I., Bengio, Y. & Courville, A. *Deep Learning* http://www.deeplearningbook.org (MIT Press, 2016).
6. Fukushima, K. Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements. *IEEE Transactions on Systems Science and Cybernetics* **5,** 322–333. ISSN: 0536-1567. http://dx.doi.org/10.1109/TSSC.1969.300225 (1969).
7. Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. *Dive into Deep Learning* https://D2L.ai (Cambridge University Press, 2023).
8. Kingma, D. P. & Ba, J. *Adam: A Method for Stochastic Optimization* 2017. arXiv: 1412.6980 [cs.LG].
9. Ioffe, S. & Szegedy, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* in *Proceedings of the 32nd International Conference on Machine Learning* (eds Bach, F. & Blei, D.) **37** (PMLR, Lille, France, July 2015), 448–456. https://proceedings.mlr.press/v37/ioffe15.html.
10. Santurkar, S., Tsipras, D., Ilyas, A. & Madry, A. *How Does Batch Normalization Help Optimization?* in *Advances in Neural Information Processing Systems* (eds Bengio, S. *et al.*) **31** (Curran Associates, Inc., 2018). https://proceedings.neurips.cc/paper_files/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf.
11. PyTorch. *AvgPool2d* Accessed: 2024-04-10. https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html.
12. Dumoulin, V. & Visin, F. A guide to convolution arithmetic for deep learning (Mar. 2016).
13. Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. *University of Toronto* (May 2012).

14. Paszke, A. *et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library* 2019. arXiv: 1912.01703 [cs.LG].