Titre: Title:	Learned Image Compression for Machine Visual Perception
Auteur: Author:	Jean-Gabriel Simard
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Simard, J.-G. (2021). Learned Image Compression for Machine Visual Perception [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/6579/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Learned Image Compression for Machine Visual Perception

JEAN-GABRIEL SIMARD
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Mai 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Learned Image Compression for Machine Visual Perception

présenté par **Jean-Gabriel SIMARD**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Giovanni BELTRAME, président
Christopher J. PAL, membre et directeur de recherche
Liam PAULL, membre

DEDICATION

To Michel and Antoine

ACKNOWLEDGEMENTS

Many thanks to Felipe Codevila for his precious help.

RÉSUMÉ

Dans cette thèse, nous explorons la compression d'image avec des techniques d'apprentissage profond. Nous proposons une méthode simple pour apprendre une représentation compressible d'image naturelle qui est structurée de façon à faciliter les tâches de vison par ordinateur que sont la classification, la détection, la segmentation sémantique et la reconstruction d'image. Nous testons l'utilisation d'un espace de représentations compressible comme seule donnée disponible pour accomplir ces différentes tâches. Nous procédons aux mêmes expériences à plusieurs taux de compression pour constater l'effet de la compressibilité sur la performance dans les différentes tâches.

Nous désirons obtenir des performances de compression supérieures au codec de compression d'image JPEG tout en obtenant des performances supérieures ou égales dans les différentes tâches quand la représentation compressible est utilisée comme seule source d'information pour les différentes tâches.

Nous présentons des résultats quantitatifs intéressants. En effet, nous démontrons que dans un contexte où peu de données d'apprentissage sont disponibles ou dans un contexte à forte compressibilité, la représentation compressible permet d'obtenir des résultats meilleurs que ceux obtenus en utilisant des images JPEG. De plus, nous publions également un répertoire sur internet pour permettre à d'autres personnes de continuer la recherche en se basant sur le travail déjà accompli.

ABSTRACT

In this thesis, we explore image compression using techniques from deep learning. We propose a simple method to learn a compressible representation of natural images that is structured so as to facilitate the following computer vision tasks : classification, objection detection, semantic segmentation and image reconstruction. We test the use of a compressible representation as the only input to those tasks. We perform these tasks at different compression ratios to explore the effect of compressibility on performance in the different tasks.

We want to obtain a compression that is better than that of the JPEG image codec in the image reconstruction / compression ratio tradeoff. Also, we want to obtain similar or better then JPEG performance when only the compressible representation is used in the vision tasks other than image reconstruction.

We propose interesting quantitative results. We demonstrate that in a low-data regime or in a highly compressed regime, the compressible representation obtains better performance than what is obtainable using JPEG images. Also, we make available on the internet our code so that other people can further explore this area without the engineering difficulty.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

MLP	Multi Layer Perceptron
CNN	Convolutional Neural Network
Conv	Convolution Layer
TConv	Transposed Convolution Layer
VAE	Variational Auto-Encoder
Resnet	Neural Network architecture
SGD	Stochastic Gradient Descent
GDN	Generalized Divisive Normalization
RPN	Region Proposal Network
ROI	Region of Interest
FPN	Feature Pyramid Network
R-CNN	Regions with CNN features
IoU	Intersection over Union
AP	Average Precision
ASSP	Atrous Spation Pyramid Pooling
CVPR	Conference on Computer Vision and Pattern Recognition
MSE	mean-squared-error
KL	Kullback Leibler
ELBO	Evidence Lower Bound
EM	Expectation Maximization
RGB	Red Blue Green
CLIC	Challenge on Learned Image Compression
CODEC	COmpressor-DECompressor program
bpp	bit-per-pixel
PSNR	Peak Signal to Noise Ratio
DCT	Discrete Cosine Transform
FFT	Fast Fourier Transform
EOF	End of File
JPEG	Joint Photographic Experts Group, a group of ISO/IEC
JPEG	a standard published by ISO/IEC
CPU	Central Processing unit
GPU	Graphical Processing Unit
TPU	Tensor Processing Unit

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

The field of machine learning has been generating a lot of interest in the past few years, particularly the subfield of deep learning with its neural networks. The first bricks of theoretical knowledge about neural networks were placed in the sixties with the development of the precursor [8] of the backpropagation algorithm [9] leading to over 60 years of development by now. Only recently, they have started to generate broad interest because of above human performance in many tasks, primarily in computer vision and natural language processing. The main cause is two-factored: the explosive growth of freely available data and available computing power specialized in parallel computation. These recent successes have led deep learning research into a multitude of directions, namely into domains that used extensive heuristics and hard-coded transformations to solve problems that can now be better approached with neural networks and other machine learning algorithms. One such domains is image compression. It is a pillar of the modern digital age because it - with the closely related video compression technology - allows multimedia content to be small enough to be streamed efficiently across the web. Nowadays, multimedia content represents the lion share of global internet traffic. The interest for learned image compression is growing quickly. For example, the Conference on Computer Vision and Pattern Recognition (CVPR), the largest computer vision conference in the world, now organizes a yearly workshop called Challenge on Learned Image Compression (CLIC) to showcase the advances in the field. Teams are invited to compete in producing the best image quality for a given compression ratio. In this thesis, we explore the usage of neural networks in image compression to improve vision tasks.

1.1 Basic concepts

Neural networks are a class of learning algorithms that are loosely modelled on the processes that occur inside the human brain. Humans perceive the world as very abstract concepts comprised of less abstract concepts. For example, we perceive a car as a unique entity, which is itself made of smaller entities such as doors, tires, etc. Trying to emulate this mechanism is the central notion of deep learning and it is called *representation learning*. It describes algorithms that extract a useful set of features, a representation, of an input that can be useful for a given task, classification to name one. These representations are hierarchical, meaning that more abstract representations are extracted from lower-level representations. This abstraction is obtained by stacking multiple non-linear units of computation, called layers, and feeding them the input. Nowadays, the number of layers can be several thousand,

hence the name *deep learning*.

For example, in the case of image classification, the input data of a neural network is a matrix of 8 bit numbers representing pixel values. The first layer of the model might extract low-level features such as lines or corners, but a deeper layer might extract faces or hands, and the last layer the class of the object in the image. Given enough time, data and training resources, the layers will learn to look only at what is truly important to solve the task and ignore everything else, such as noise.

In conventional machine learning, there is no learned representation learning. The developed algorithm takes a set of features and try to complete a given task directly. The features are not learned and are often extracted by highly complex hand-crafted methods. The problem with this feature extraction method is that there are no guarantees that the resulting features will help the model solve the task. For example, in object detection, a number of techniques were developed to extract a set of features that could be recognizable. The Histogram of Gradient features were used successfully used in pedestrian detection. Haar features were used for face detection. Many more were developed for domain-specific applications. If a new unfamiliar object was needed to be detected, there was the associated need to find a good set of features to detect it. This is in no way scalable. Deep learning solves this problem by learning the features that will be used by a classical machine learning algorithm so there is no need to spend time hand-crafting a custom feature detector. If we probed the features learned by a neural network, we could find features that a human could have come up with, but also features the we could not have not come up with. This, in part, explains the dominance of deep learning over classical machine learning in almost every applications.

Conventional image compression is based on a few simple principles. The first one is the Fourier transform. It is an operation from mathematical analysis that can represent any signal into its constituting basic frequencies. This can be easily understood in the one-dimensional case. For example, a song is just a one-dimensional signal of air pressure in the ear. At any given moment of the song, it can be decomposed into its different constituting pitches, such as a low-frequency note of a bass and a high pitch coming from a singer. The same thing can be done with images, but the transformation is in two dimensions. The basic frequencies we obtain are two-dimensional pixel intensity frequencies across the whole image. High frequencies are used to express fine details in the images. Modern image compression schemes, such as JPEG, use the Discrete Cosine Transform (DCT), a variant on the Fourier transform which is well suited for image compression. The high compression ratios are obtained by removing high frequencies in the images. Another approach is entropy coding. It is a class of algorithms that can compress an input to a short bitstream for which the length

is very close to a theoretical limit, if a good probabilistic model of the model is available. Entropy coding is not used on images directly, is it used on a filtered view of representation given by the DCT.

Modern image compression research is focused on creating a neural network-based model to replace two elements of the conventional image compression pipeline. The first one is the DCT. It is a linear representation of the input, meaning that it cannot represent the non-linear features in images in the most efficient manner. Neural networks can produce a better representation because they can learn an optimal non-linear representation of the input that captures more image details. The second one is the probabilistic model of the entropy coding algorithms. The ones used in modern image compression schemes are hard-coded and not adaptive, meaning that the same model is used for all images. A well designed neural network could produce an image and pixel specific probabilistic model for each image, rendering the compressed bitstream shorter. Overall, neural networks have the potential to deliver improved performance over modern image compression schemes in both compression ratio and image quality.

1.2 Research objectives

This thesis had three research objectives. The first one is to build a learned compression algorithm that is superior to JPEG given common metrics to evaluate the performance of image compression techniques. It should use neural networks and produce a compressible representation. The second objective is to find out if it is possible to influence this compressible representation to take into account downstream tasks such as image classification, object detection and semantic segmentation. The third objective is to find out if that compressible representation gives an advantage in densely labelled tasks, such as semantic segmentation, and in the few-shot setting. We hypothesise that it does since the compressible representation should contain all image information in a smaller data structure, making it easier to extract useful information for downstream tasks.

1.3 Thesis outline

This thesis first presents a literature review of the concepts in information theory, deep learning and learned compression necessary to understand the subjects discussed further. Then, we present the neural network compression framework that we used to compress natural images. Then, we present how we add semantic structure to the compressible representation our compression framework produce using the computer vision tasks of classification, detection

and semantic segmentation. Then, we present the experimental results we obtained. These include compression results, and the performance on the different computer vision tasks both from images and from the compressible representation. We obtain very good results in both cases. Then we summarize the major difficulties of this research, which made the training of neural networks difficult and unstable. Finally, we conclude, and propose avenues to build on this research.

CHAPTER 2 REVUE DE LITTÉRATURE / LITERATURE REVIEW

2.1 Information Theory

Information theory answers two fundamental questions in communication theory: what is the ultimate data compression, and what is the ultimate transmission rate. In this thesis we are interested in the first question. The next sections will introduce the basic ideas of this theory and the data compression algorithm that will be used.

First, we define some quantities that will be used throughout the text. The following is adapted from the classic information theory book [10]. The central concept of information theory is the measure of the uncertainty associated with random variables. This idea is formalized as the *Entropy* (2.1) of a random variable.

Definition 2.1 (Entropy) *Let X be a random variable taking values in the finite set $\mathcal{X}$. The quantity*

$$I(x) = \log \frac{1}{p(x)}$$

can be interpreted as a quantity of information carried by the occurrence of x. The Entropy is defined as the expected amount of information of the random variable.

$$H(X) = E_p[I(X)] = -E_p[\log p(X)] = -\sum_{x \in \mathcal{X}} p(x) \log p(x)$$

This definition arises from the answer to the question of what is the average length of the shortest description of a random variable X following distribution p. Then, a natural question to ask next is: what is the average length of the shortest description of a random variable X following p assuming it follows distribution q. Meaning that we cannot encode optimally because we make assumption errors. This concept is formalized as the cross entropy.

Definition 2.2 (Cross entropy) *Let p and q be two finite distributions on $\mathcal{X}$. The cross entropy between p and q is defined by*

$$H(p, q) = -E_p[\log q(X)] = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$$

The measure of the inefficiency between an encoding following entropy and another one following the cross-entropy is formalized as the *Kullback Leibler Divergence* (KL), also known as the *Relative Entropy*.

Definition 2.3 (Kullback Leibler Divergence) *Let p and q be two finite distributions on $\mathcal{X}$. The Kullback Leibler Divergence between p and q is defined by*

$$KL\left[p \parallel q\right] = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} = E_p \left[\log \frac{p(X)}{q(X)}\right]$$

The Kullback Leibler Divergence is a measure of similarity between distributions. It is not a full-blown metric because it does not follow the triangle identity Moreover, it is not symmetric.

We now present mutual information which is a measure of the amount of information one random variable contains about another variable. It represents the reduction of uncertainty of one random variable due to knowledge of another.

Definition 2.4 (Mutual information) *Let X, Y be two random variables of a joint distribution $p_{X,Y}(x,y) = P(X = x, Y = y)$ with marginal distributions $p_X(x) = \sum_y p_{X,Y}(x,y)$ and $p_Y(y) = \sum_x p_{X,Y}(x,y)$. The mutual information of X and Y is defined by*

$$\begin{aligned} I(X,Y) &= \sum_{x,y} p_{X,Y}(x,y) \log \frac{p_{X,Y}(x,y)}{p_X(x)\, p_Y(y)} \\ &= KL\left[p_{X,Y} \parallel p_X p_Y\right] \\ &= H(X) - H(X|Y) \end{aligned}$$

We also know that if the mutual information between two random variables is null then the two variables are independents ($I(X,Y) = 0 \Leftrightarrow X \perp\!\!\!\perp Y$). This is easy to show using the fact that a KL divergence equal to zero happens only if the joint distribution of the two random variables is the product of the marginals, which is the definition of independence ($D(p_{X,Y} \parallel p_X\, p_Y) = 0 \Rightarrow p_{X,Y}(x,y) = p_X(x) p_Y(y)$).

Variational inference

A common saying in statistics is "All Models Are Wrong, but Some Are Useful" [11], meaning that all statistical models, for example neural networks, are approximations of reality but they can make interesting inference from data. Variational inference is a general technique to model complex distributions composed of observed random variables x and latent random variables z. The basic idea is to impose a family of distributions $\{q_\phi\}$, with parameters ϕ, over the latent variables as a modelling assumption. When a model has latent variables, it is impossible to compute the Kullback Leibler Divergence between the current variational model $q_\phi(z)$ and the real distribution of the latent variables $p(z)$ directly. To circumvent this

problem, it is possible to minimize an auxiliary function that is equal to the KL divergence up to constant. This function is presented next.

Definition 2.5 (Evidence Lower Bound) *The Evidence Lower BOund (ELBO) $\mathcal{L}(x)$ is defined as*

$$\mathcal{L}(\phi, x) = \mathbb{E}_{q_\phi}[\log p_{X,Z}(x, z)] - H(q_\phi) \tag{2.1}$$

It can be shown (A.1) to be a lower bound of the likelihood of the observed variable, which means that $p_X(x) \geq \mathcal{L}(x)$ for any variational parameter ϕ. It can also be reformulated (A.1) as

$$\mathcal{L}(\phi, x) = \log p_X(x) - KL\left[p_{Z|X} \parallel q_\phi\right] \tag{2.2}$$

This formulation means that the difference between the ELBO and the true data distribution is the KL divergence between the learned density function and the true posterior.

Maximization of the ELBO is a widely used technique in machine learning. For example, it is the core of the EM algorithm [12] used to find the maximum likelihood estimate of latent variable models such as the Gaussian Mixture Model.

2.2 Deep Learning

Deep learning is the name given the modern use of neural networks. A neural network is a statistical model made of layers and activation function. A layer is a group of simple numerical operations and their corresponding parameters, and an activation function is an element-wise non-linear function. These non-linear activation functions enable neural network models to adapt to non-linear situations, which are the most common. A deep learning model can be viewed as just a complicated function parametrized by its set of parameters θ. It is called a deep model because neural networks usually have many successive layers where each layer's output is the input of the next one and it is called a learning model because θ is automatically learned from data using the back-propagation algorithm instead of being predetermined. The output of a layer is a representation of its input because it is the same information but transformed by a function. A key to understand deep learning is that each consecutive representation is a more abstract version of the input. This means that the deeper a network is the more abstract it can represent the input. A good example is image classification, where a network must abstract the class of the object in an image from raw pixels values.

Most deep learning problems are framed as optimization problems. Usually the minimization of a scalar L named the loss that measures the sum of the predictions errors using a loss

function l. The loss is evaluated over a dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^n$, which is a collection of n pairs of input x_i and their corresponding label y_i. The optimization problem is formulated in this way.

$$\arg\min_\theta L(\theta, \mathcal{D}) = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} l(f_\theta(x_i), y_i) \tag{2.3}$$

The question is how to solve this minimization problem. Deep learning is based on the concept of gradient descent. In successive steps, the weights of the model are updated in the direction inverse to the gradient of the loss function. The gradient is the direction in parameter space where the loss function is maximized, so taking the inverse direction minimizes it if the loss function is linearly approximated. The loss is supposed to be computed over the full dataset, but usually the dataset it too large, making calculation impractical. The solution is to minimize the expected loss. The expected loss can be estimated using a few gouped samples called batches. This is called the stochastic gradient descent algorithm. The general formula is

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \, \mathbb{E}_{X,Y \sim P_\mathcal{D}}[L(\theta, (X, Y))] \approx \theta_t - \alpha \frac{1}{b} \sum_{i=1}^{b} \nabla_\theta l(f_\theta(x_i), y_i) \tag{2.4}$$

In the previous equation, α is the learning rate. It controls the size of the steps the gradient descent algorithm will take. A small learning rate can cause the algorithm to converge slowly and a large learning rate can cause the algorithm to diverge. Finding the best learning rate is a difficult task, and can be solved by trial and error, or with a more systematic approach such as a grid search, or by meta-learning, presented in section 2.2.9. Modern deep learning can use more sophisticated algorithms than stochastic gradient descent such as Adam [13], but they are based on the same principle of gradient descent. The next question is how to compute the gradient in an efficient way. This is where the backpropagation algorithm, presented next, is used.

2.2.1 Backpropagation in a Computational Graph

We introduce the backpropagation algorithm in a computational graph with a basic example to show its importance in deep learning. Following this section, we will introduce some of the more commonly used deep neural networks that will be useful for the thesis such as multi-layer perceptrons, convolutional neural networks, autoencoders and variational autoencoders. The details behind backpropagation and computational graphs have been abstracted away in modern frameworks such as Pytorch [14] and Tensorflow [15]. The goal of this approach is to give the reader an appreciation for what is happening behind the scene when one is using

neural networks.

One of the simplest models f_θ to approximate a function is linear regression. It can be trained in the same way as neural networks, so we can use it as an example to understand the backbone of deep learning. We will present the single variable case for easier reading.

This model takes an input feature x and tries to predict the true label y by a linear approximation $\hat{y}$. The parameters of this model are the weight w and the bias b, so $\theta = \{w, b\}$. The corresponding functional code is the following:

```
def linear_model(x, w, b):
    a = x * w
    return a + b
```

From that code we can extract a computational graph that is presented in figure 2.1, where blue nodes are inputs, rectangular nodes are operations and the red node is the output. The arrows indicate the direction of computation. As a convention, in this thesis, the forward computation goes from the bottom to the top. This is used because higher nodes produce higher representation of the input. This choice will be useful for the backpropagation algorithm presented later.

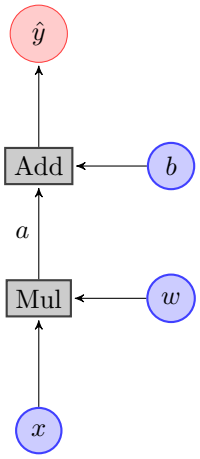


Figure 2.1 Linear Regression Model Computational Graph

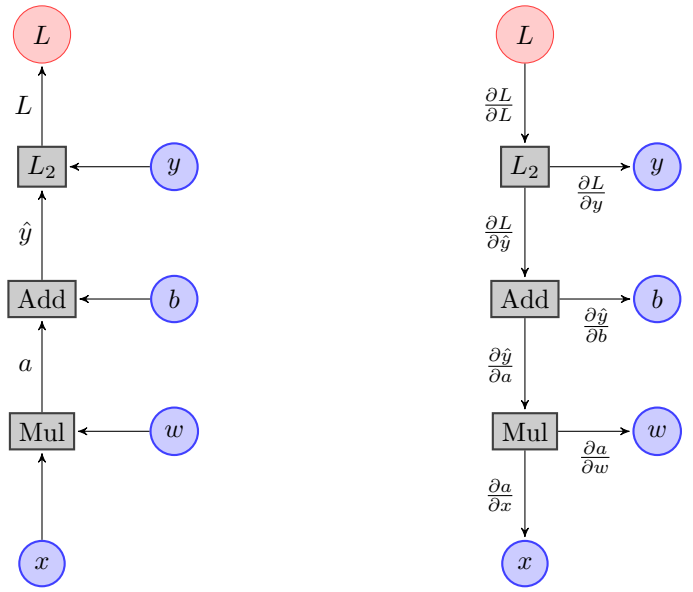
For linear regression, the loss function is quadratic ($l : x, y \to (x - y)^2$). The optimization problem is formulated in equation 2.5 and the computational graph corresponding to this minimization problem is presented in figure 2.2a.

$$\arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} (f_\theta(x_i) - y_i)^2 \tag{2.5}$$

The gradient of the loss function, with respect to parameters of the model, is the vector of all the partial derivatives of the loss with respect to each parameter. The solution is to use the chain rule of calculus as presented in equation 2.6.

$$\left.\frac{\partial z}{\partial x}\right|_{x=x'} = \left.\frac{\partial z}{\partial y}\right|_{y=y(x')} \left.\frac{\partial y}{\partial x}\right|_{x=x'} \tag{2.6}$$

Using the chain rule, it is very easy to compute all the necessary partial derivatives. In our case of linear regression, the first partial derivative needed is $\frac{\partial L}{\partial b}$, and it can be decomposed in two parts: $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b}$. The second partial derivative that is needed is $\frac{\partial L}{\partial w}$. Again using the chain rule we can decompose it in the following way: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial w}$. The computation of all necessary partial derivatives can be easily handled by the computational graph. In a step known as the backward propagation, the flow of computation is inverted. It then becomes a tree where the loss is the root, leaves are either inputs or parameters and the other nodes are operations. This step is shown in figure 2.2b.



(a) Forward Propagation (b) Backward propagation

Figure 2.2 Backpropagation in the Linear Regression Model Computational Graph

In the backward propagation, the partial derivatives of the loss with respect to a given parameter can be obtained by following the path of computation from the loss to the leaf of that parameter and multiply every intermediate value. Intermediate computations needed in multiple partial derivatives, such as $\frac{\partial L}{\partial \hat{y}}$ which is present in both $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$, are only computed once, because each arrow is only used once. All the partial derivatives that need

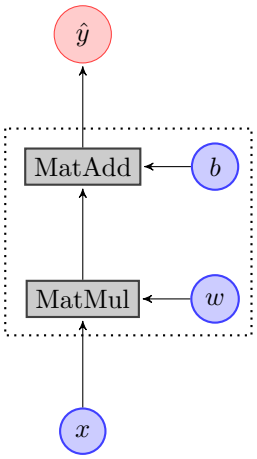
to be computed are the derivative of the operation nodes. This means that to add new operation nodes to a computational graph, one only needs to implement two functions: the forward function and the backward function.

To use the stochastic gradient descent algorithm, the same computation must be performed for multiple samples. The computations are all independent before they are averaged. The trick that all deep learning frameworks use it to compute these independent computation in parallel on specialized devices made for these tasks, such as GPUs or TPUs. For the computation to be done in parallel, a given layer will inject multiple times the same type of input in a data structure called a tensor. Tensors are a generalization of matrices to any number of dimensions. Indeed, a matrix is a two-dimensional tensor. For our problem of linear regression, each input is of dimension 1 and if we want to pack B of them in a mini-batch we put them in a tensor of shape $(B, 1)$. The parallel computing device will automatically compute the same operations of each element along the batch size dimension.

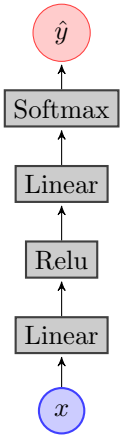
2.2.2 Multi-layer Perceptron

The multi-layer perceptron (MLP) is the simplest type of neural network architecture and it is a generalization of the linear regression model presented in figure 2.1. The first difference is that it operates in multiple dimensions instead of just one. The input is a vector in dimensions m instead of a scalar and the output is a vector in dimensions n, n and m do not have to be equal. The weight scalar becomes a weight matrix of dimensions $n \times m$ and the scalar multiplication a matrix multiplication. The small model is called a linear layer because matrix multiplication it is a linear operator, but it is also called a fully connected layer because each scalar in the output vector of the layer is computed using all the input scalars of the input vector. The computational graph of that layer is enclosed in dotted lines in figure 2.3a. The second difference is that the output of a linear layer is passed into an activation function that is non-linear such as sigmoid, softmax or relu. The last difference is that multiple layers can be stacked on each other in the order: linear, activation, linear, activation... and so on.

The input tensor of a linear layer is of shape (B, m) and the output shape (B, n), where B is the batch size. If the network has multiple layers on top of each other, the intermediate layers between the input layer and the output layer are called hidden layers. The computation graph of an MLP with one hidden layer is shown in figure 2.3b, where the layer's computational graphs are grouped in one operation. The width of a layer is the number of outputs of that layer. It has been shown [16] that a MLP with one hidden layer and sufficient width can approximate any function. The number of parameters in a linear layer is $n \times (m + 1)$ because



(a) Linear Layer



(b) MLP with 1 hidden layer

Figure 2.3 Multilayer perceptron

the weight matrix contains $n \times m$ parameters and the bias vector n. This means that the number of parameters grows very fast in an MLP.

Nowadays, MLPs are put at the end of neural networks and are used to output the desired output distribution. If the output must be a probability distribution, the last layer of an MLP is usually a softmax function because it guarantees that the output vector for one sample will be only positive elements that sum to one.

2.2.3 Convolutional Neural Network

The Convolutional Neural Network (CNN) was developed specifically to deal with images and is based on the discrete convolution operation. A discrete convolution is a linear transformation that preserves the notion of order. The operation takes two matrices inputs: a feature map and a kernel. The kernel slides across the feature map and at each location the dot product is computed. The result is put at the current location in the output feature map. The dot product between two vectors is the sum of the product of elements at the same index in both vectors. This operation is shown in figure 2.4, where the blue matrix is the kernel and the green square is the output of the computation taking place at the red squares in the feature map. In traditional image processing, a number of operations on images can be done using a discrete convolution with a carefully crafted kernel. Indeed, edge detection, sharpening, blurring and a multitude of other operations on images can be computed by the discrete convolution operation. The advantage of CNN is that the kernel is learned.

The formula for this operation for the 2 dimensional case is given in equation 2.7 where (i, j)

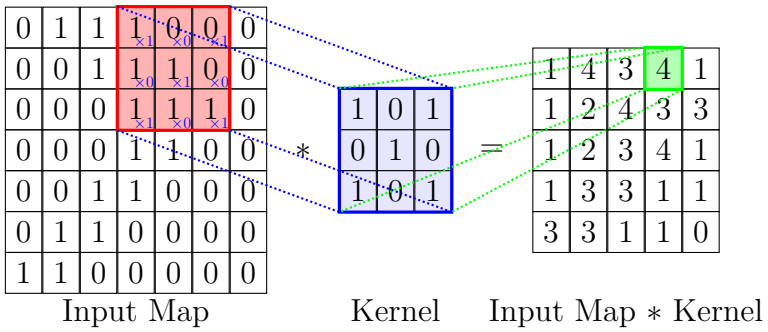


Figure 2.4 Convolution of a 1-Channel Feature Map

is the position in the output feature map x, f is the input kernel, k is the kernel and m is the width of the kernel.

$$x_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} k_{ab} f_{(i+a)(j+b)} \tag{2.7}$$

In a convolution neural network, there can be more than one input feature map and there can be more than one kernel to learn. The feature maps are organized as 4 dimensional tensor (B, C_{in}, H, W), where B is the batch size, C_{in} represents the number of channels, H the height and W the width of the feature map. The kernels are organized in one tensor with dimensions $(C_{in}, C_{out}, K_H, K_W)$ where C_{in} is the same at the one in the feature map, C_{out} the number of channels of the output features map, K_H the height of all the kernels and K_W the width of all the kernels. l Next, we present a concrete example using the MNIST dataset, which is a collection of grayscale images of handwritten digits. The input feature map would be of dimensions $(1, 28, 28)$ for a total of 728 inputs features. If we choose the image to be convolved with 64 kernels of dimensions 3×3, there is a total $64 \times 3 \times = 576$ parameters. If we choose a stride of 2, the output dimension of the convolution operation will be of dimensions $(64 \times 14 \times 14)$ for a total of 12544 output features. In an MLP, the number of parameters to obtain the same amount of output features would be $728 \times 12544 = 9,132,032$. The number of parameters quickly becomes unmanageable.

A major difference between these two architectures is the *translational invariance* of the CNN. A convolution layer can be seen as a feature detector tuned for a given feature map. This means that a relevant feature to detect could be at any position in a feature map and still be detected. This comes from the fact that the same kernel is applied at every point in the image. An MLP would have different weights for each position, so the same signal at a different position in the feature map would be interpreted very differently.

The CNN layer is responsible for the resurgence of neural networks in artificial intelligence

because of its recent results in image-related tasks. Indeed, in 2014, the AlexNet architecture [17], based on convolutional layers, competed in the *ImageNet Large-Scale Visual Recognition Challenge* [18], a classification challenge, and achieved a top-5 error of 15.3%, which was 10.8 points lower than the runner-up model of that year. Since then, all top-performing architectures at this annual competition use neural networks and the convolutional layer. Since this first result, many models improved on the metric. Recently, the best result is a top-5 error of 1.3 % [19], outperforming even the human performance of 5.1 % top-5 error [18].

As in most deep learning, successive layers learn a more abstract representation. We can see this in figure 2.5 (reproduced from [20]) in which the very first convolutional layer learns simple edge detectors and the last layer computes features akin to entire face detectors.

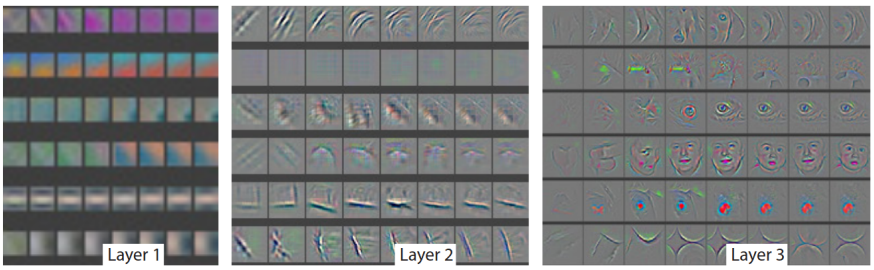


Figure 2.5 Filter visualizations of a convolutional neural network

2.2.4 Resnest

A very common neural network architecture used for modern computer vision research is the Resnet architecture [21]. It was developed with the objective of training very deep neural networks, where a deeper network would always outperform a shallower one. The problem was that the magnitude of the partial derivatives for the first layers of very deep networks was very low. This would lead to very low learning of the first layers and poor performance with deep networks. The solution they proposed is the residual block. It can be summarized in the function $B(x) = x + F(X)$, where B is the block function and $F(x)$ is a non-linear representation of x. The addition of the identity function is called a *skip connection* and it allows a better flow of gradients from the last layers to the first layers of the network during the backpropagation algorithm. The original work also proposed a recipe that can be used to construct neural networks with an arbitrary number of layers.

The Resnet architecture is always composed of 5 groups of layers named the stem and layers 1 to 4. The stem is a module that takes raw RGB images and down samples them by a factor of four. The layers 1 to 4 repeat the same basic block, and what changes between

the different Resnet architecture is the number of repetitions. This architecture is presented in the second column of the figure **??**. In this figure, the bracket represents the non-linear function in a residual block. Each line represents a convolutional layer followed by a Relu activation function. The numbers represent the size of the convolutional kernel, followed by the amount of output channels.

We use this architecture extensively in this work.

2.2.5 Autoencoder

The autoencoder creates a lower-dimensional representation of the original input that can be used to reconstruct the original input. The autoencoder is composed of two neural networks: the encoder E_ϕ and the decoder D_θ. The encoder is a learned function $E : \mathbb{R}^m \to \mathbb{R}^n$ where n is lower than m and the decoder is an inverse function $E : \mathbb{R}^n \to \mathbb{R}^m$. The optimization problem associated with this architecture is

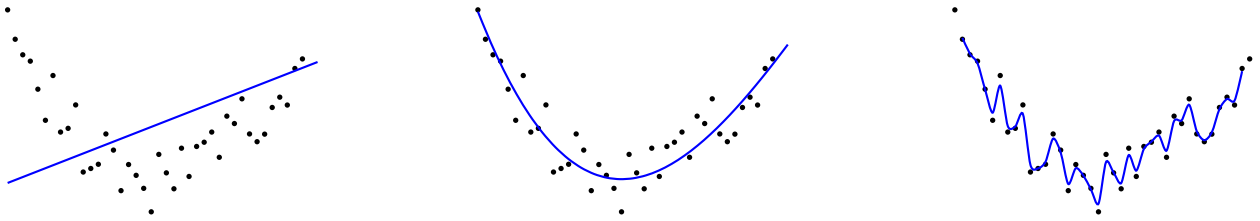
$$\arg\min_{\phi,\theta} \frac{1}{n} \sum_{i=1}^{n} l(D_\theta(E_\phi(x_i)), x_i) \tag{2.8}$$

Where l is usually the Frobenius norm of the difference. The autoencoder can be seen as a hidden variable model, where the lower-dimensional representation $z = E_\phi(x)$ is the hidden variable. It can be used to create a rich feature space on which some task other than reconstruction can be executed. The dimensionality of the latent representation is smaller than the feature space, meaning that the autoencoder can be seen as a lossy compression algorithm, because there is no guarantee that an arbitrary input can be reconstructed without artifacts. The architecture of an encoder and decoder can be composed of any type of layer.

2.2.6 Regularization

Regularization encompasses a multitude of techniques in artificial intelligence to help learning algorithms generalize to new datasets by minimizing overfitting to a particular dataset. An algorithm can overfit, meaning that the training error is low but the testing error is high. It can under fit, meaning that both the training and test error are high. It can also be between those two cases, meaning that both the training and testing error are low. These three cases are presented in figure 2.6, where the learned function in blue tries to approximate the two-dimensional dataset.

There are multiple ways to regularize a deep learning problem. A common way is to add terms to the optimization problem the neural network is trying to solve. This is formulated in



(a) Under (b) Good (c) Over

Figure 2.6 Types of fit

equation 2.9, where R is the regularization term and λ is the weight of that regularization. A bigger weight means that the regularization is more important. In this class of regularization, the most basic approach is to softly penalize the parameters of the network to force them to be numerically stable. This is done because the original optimization problem is ill-defined meaning that there is an infinite number of solutions. The two main regularization techniques in this category are the L1 and L2 regularization. The L1 regularization is defined as $\sum_i \parallel \theta_i \parallel_1$ and can be proven to force the parameters to be sparse, meaning that most elements are zeros. The L2 regularization is defined as $\sum_i \parallel \theta_i \parallel_2$ and can be proved to force the parameters to be close to the zero vector.

$$\arg\min_{\theta} L(\theta, \mathcal{D}) + \lambda R(\theta, \mathcal{D}) \tag{2.9}$$

One of the main bottlenecks in deep leaning is the amount of data on which the training can be done. Data is usually limited and expensive to get. A solution to this problem is to create new data points by modifying existing data points without changing the corresponding semantic meaning. These augmentations are dataset specific. A basic example of this technique in image classification could be as follows: If the dataset is images of cats and dogs and the augmentation is flipping horizontally and vertically, we can see that the flipped images are completely new data points, but the class of the images is the same. If the dataset is MNIST, the same augmentation cannot be used because the image of a six flipped can look like the image of nine. Data augmentation is a type of regularization that does not penalize the parameters directly but that makes the learning algorithm invariant to data noise, decreasing the chances for the learning algorithm to overfit.

2.2.7 Variational autoencoder

The variational autoencoder (VAE) is a particular type of autoencoder with a soft constraint on the latent variable distribution to be part of a parametric distribution via an additional

loss term to the minimization problem of training the neural network. This way, because the latent space is part of known family of distribution, it becomes easy to sample a latent sample with non-negligible probability mass. This means that the VAE model can be used as a generative model to produce new samples of the observable variables that have not been seen before and are realistic. This is a soft version of variational inference presented in section 2.1. Just like in typical variational inference the ELBO is maximized but it is not directly optimizable. However, it can be reformulated, as shown in A.2, in a way that is optimizable and can be used directly to formulate the optimization problem.

$$\arg\max_{\phi,\theta} \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\phi,\theta,x_i) = \arg\min_{\phi,\theta} \frac{1}{n}\sum_{i=1}^{n} -\mathbb{E}_q[\log p_\theta(x|z)] + KL\left[q_\phi(z|x) \parallel p(z)\right] \tag{2.10}$$

In this reformulation, $q_\phi(z|x)$ and $p_\theta(x|z)$ are the encoder and the decoder of an autoencoder. The first term of equation 2.10 is a reconstruction loss, because if $p_\theta(x|z)$ is assumed to be a Gaussian distribution with the input x as mean and an identity covariance matrix then $\log p_\theta(x|z)$, reduces to $\parallel \hat{x} - x \parallel^2$ (A.2) which is the L_2 norm between the input x and the reconstructed input $\hat{x}$.

Reparametrization trick

To compute $\mathbb{E}_q[\log p_\theta(x|z)]$, a latent representation must be sampled using q_ϕ as a density function. This operation is not directly differentiable, it would block the backward propagation in a computational graph like we can see in figure 2.7a. This comes from the fact that taking the gradient of the expectation of a function $f_\theta(z)$ with respect to a density $p_\theta(z)$ when both are parametrized by the same parameter θ is not a differentiable operation. This can be shown by

$$\nabla_\theta \, \mathbb{E}_{p_\theta}[f_\theta(Z)] = \int_z f_\theta(z) \nabla_\theta p_\theta(z) dz + \mathbb{E}_{p_\theta}[\nabla_\theta f_\theta(Z)]$$

The expectation can be obtained by averaging the gradient from multiple samples, but the integral cannot because $f_\theta(z)$ is not a density function. To alleviate this we use the reperametrization trick. In annex A.2 we show how the reparametrization trick is a special case of the Infinitesimal Perturbation Analysis technique, a well-known technique in the simulation community. The idea behind this trick is to put the sampling node out of the flow of computation between the encoder and the decoder in the computational graph to allow the backpropagation algorithm to work. This means that randomness is introduced not by the parameters of the latent distribution, but by another variable. This can be done when there a deterministic mapping $\boldsymbol{z} = g(\psi, \boldsymbol{\epsilon})$, where ϵ is an independent random variable with

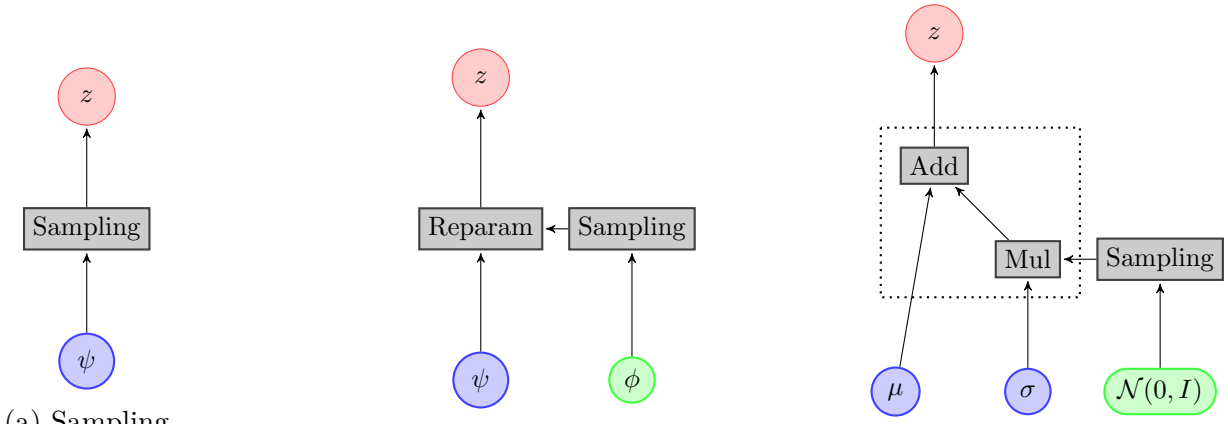
density function $p_\epsilon(\epsilon)$, and ψ are the parameters of the function to sample from produced by the encoder. Then

$$\mathbb{E}_{p_\theta}[f_\theta(Z)] = \mathbb{E}_\epsilon[f_\theta(g(x,\epsilon))] \tag{2.11}$$

We can apply this result to $\mathbb{E}_q[\log p_\theta(x|z)]$ using g as the reparametrization function.

$$\nabla_{\phi,\theta}\, \mathbb{E}_q[\log p_\theta(x|z)] = \mathbb{E}_\epsilon[\nabla_{\phi,\theta} \parallel p_\theta(g(\epsilon, q_\phi(x))) - x \parallel^2]$$

The gradient that will be used is the one coming from the average reconstruction loss. The computational graph for the reparametrization trick in the general case in shown in figure 2.7b. Then the optimization problem of a VAE as presented in equation 2.10 simplifies to the autoencoder optimization problem presented in equation 2.8 with the addition of a regularization term on the latent space.



(a) Sampling

(b) Reparametrization

(c) Reparametrization for a diagonal Gaussian

Figure 2.7 Repametrization Trick

Original formulation

In its original formulation [22], the variational autoencoder used a Gaussian distribution with zero mean and identity covariance matrix as the goal distribution $p(z)$ and forced the encoder to output the parameters of a diagonal Gaussian. This allowed for a simple analytic formula of the KL divergence (A.2) and for a simple reparametrization function which is presented in equation 2.12 with the corresponding computation graph presented in figure 2.7c.

$$z = \mu + \sigma \odot \epsilon \tag{2.12}$$

β-VAE

An extension of the Variational Auto Encoder that will be relevant in this thesis is the β-VAE [23]. The new objective function is obtained by imposing the constraint that the KL divergence must be bounded by a value δ. This is formulated as follows.

$$\arg\max_{\phi,\theta} \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\phi,\theta,x_i) \tag{2.13}$$

$$\text{subject to } KL\left[q_\phi(z|x) \parallel p(z)\right] < \delta \tag{2.14}$$

This constraint optimization problem can be solved by using the Lagrangian method to integrate the constraint into an unconstrained optimization problem. The resulting Lagrangian equation is the following:

$$\mathcal{F}(\phi,\theta,\beta) = \mathbb{E}_q[\log p_\theta(x|z)] - \beta(KL\left[q_\phi(z|x) \parallel p(z)\right] - \delta) \tag{2.15}$$

where β is the Lagrangian multiplier. A lower bound on the Lagrangian can be used at the optimization problem to train a β-VAE where the δ is removed because it is a constant so it wont affect the minimization. The resulting optimization problem that will be used to train the encoder and the decoder networks is

$$\arg\min_{\phi,\theta} \frac{1}{n}\sum_{i=1}^{n} -\mathbb{E}_q[\log p_\theta(x|z)] + \beta KL\left[q_\phi(z|x) \parallel p(z)\right] \tag{2.16}$$

The only difference between this optimization problem and the one presented in equation 2.10 is the term β. If this therm is equal to 1, then it reduces to equation 2.10 of the original VAE problem. The magnitude of β applies a constraint on the representation capacity of the latent representation. A higher β reduces the capacity.

It was suggested that the β-VAE learns a disentangled representation, meaning that each element of the latent representation would be responsible for only a few things in the reconstruction. This can can be understood by looking at figure 2.8, where one element at a time of the latent representation is changed in a VAE and a β-VAE. It is clear that for the β-VAE some latent elements are responsible for specific parameters of the reconstructed images such as the position in the horizontal axis of the white blob. The same behaviour is not observed in the plain VAE, where each used element is responsible for many things at once.

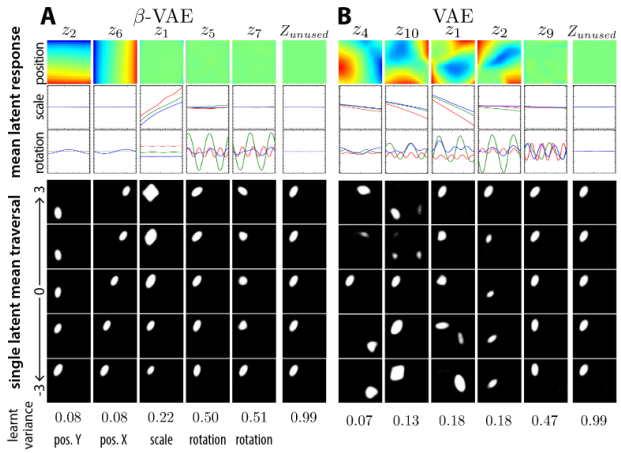


Figure 2.8 Representation of the reconstructed image when one element of the latent representation of a VAE and a β-VAE. Reproduced from [1]

2.2.8 Information Bottleneck

The information bottleneck [24] (IB) method was proposed to give a theoretical background to the extraction of the relevant information in an input variable X to predict an output variable Y. The relevant information is defined as the mutual information $I(X, Y)$ between the two variables. An optimal representation of X would only keep the information necessary to get Y and would disregard the rest. This is noted as $\hat{X}$. It poses the following Markov chain $Y \to X \to \hat{X} \to \hat{Y}$. Finding $\hat{X}$ is found by minimizing the following equation

$$\min I(X, \hat{X}) - \beta I(\hat{X}, Y) \tag{2.17}$$

This a rate-distortion problem from compression, which is very close to what we will use in this work. In fact, this can be interpreted as a balance between the complexity of the representation $I(X, \hat{X})$ and the extracted information $I(\hat{X}, Y)$.

In the context of deep learning, this theory had to updated a bit, because there is not only one intermediate representation between the input variable and the output variable but several given by the different layers. It has been shown that the some information about about Y is lost in every layer [25]. This can be expressed as

$$I(Y, X) \geq I(Y, h_i) \geq I(Y, h_j) \geq I(Y, \hat{Y}) \tag{2.18}$$

where h_j is a layer deeper then the layer h_i in a deep neural network. The IB problem is also

posed for all the intermediate representation of the network and not only the last last one

$$\min I(h_{i-1}, h_i) - \beta I(Y, h_{i-1}|h_i) \tag{2.19}$$

This means, that each layer decreases the complexity of the representation but increases the distortion.

2.2.9 Meta Learning

Meta learning, also known as *learning to learn*, aims to create neural networks that can learn new tasks or adapt to new environments rapidly with only a small amount of data. We present the meta leaning problem, in the special case where the tasks are restricted to be supervised learning tasks. The general view optimization problem is

$$\arg\min_{\theta} \mathbb{E}_{\mathcal{D}\sim p(\mathcal{D})}[L(\theta, \mathcal{D})] \tag{2.20}$$

We note that this problem is very similar to the one presented in equation 2.3, but with addition of an expectation. In this formulation, the dataset is not fixed but is considered a sample. This means that the model must learn a representation that is good with respect to a class of datasets instead of just one.

In this framework, the dataset $\mathcal{D}$ is split in two different subsets, the support set $\mathcal{S}$ and the prediction set $\mathcal{B}$. In the case of few-shot classification, the meta learning task is called a N-way-K-shots classification task, meaning that $\mathcal{S}$ contains K samples for each of the N classes.

There are many approaches to meta-learning, but they are mainly grouped in three categories. First, Metric-based methods learn embeddings that then tend to separate the different classes even the unseen ones. A model that proved influential in that category is the Prototypical networks [2]. The main idea behind this model is to learn a *prototype* feature vector for every class, that is the average embedding of a few samples for a given class. The class of a new sample is given by the prototype closest to the embedding of the new data point. This is very similar to k-nearest neighbour algorithm [26] but in embedding space. The idea is well presented in figure 2.9.

Second, Model-based methods learn models with parameters that can easily be fine-tuned to give good results. Third, Optimization-based methods put constraint on the learning algorithm to choose parameters that generalize well from few examples. One of the most important contribution in this area is Model-Agnostic Meta-Learning (MAML) [3]. It is

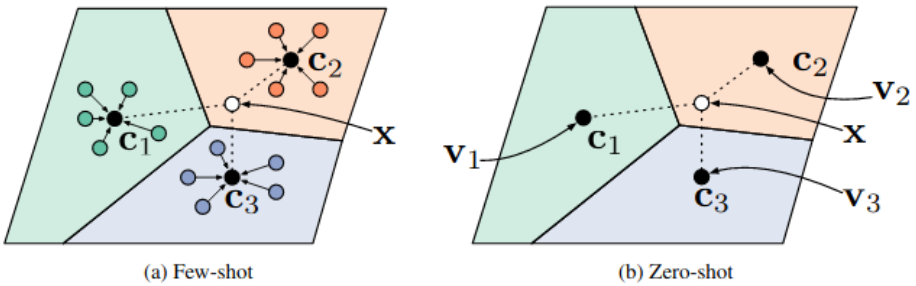


Figure 2.9 Prototypical Networks in the few-shot and zero-shot scenarios. Reproduced from [2]

compatible with any differentiable model, making it wildly applicable in deep learning. The main idea behind this paper is to find θ^*, the optimal parameter of the model f_θ, that will enable efficient task specific fine-tuning. This is obtained by iteratively updating the model parameters using the average of the gradients produced by all the different tasks. This is illustrated well in figure 2.10.

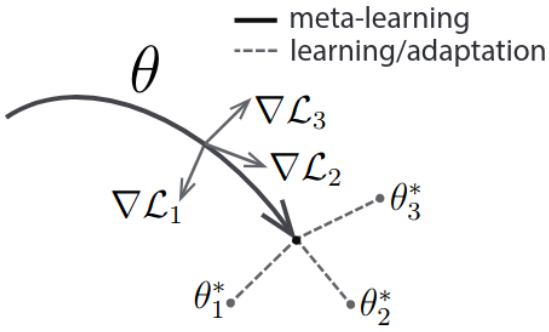


Figure 2.10 Model-Agnostic Meta-Learning, reproduced from [3]

2.3 Compression

The overall goal of data compression is to encode a message to a representation that is shorter than the original message and be able to decode the representation to get back the original message. The process can be lossless, meaning that the reconstruction is identical to the original input. The process can also be lossy, meaning that the reconstruction is different from the input but it is still recognisable. The advantage of lossy compression over lossless compression is that the compressed representation can be much smaller because fine details are ignored.

Image compression is a fundamental technology of the digital age because it is the basis of visual communication. Its development dates back to the 1960s. The first forum on the subject, Picture Coding Symposium was created in 1969. It was devoted exclusively to advancements in image and video compression. Since then, both academia and industry researchers have produced tremendous efforts to improve the technology. Because of the importance of the topic, the International Standardization Orginizaiton (ISO) created an expert group called Joint Photographic Experts Group (JPEG) to produce standard lossy compression formats, namely JPEG [27] and JPEG2000 [28]. We will compare our method to those formats.

A lossy image compression algorithm is evaluated on two aspects: the compression efficacy, measured in bit-per-pixel (bpp), the less the better, and on the image quality, measured with Peak Signal to Noise Ratio (PSNR) in decibels (dB), which is directly obtained from the mean-squared-error (MSE) as shown in this equation.

$$PSNR = 20 \log_{10} \frac{255}{MSE} \tag{2.21}$$

These two measures are competing with each other, because a higher image quality will directly result in lower compression efficacy and vice versa. This problem is called the rate-distortion trade off. To compare the performance of lossy codecs, the psnr/bpp curve of the algorithms are compared. The one with the largest area under the curve is the better algorithm.

In this thesis, we will use both lossy and lossless compression. We use an autoencoder to compress an image to a representation which is a lossy process. Then we use the lossless algorithm presented in the next section to encode the representation obtained at the autoencoder bottleneck.

2.3.1 Lossless Compression: Arithmetic Coding

It can be shown that when encoding a message in a lossless fashion the minimum length of the encoding is given by the entropy of the message. Entropy coding is a class of lossless compression algorithms that can achieve the theoretical limit. They are called entropy models because they use a probabilistic model of the input. The Huffman coding procedure, an instance of entropy coding, is optimal for encoding a random variable with a known distribution that has to be encoded symbol by symbol. A symbol is an input unique charachter, such as a letter. However, because of the constraint that code-word length for a Huffman code must be an integer, there can be a loss of up to 1 bit per symbol in coding efficiency [10]

which can lead to encoded representations that are a few percentage points longer than the optimal length. We could alleviate this loss by using blocks of multiple input symbols, but the complexity of this approach increases exponentially with block length, limiting its usefulness. Instead we will use arithmetic coding, another entropy coding method, to compress without this inefficiency.

Arithmetic coding can be seen as an extension of Huffman coding wherein the whole message is encoded in one block. To encode a message, instead of using a sequence of bits to represent each symbol, arithmetic coding represents the whole message by the binary representation of a number inside a carefully chosen subinterval of the $[0, 1[$ interval. This algorithm can be viewed as Huffman codding but with a unique symbol representing the whole message. Then, the inefficiency caused by the integer number of bits for each block is spread on only one instead of a multitude. From the uncountable property of the natural numbers, it can be proven that each message can be mapped to a unique subinterval of the $[0, 1[$ interval.

The goal of the algorithm is to find the subinterval corresponding to the message during the encoding phase, and inversely to find the message corresponding to the subinterval during the decoding phase. The algorithm works in a recursive fashion. Given we already encoded part of a message we know the subinterval corresponding to the already encoded part of the message. To encode each new symbol, the algorithm adds bits at the end of encoding without modifying previously encoded symbols, by choosing a shorter subinterval inside the current one. A shorter interval is represented with high precision number that has a long decimal expansion. The shorter the interval, the longer the decimal expansion and the longer the encoding.

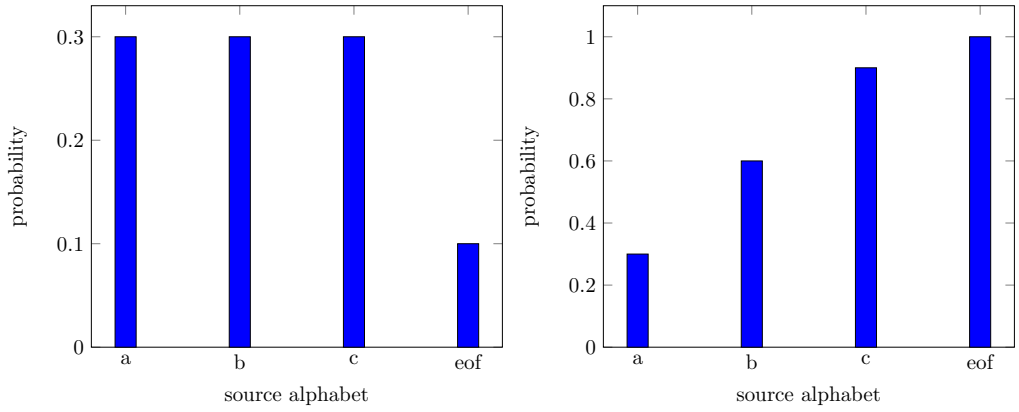
The algorithm needs only three inputs at each step: the current interval, the next symbol to encode, and a model that will predict a probability distribution over the source alphabet from which the next symbol to encode is drawn given all the previous symbols that were already encoded. To produce the new interval, the algorithm divides the current interval into subintervals, each representing a fraction of the current interval proportional to the probability of that symbol. The subinterval that corresponds to the symbol to be encoded becomes the interval used in the next step. Once the final symbol of the message is encoded, an *end-of-file* (EOF) symbol is also encoded to indicate to the decoding algorithm where to stop decoding bits. Otherwise it could continue decoding the subsequent bits in memory adding random decoded bits at the end of the reconstruction. The final encoding could be any number in the final subinterval, but in practice the number in that range with the shortest binary representation is chosen. The decoding step performs the same steps in the same order, because the encoding point will always be in the relevant subinterval at any given

step.

With this description, the algorithm would need infinite precision, which is not achievable on current computers. Instead, the intervals are rescaled to the [0,1[interval at each step of the algorithm and the subinterval fractions are rounded using a given number of bits of precision which allows the use the integer arithmetic in CPU.

It can be shown that if the probabilistic model matches the real distribution of the message to encode, then the length of the encoded representation of the message is at most two bits longer than the entropy of that message [10]. This means that the additional encoding cost is spread across the whole message instead of being present for each code as in Huffman coding. This means that for longer messages, such as images, this additional cost becomes negligible and therefore the encoding is nearly optimal. This thesis uses neural networks to compute this probabilistic model.

As an example, we present the encoding of the sequence $S = bac$, where the source alphabet is $\{a, b, c, EOF\}$ and EOF is used to signify the end of the message. We make a few assumptions to make the example simple to follow. First, we make the probabilistic model memory less. This means that each time a new symbol needs to be encoded, the same distribution over the source alphabet will be used without taking into account the symbols already encoded to produce a better distribution. Second, we choose the probabilistic model to be the distribution presented in figure 2.11. We choose this distribution because it is uniform over the elements in the alphabet used in the message and we put a small probability mass for the EOF symbol.



(a) Probability mass function (b) Cumulative function

Figure 2.11 Probability function of over the source alphabet that will be used for each symbol to encode

At the start of the algorithm, the available interval is [0,1[. Then we need to encode the

symbol b, so we partition the current interval in multiple subintervals with lengths that are proportional to the probability masses presented in figure 2.11a. We can see this partition on the left of figure 2.12. The subinterval that is chosen is [0.3, 0.6[, this is indicated by the red bar. Then we repeat the same procedure but using the subinterval as the one to be partitioned, this is shown by the selection of the [0.3, 0.39[interval because it is the one associated with the symbol a at the second step. We then repeat the same operation until we encode the symbol *EOF*. At this point, the subinterval is [0.381, 0.3783[. The number that will represent the encoding could be any number in that interval. We choose the number 0.37890625 because it is the one that has the shortest binary representation with 0.01100001. This number is represented by black dots in figure 2.12. The encoding is then the binary string 01100001, which is the decimal digits of our chosen number. We note that this binary string has a length of 8, it is only 0.42 bit bigger than the entropy of the sequence at 7.58. It represents a 5.5% inefficiency. If the message was much longer, for example a whole book or an image, then the inefficiency would rapidly approach 0 %.

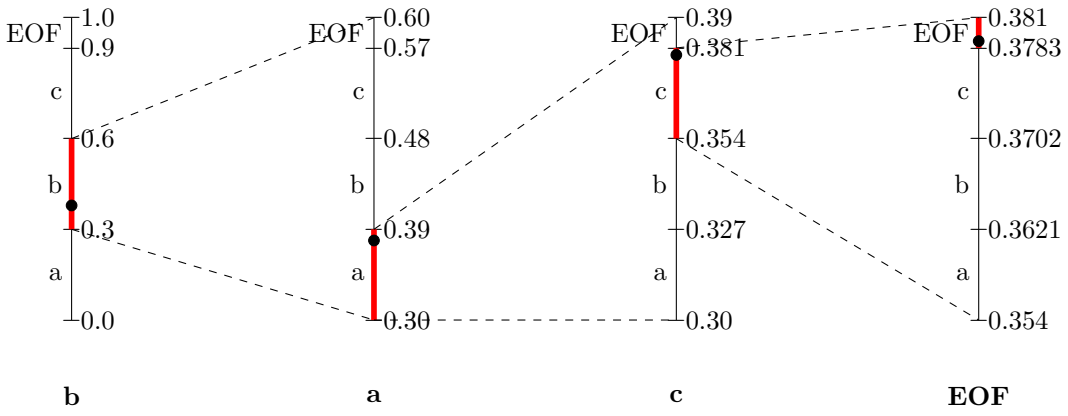


Figure 2.12 Application of the Arithmetic Coding algorithm to the *bac* sequence given the memory-less probabilistic model presented in figure 2.11. The red bars show the current subinterval and the black dots represent the position of the number chosen as the encoding representation. The encoding is done from left to right.

To decode the binary string back into the original message, we apply a procedure similar to the one used for the encoding. We repeat the same procedure of interval partition, and the number chosen to be the representation will always fall in the subinterval corresponding to the next symbol to decode. When the number falls into a subinterval corresponding to the symbol *EOF*, the algorithm finishes because the original message has been recovered without loss of information.

2.3.2 Learned Lossy Compression using Neural Networks

Lossy compression is usually performed using the Transform coding [29] paradigm. In this pipeline, the input image is transformed using an encoder module and then quantized. The quantized representation can then be encoded to a bitstream using an entropy coding algorithm. This is formulated by the equation 2.22, where $\mathcal{E}$ is the encoder module, θ_E the parameters of the encoder and $\mathcal{Q}$ the quantization operation.

$$z = \mathcal{Q}(\mathcal{E}(x; \theta_{\mathcal{E}})) \tag{2.22}$$

To obtain the original image a decoder $\mathcal{D}$ is used as shown in 2.23. The resulting image $\hat{x}$ is an approximation of x.

$$\hat{x} = \mathcal{D}(z) = \mathcal{D}(\mathcal{Q}(\mathcal{E}(x; \theta_{\mathcal{D}})), \theta_{\mathcal{D}}) \tag{2.23}$$

As an example, JPEG [27] uses this pipeline. It is used because the image space is often not adapted to compression because it is difficult to come up with a good probabilistic model over it.

The main difference between traditional and neural network based codecs is that $\mathcal{E}$ and $\mathcal{D}$ are tuned using the backpropagation algorithm instead of by hand. Early work using auto-encoder architectures for image compression dates back to the late 1980s [30] and many extensions were developed during the following decades [31]. The use of recurrent architectures [32, 33] was proposed to allow configurable rate-distortion trade-offs by progressively encoding image residuals, this can also be improved by tiling [34].

The quantization operation is not differentiable. A multitude of approaches were proposed to deal with this because if used directly it makes training in neural network impossible. A number or methods were proposed to give this operation a gradient that makes sense. A simple approach was to perform quantization in the forward pass and consider the operation as the identity function during the backpropagation phase [35]. Another approach proposed a winner-takes-all mechanism where the values of the compressible representation with the highest values are passed through and the others are set to zero [36]. Another approach is to do a *soft* quantization using the softmax operation. This was proposed using scalar quantization [37] and vector quantization [38]. The scalar quantization method uses a set of L centres. Each input value is quantized by assigning it the index of the centre closest to itself during the forward pass as shown in the next equation.

$$\mathcal{Q}(z) = \arg\min_{j} \|z - l_j\| \tag{2.24}$$

The soft approximation used for the backward pass is represented in equation 2.25 where σ_q is a hyper-parameter to adjust the *softness* of the operation.

$$\mathcal{Q}(z) = \sum_{i=1}^{L} \frac{\exp(-\sigma_q \left\| z - l_i \right\|)}{\sum_{j=1}^{L} \exp(-\sigma_q \left\| z - l_j \right\|)} l_i \tag{2.25}$$

In this work we instead approximate the quantization operation with the addition of uniform noise during training as presented in [39]. More details are presented in the section 3.2.2. The last two quantization techniques are presented in the figure 2.13.

A lot of work has been invested in auto-encoder-style image compression in last few years. The Generalized Divisive normalization (GDN) activation function was introduced [40], and shown to work well to make activation maps more *gaussian*, helping with entropy coding. In painting was used as a way to improve compression performance by reducing to amount of information that needed to be encoded [41]. The use of an adversarial loss was used to provide more realistic-looking images at low bit rates [42].

Most relevant to our work is a hyper-prior network to produce a probabilistic model of the quantized latent representation [43]. The hyper-prior network uses the latent representation to build a probabilistic model of the latent representation. Later works extended the hyper-prior network by adding an autoregressive component to it [44] to take into account the local context using masked convolution. Another work used a 3D-CNN to compute a conditional probability model for entropy coding [37]. Generative adversarial networks (GAN) were used to create an extremely low bit rate codec [45]. The utility of GANs remains to be shown because they generate a sample semantically similar to the input but with different pixels values. It could be useful in environments with low diversity. Another work proposed to add a spatial energy compaction regularization [4] to force fewer elements of the compressible representation to hold most of the pixel information. Inspired by the immense success of the Transformer [46] architecture in every area of machine learning, the current state-of-the-art compression codec proposed a method that uses self-attention modules in a hyper-prior network. It also uses a mixture of gaussian as the entropy model [47]. In this work, we were not in pursuit of state-of-the-art compression performance, so we only use a basic hyper-prior network and we provide more details about it in the section 3.4. Future work using more recent techniques would be interesting but this is not the goal of the thesis.

The idea of using a compressible representation to do vision tasks is very new. An initial work proposed to use the JPEG representation as input to reduce computational and memory costs while maintaining good performance on the ImageNet classification task [48]. A recent work,

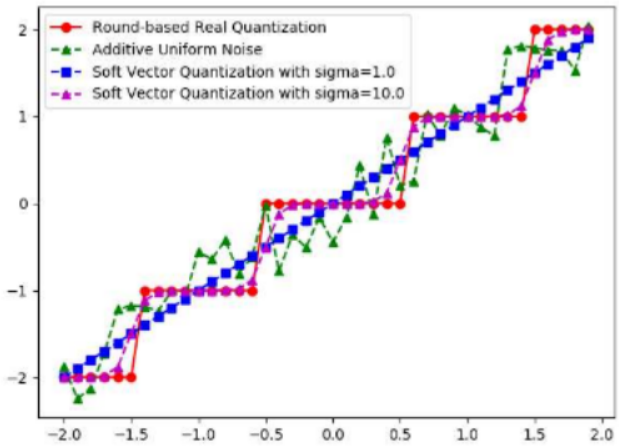


Figure 2.13 Comparison of *soft scalar quantization* with uniform noise approximation. Replicated from [4]

that was not out when this thesis started, proposed to learn a compressible representation as input for the classification task [49]. This work does not learn a compressible image format but only a compressible representation. Meaning that it is not training to recover the initial image. This is similar to early works that trained auto-encoders to learn image features via low-dimensional embedding, and did not explicitly attempt to compress the latent representations [50, 51].

To the best of our knowledge, the only work closely related to ours is *Towards image understanding from deep compression without decoding* [52], which was the first to perform classification and semantic segmentation directly from a learned compressible representation that is reconstructable. However, in their work, performance was compromised on semantic tasks, and they presented very crude techniques to take into account semantic tasks. In this work, we also explore inference from a compressible representation. However, we aim to develop an algorithm that produces *semantically sensitive compressible representations without sacrificing their ability to compute the information necessary to reconstruct the input image*. We perform on the same tasks, but we also add the detection and few-shot detection tasks. We demonstrate the superiority of our proposed format, particularly on out-of-domain, few-shot generalization.

Video compression has also been improved by neural networks methods. Most of these methods use the same core idea has the one used in engineered video codecs. The frames in the video feed are separated in two groups. First, the **I-frames** which are encoded using an

image compression codec. Second, the **P-frames**. They are not encoded but their difference with the closest I-frame is encoded instead. This difference is usually very small because there is usually very little change between frames the are close in time. Learned techniques use the exact same idea, but replace the image compression module by one presented above and add a learned prediction module. The difference image are also encoded using the learned image compression module. This is what this work [53] did and they currently hold the state of the art.

CHAPTER 3 SEMANTIC COMPRESSION USING NEURAL NETWORKS

3.1 Motivation

We observe that a lot of modern deep learning vision models are based on an encoder-decoder architecture. Densely labeled tasks almost always use this architecture. As an example, the recent DeeplabV3+ architecture used in semantic segmentation used this architecture. For more details about this see section 3.3.1. Also, to feed these models RGB images, a program must first uncompressed the images that are usually stored and shared using the JPEG format. This means that in most current image frameworks, two encoding and decoding steps must be applied to perform an image task. This is illustrated in the figure. 3.1 where t is a computer vision task.

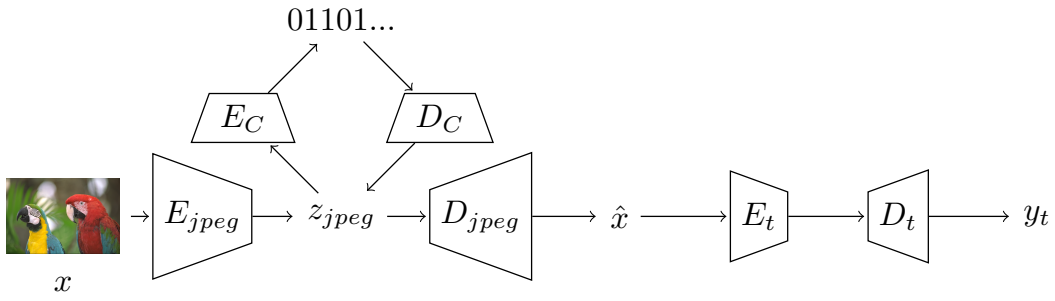


Figure 3.1 Typical vision pipeline to accomplish one vision task

In this figure, all blocks that start with an E are encoder models. The incoming arrow is on the larger side of the trapezoid and the outgoing arrow in on the smaller side. All blocks that start with a D are decoder models, and the arrow convention is the inverse of the one used for the encoder modules. The variable z_{jpeg} represent the filtered Discrete Cosine Transform (DCT) [54] coefficients of the input image. The module E_C in this figure represents the entropy coding algorithm encoder module, also named the compressor. It outputs a binary stream and D_C is the entropy coding algorithm decoder. The JPEG encoder and decoder are simple linear transforms, meaning that they are not representing optimally a non-linear input given a fixed representational budget, which is the case for lossy image compression. This means that there is a great potential for improvement in using non-linear encoders and decoders to get a compressible representation of the input images. Neural networks are non-linear units of computation that learn the best non-linear representation for a given task, so they have the potential to yield a much better image codec then JPEG.

We motivate this work by the intuition that if a lossy compressible representation can be used to reconstruct an image almost perfectly, it should contain enough information to accomplish any vision tasks. Then the representation of the image could be used directly as input to vision tasks with no need for a reconstruction of the image. In fact, it has been shown, in recent research, that the DCT coefficients obtained from the JPEG encoder could be used directly to achieve competitive results in image classification [55] while being much faster because there is no need to decode the image.

In this work, we propose the next logical step of this approach. We propose to learn a non-linear encoder, with neural networks, coupled with arithmetic coding to obtain superior compression and reconstruction performance compared to JPEG while maintaining high performance in a three vision tasks that only take the compressible representation as input. This new pipeline is presented in the figure 3.2. We note that in this new pipeline, the image reconstruction can be seen as just one more task that the compressible representation should be able to perform. On the other hand, we prioritize the image decoder over the decoders of the other tasks, because if the other tasks perform poorly from the compressible representation, they should still be able to obtain good performance from the reconstructed images, as it is already the case.

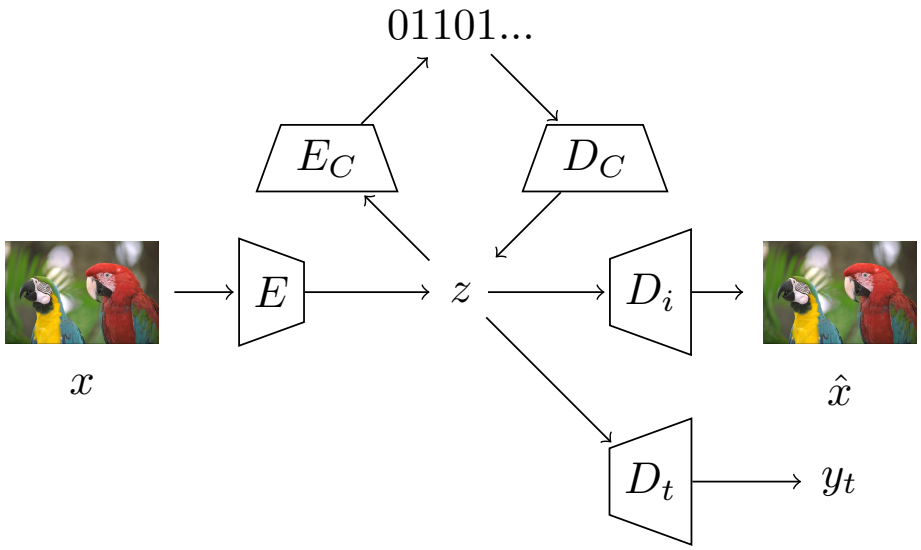


Figure 3.2 Proposed vision pipeline to accomplish one task other then image reconstruction

We call this representation semantically sensitive because the compressible representation is trained not only with a pixel-level task, such as image reconstruction, bust also tasks that extract semantic meaning from the pixel values. The tasks we chose are object classification, object detection and semantic segmentation and few-shot object detection. We add few-

shot object detection because we hypothesise that the compressed representation should yield better results for this specific tasks because the compressible representation should concentrate information making it easier to identify. If this shows to be true, then the compressible representation could become a new away to train low-shot detectors. This could prove useful in to world of robotics, where the available bandwidth is limited and where data acquisition is expensive, to personalise the experience to users that cannot label endless amount of new data such as people who are blind or have low vision.

If this method work, it would mean the the compressible representation of I-frames and P-frames could be used directly to accomplish the different vision tasks in a video feed.

3.2 Compression

In this section, we present the compression framework that we use to build our semantically sensitive compressible representation. Lossy compression using neural networks is framed as the minimization of the rate-distortion problem as formulated in

$$\min R + \lambda D \tag{3.1}$$

where D is the expected image distortion between the input image x and the reconstruction $\hat{x}$ and R is the expected code length/rate of z, a compressible representation of x. We use the transform coding framework for compression, where a representation of the input is compressed instead of the input directly. This is the case of JPEG image compression [27], where the DCT coefficients are encoded instead of the image directly. Our compressible representation is obtained with an autoencoder as presented in the section 2.2.5. The compressible representation will be encoded to a bit stream using the arithmetic coding algorithm presented in the section 2.3.1, and the associated probabilistic model will be presented in the section 3.2.2. The relative importance between D and R is given by λ. If that parameter is too high, only the rate becomes important and the network becomes a generative model. On the contrary, if λ is too low, only the distortion maters and the networks become the identity function with no regard for compression. By modifying λ we obtain a model operating at a different trade-off.

3.2.1 Quantization

Arithmetic coding can only be applied to a source with a finite alphabet, meaning that the compressible representation produced by the autoencoder cannot be directly used because it is a tensor made of real values. Instead of real values, we will use a finite subset of the

integers. This means that the real number is put into bins. The operation of transforming a real-valued tensor to an integer-valued tensor is called quantization. We choose to quantize with bins of length one. This function is also known as rounding to the nearest integer. The problem with this operation is that it does not have a useful derivative. Indeed, the derivative is either infinite of null as presented in the figure 3.3. This means that if quantization is used directly in the computational graph of a neural network, it will destroy the signal during the backward propagation, effectively halting learning.

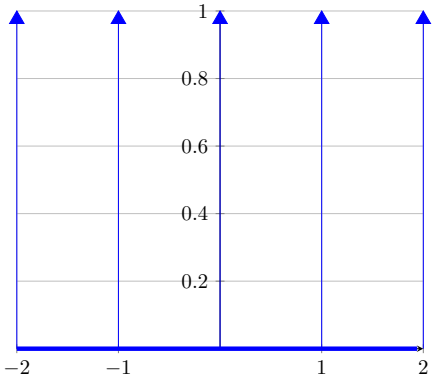


Figure 3.3 Derivative of the integer quantization function

We will denote the quantized version of a variable with a hat, for example, the quantized version of compressible representation z will be noted $\hat{z}$. To give the operation a useful derivative during the training phase, we model the quantization operation with i.i.d. additive uniform noise which has the same width as the quantization bins. During the test phase, actual quantization is performed because there will be no backward propagation. In effect this encourages the decoder to learn a mapping that is invariant to the fractional values of the codes. The result of this noise addition is that the density function of the quantized representation is a continuous relaxation of the discrete probability mass function of the representation.

In probability theory, the density function resulting from the addition of two independent random variables is obtained by the convolving of the density functions of the two random variables. In the single variable case, the two random variables are the representation z, with its associated probability mass function $p_z(z)$ and the noise with uniform density. We present the convolution of the two density functions in equation 3.2, where the symbol $*$ is the continuous convolution operation.

$$p_{\hat{z}}(\hat{z}) = \left(p * \mathcal{U}\left(\frac{1}{2}, \frac{1}{2}\right)\right)(\hat{z}) \tag{3.2}$$

In the annexe A.3 we show that this convolution of density function can be reduced to

$$p_{\hat{z}}(\hat{z}) = F_z\left(\hat{z} + \frac{1}{2}\right) - F_z\left(\hat{z} - \frac{1}{2}\right) \tag{3.3}$$

where F_z is the cumulative function of the density function $p_z(z)$. This means that using additive uniform noise in this fashion, we can model cumulative functions instead of density functions. This is what we do in this work.

3.2.2 Architecture

We used a compression architecture called the *hyper-prior* [56] architecture and it is presented in the figure 3.4. The boxes represent neural networks and the dotted lines represent quantization. In this architecture two quantized quantities will be encoded without loss us-

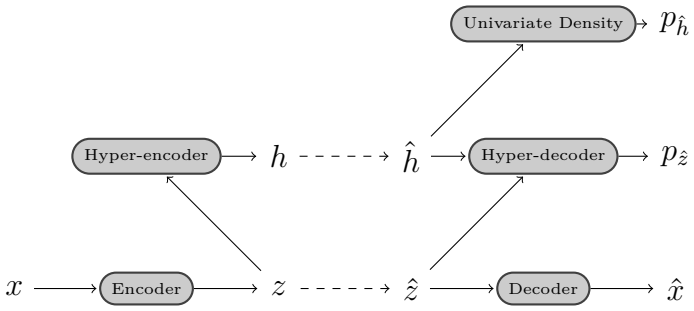


Figure 3.4 Compression framework.

ing arithmetic coding, then entropy coding algorithms presented previously. The first one is $\hat{z}$, which represents the quantized latent representation of the input x. It is obtained by quantizing the representation obtain through an autoencoder. The second is $\hat{h}$, which is a quantized representation of the side-information of the input x. Meaning the it only contains information to better encode a representation of x not the representation itself. It can be decoded, using the *hyper-decoder*, to produce the parameters of the probabilistic model of z. This is where the name *hyper-prior* comes from, because $\hat{h}$ is a prior on a probabilistic model which is itself a prior on $\hat{z}$. Next we present the two probabilistic models that are used to losslessly encode $\hat{z}$ and $\hat{h}$

Univariate Density Model

We present the neural network model we used to produce a probabilistic model over the quantized hyper-latent representation $\hat{h}$. This model is called the *univariate density model*

because it models the cumulative function of a one-dimensional random variable. To ensure that the learned function is a valid cumulative function we impose a few constraints in the set of learnable functions. The first one is that any real number input must output a number between 0 and one.

$$F : \mathbb{R} \to [0, 1]$$

This constraint is always satisfied by making the last layer of the neural network be a function that has an image of $[0, 1]$. As such we choose the sigmoid function. The next two constraints are about the output values at the limits of the input domain.

$$F(-\infty) = 0 \qquad F(+\infty) = 1 \tag{3.4}$$

The last one is that the derivative must always be greater or equal to 0 because a cumulative function must be a monotonic increasing function. This constraint is equivalent to saying that the corresponding density function p is always greater or equal to zero, because a negative probability is not feasible.

$$p(x) = \frac{d}{dx}F(x) \geq 0 \tag{3.5}$$

The last three constraints are followed by choosing a structure to the neural network that ensures they will always make sure that the constraint cannot be violated. In accordance with most deep learning architecture, we model the cumulative function as a composition of K vector functions $f : \mathbb{R}^{a_k} \to \mathbb{R}^{b_k}$. It is a neural network with K layers.

$$F = f_K \circ f_{K-1} \dots f_1 \tag{3.6}$$

For the model to be univariate we set the input dimension of the first layer and the output dimension of the last layer to be one, meaning that $a_1 = 1$ and $b_k = 1$. From the cumulative function, we can extract a definition of the corresponding density function p using the chain rule presented in equation 2.6. We find

$$p = f_K^{'} f_{K-1}^{'} \dots f_1^{'} \tag{3.7}$$

where $f_k^{'}$ is the Jacobian matrix of the layer f_k. We follow the last constraint by ensuring that all the elements of all the Jacobian matrices are positives. To accomplish this goal, we define all the functions as a modified linear layer followed by a special kind of activation function. This is formulated as

$$f_k(x) = g_k(A_k x + b_k) \tag{3.8}$$

where the activation function is

$$g_k(x) = x + a_k \odot \tanh(x) \tag{3.9}$$

To ensure that the Jacobian of that layer is positive we check the possible values. We take the derivative of the layer.

$$f_k^{'}(x) = \text{diag}(g_k^{'}(A_k x + b_k))A_k \tag{3.10}$$

We can see that if A_k has a negative eigenvalue, then the resulting derivative could be negative, so we force A_k to be a positive matrix using a reparameterization, where the learned matrix is $\tilde{A}_k$ and not A_k.

$$A_k = \text{softplus}(\tilde{A}_k) \tag{3.11}$$

We also take the derivative of the activation function to obtain

$$g_k^{'}(x) = 1 + a_k \odot \tanh^{'}(x) \tag{3.12}$$

In this function $\tanh^{'}(x)$ is bounded in $[0, 1]$ so for the whole derivative to always be greater or equal than zero, a_k must always be in the range $[-1, 1]$, so we reparametrize it.

$$a_k = \tanh(\tilde{a}_k) \tag{3.13}$$

Then the function f_k are formulated in this way

$$f_k(x) = g_k(\text{softplus}(\tilde{A}_k)x + b_k)$$
$$g_k(x) = x + \tanh(\tilde{a}_k) \odot \tanh(x)$$

Using this construction the last three constraints are met because of the monotone increasing nature of the model. This means that all the possible output distributions that this model can learn are valid probability cumulative function. This model can be made to be arbitrarily complex by using a variable number of composition function and a variable number output dimensions for the hidden layers. To give a probabilistic representation of a tensor of dimension (c, h, w), the model learns a different density distribution for each channel, by assuming that all elements on one channel follow the same distribution. So the network would treat a batched input of shape (B, c, h, w) as one of shape $(B, c, h * w)$.

Conditional Gaussian model

This model is the probabilistic model used in the arithmetic coding of the quantized compressible representation $\hat{z}$. We view this tensor as simply a list of N elements, where each element is an instance of a random variable unique to that element. This means that the probabilistic model over the whole representation can be formalized as

$$p_{\hat{z}}(\hat{z}) = \prod_{i=1}^{N} p_{\hat{z}_i}(\hat{z}_i) \tag{3.14}$$

Each element $\hat{z}_i$ is modelled by an independent Gaussian with its own mean μ_i and its own standard deviation σ_i. Both of these parameters are obtained by decoding $\hat{h}$ using the *hyper-decoder* network as presented in the figure 3.4. As shown before we use additive uniform noise to simulate quantization. We can formalize the density function $\hat{z}$ as the following.

$$p_{\hat{z}_i}(\hat{z}_i) = F_{\mathcal{N}}\left(\hat{z}_i + \frac{1}{2}\middle|\mu_i, \sigma_i\right) - F_{\mathcal{N}}\left(\hat{z}_i - \frac{1}{2}\middle|\mu_i, \sigma_i\right) \tag{3.15}$$

The cumulative function of a single variable Gaussian is known analytically if both the mean and the variance is known, which the case here, so it can be used directly.

To illustrate more the *Conditional Gaussian model*, we take the example of a element of z valued at 3.25. The corresponding quantized value is 3. To arithmetically encode this value we need a probabilistic model over the integers. The hyper-prior network predicted this number with a Normal function with a mean of 4 and a variance of 1. It is not perfect, because the mean could be better and the variance lower. The continuous density function is presented by the blue line in figure 3.5. To obtain a probability mass function we perform 3.15 at each of the integers. This is represented by the Diracs in the same pictures. In practice do not track the probabilitic mass of all the integers but only a small subset of them, specifically the one above a very low hyper-parameter value.

3.2.3 Hierarchical Variational Auto-Encoder

Now that all the different parts of the compression architecture are defined, we can describe how to actually train them.

In the univariate case, the added quantization noise around a value can be modelled as a uniform density model with the input as the middle point and the quantized representation as a sample from that distribution. We also model each element of the quantized latent representation $(\hat{z}, \hat{h})$ as independent, meaning that the density function of the whole density

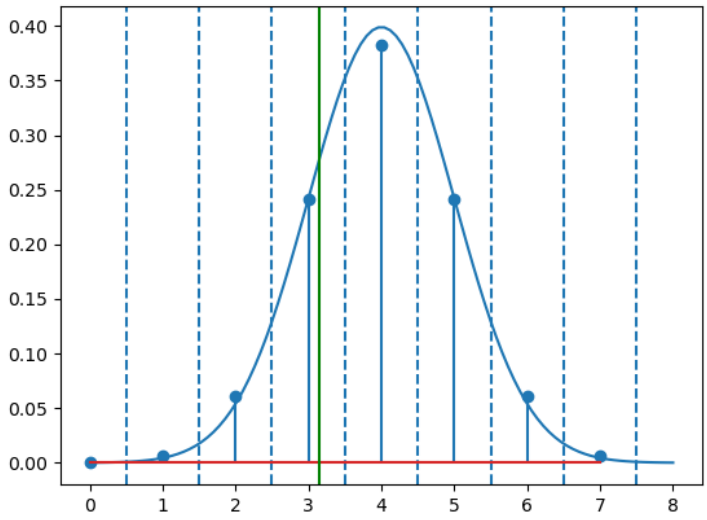


Figure 3.5 Probability model used for an element of the latent representation if the associated gaussian has a mean of 4 and variance of 1

distribution is given by multiplying the density function of all the elements. The resulting density function, given by equation 3.16, is uniformly distributed in the unit hypercube around the point (z, h) resulting in the processing of a given x.

$$q(\hat{z}, \hat{h}|x) = \prod_i \mathbb{1}_{\left[\hat{z}_i - \frac{1}{2}, \hat{z}_i + \frac{1}{2}\right]} \prod_j \mathbb{1}_{\left[\hat{h}_j - \frac{1}{2}, \hat{h}_j + \frac{1}{2}\right]} \tag{3.16}$$

The way to train this density function is to minimize the KL divergence between $q(\hat{z}, \hat{h}|x)$, the learned density function of the quantized representation $(\hat{z}, \hat{h})$ and the probabilistic model over it. This is formalized as

$$KL\left[q \parallel p_{z,h|x}\right] = \mathbb{E}_q\left[\log \frac{q(\hat{z}, \hat{h})}{p_(z, h|x)}\right] \tag{3.17}$$

As seen in annexe A.3, This is reduced to

$$KL\left[q \parallel p_{z,h|x}\right] = -\mathbb{E}_q[\log p(x|\hat{z}, \hat{h})] + KL\left[q \parallel p_{\hat{z},\hat{h}}\right] \tag{3.18}$$

This resulting objective function is the same as the one presented in equation 2.10, meaning that it is a VAE. To be more precise, it can be seen a Hierarchical Variational Auto-Encoder and $(\hat{z}, \hat{h})$, can be seen as a sample of the latent representation of the model. In this view,

quantization is equivalent to sampling. This is very similar to the reparametrization trick presented in 2.2.7. The resulting graphical model is the figure 3.6. $q(\hat{z}, \hat{h}|x)$ is the learned density function of the quantized latent representation, which is an approximation of the true posterior $p(\hat{z}, \hat{h}|x)$.

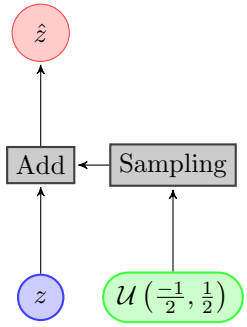


Figure 3.6 Computational graph of the additive uniform noise to simulate quantization, that is formally equivalent to the reparemetrization trick

The equation 3.18 can further be modified to obtain this form

$$KL\left[q \parallel p_{z,h|x}\right] = ||\hat{x} - x||^2 + E_q[-\log p(\hat{z}|\hat{h})] + E_q[-\log p(\hat{h})] \tag{3.19}$$

where $E_q[-\log p(\hat{h})]$ is the expected compression rate using entropy coding of $\hat{h}$ when the univariate density model is used as the probabilistic model and $E_q[-\log p(\hat{z}|\hat{h})]$ is the expected compression rate using entropy coding of $\hat{z}$ when the Conditional Gaussian model is used as the probabilistic model. Both of these expectations are cross-entropy losses as presented previously in section 2.1.

Equation 3.19 is not a compression loss because the trade-off between compression and image fidelity cannot be adjusted. So it is modified into the following equation

$$\min E_q[-\log p(\hat{z}|\hat{h})] + E_q[-\log p(\hat{h})] + \lambda||\hat{x} - x||^2 \tag{3.20}$$

In this formulation $||\hat{x} - x||^2$ is the D in equation 3.1 and $E_q[-\log p(\hat{z}|\hat{h})] + E_q[-\log p(\hat{h})]$ is R. We also note that this formulation is equivalent to the β-VAE presented in section 2.2.7. This means that the representations that will be learned will be disentangled, making them useful as features for other tasks because new tasks could focus on specific features to accomplish specific objectives.

3.3 Learning Compressible Representations for Machine Perception Tasks

The compressible representation and the set of encoded parameters are not uniquely defined, *i.e*, there is a multitude of ways to represent an image in a compressible space while keeping the image quality versus compression trade-off the same. This means that the same information could be encoded in a format that is more useful for tasks other than reconstruction while keeping the reconstruction and compression quality high. We propose to use the gradients coming from decoders of a set of computer vision tasks that extract semantic segmentation from the image while keeping the image quality versus compression trade-off roughly the same as a training signal for the encoder. This would render the compressible space *semantically* structured. We note this set of tasks $\mathcal{T}$ and it is composed of the three classical computer vision tasks.

(a) Input

Task	Output
Classification	Parrots (class=123)
Detection	
Semantic Segmentation	

(b) Vision Tasks

Figure 3.7 Visions tasks on the same input

This first downstream task is classification, it consists of predicting the class of the object in an image. The predicted class in among a set of classes that can be very large. The second task is object classification, which consists of classification with the addition of location. Object locations in an image are given by a tight bounding box around the object. There can be multiple objects of multiple sizes and of multiple classes in a single image. The last task is semantic segmentation, which is a generalization of the classification and detection tasks. Instead of predicting bounding boxes around objects, every pixel of the input image must be given a class. In addition to the object classes there is an additional background class, which the class given to all pixels that are not part of the object classes. These background pixels can represent the majority of the pixels, leading to a big imbalance in the

classes frequencies. There exist a multitude of ways to handle this problem, and we discuss the technique we used to mitigate this problem further in the text. As an example, we show in the figure 3.7b, the expected outputs of the image presented in the figure 3.7a in the three downstream tasks where yellow is the colour of the parrot class and black the colour of the background class.

The resulting architecture is presented in the figure 3.8. All the blocks are neural networks modules. where the downstream tasks are grouped inside the dotted lines. The classification decoder is noted D_C, the detection decoder by D_d, the semantic segmentation decoder by D_s and the image reconstruction decoder by D_i. The decoders D_C, D_D and D_S are separated into a *cBackbone* and a head. The *cBackbone* is an architecture that will be the same, but not shared, for all the downstream tasks. The head modules are task specific. In this pipeline, image reconstruction is simply one more task that the compressible representation must be able to accomplish. The different models composing the compressor are not shown here, because they were already presented previously in the figure 3.4, but they are grouped in the C model. Just like in the figure 3.4, the dotted arrow represents the quantization process presented previously.

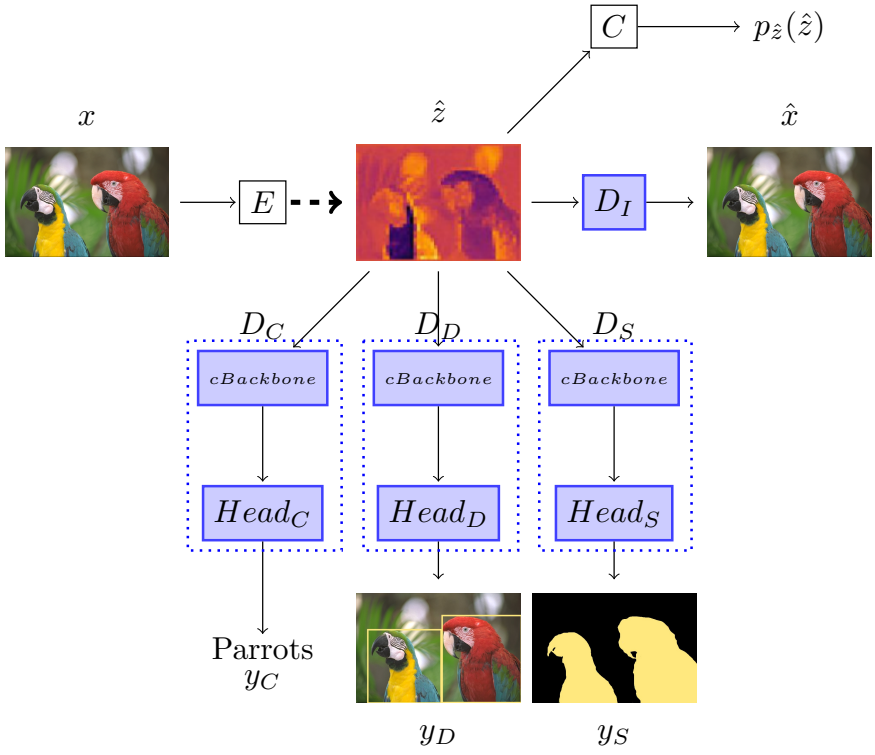


Figure 3.8 Semantic compression architecture

We propose a very simple way to combine the gradients of the downstream tasks. We add

the downstream decoder losses to the original compression problem and weigh them. This leads to the optimization problem we will solve in this thesis:

$$\min \lambda_c(\mathbb{E}[-\log_2 p_{\hat{z}}(\hat{z})] + \mathbb{E}[-\log_2 p_{\hat{h}}(\hat{h})]) + \lambda_r \, \mathbb{E}[\|x - D_i(\hat{z})\|^2] + \sum_{t \in \mathcal{T}} \lambda_t \, \mathbb{E}[l_t(D_t(\hat{z}), y_t)] \quad (3.21)$$

This can be understood as an extended problem in the Information bottleneck theory because we try to extract the relevant information in an input image but using Lagrangian multipliers. This is then a concrete example of the abstract theory of the information bottleneck.

3.3.1 Architecture

Image Compression framework

This encoder is composed of four convolutional layers interlaced by Generalized Divisive Normalization (GDN) layers [40] as activation functions. It was inspired by Divisive Normalization (DN) [57] which is a multivariate non-linearity designed to model the responses of sensory neurons in biological systems, and it was shown to play a role in efficient information processing. It is defined as:

$$z_i = \frac{x_i}{(\beta_i + \sum_j \gamma_{ij} |x_j|^{\alpha_{ij}})^{\epsilon_i}} \quad (3.22)$$

where x_i and z_i denote the input and output vectors. The parameters α_{ij}, β_i, γ_{ij} and ϵ_i are trainable parameters. i is the index of the element on which the activation function is computed and j is the index of elements in the neighbourhood of element x_i. The neighbourhood is the receptive field of a convolution operation. It has been shown that in practice, GDN, compared to common pointwise activation functions such as relu or tanh, yields better performance at the same number of hidden units and comes with a negligible increase in number of model parameters when used in an image compression pipeline using neural networks [40]. The first simplification of GDN we use is the following formulation:

$$z_i = \frac{x_i}{\sqrt{\beta_i + \sum_j \gamma_{ij} x_j^2}} \quad (3.23)$$

where $\alpha_{ij} = 2$ and $\epsilon_i = \frac{1}{2}$. We also use a more simplified version of GDN formulated as:

$$z_i = \frac{x_i}{\beta_i + \sum_j \gamma_{ij} |x_j|} \quad (3.24)$$

As formulated in [58] were $\alpha_{ij} = 1$ and $\epsilon_i = 1$ that was shown to give the same results in a compression framework while being faster. Also, it showed greater stability in training.

This is due to the fact that the square root of a very small number produces an even smaller number, and a small number as a denominator in a division in a computational graph can lead to very big gradients with respect to the norm of the weights for the numerator, causing instability during training.

The decoder uses the Inverse Generalized Divisive Normalization (IGDN) and it is defined as :

$$z_i = x_i(\beta_i + \sum_j \gamma_{ij}|x_j|^{\alpha_{ij}})^{\epsilon_i} \tag{3.25}$$

It uses the same operation as the GDN, but instead if dividing by the normalization factor, it multiplies by it. This is seen as doing the inverse work of the GDN, this is why it is used in the image decoder because it does the inverse function of the encoder. We used the same simplifications in the IGDN as the ones we used for the GDN, when it was used in the image decoder.

The encoder expects an RGB image of size $3 \times W \times H$ and outputs a tensor of shape $256 \times W/16 \times H/16$. Note, the rank of the compressible format is a third of the input rank and is still much smaller than the input (in bits/pixel) because of its smaller spatial resolution and redundancy. The actual architecture of the encoder and the image decoder are presented in the table 3.1. In this table, *Conv* represents a convolution layer where k represents the kernel size, c represents the amount of output channels and s represent the stride. We use the stride to do the downsampling instead of using a pooling operation. *TConv* is a transposed convolution, it is used to upsample the representation.

Encoder	Image Decoder	Hyper-Encoder	Hyper-decoder
Conv : 5k, 256c, 2s	TConv: 5k, 256c, 2s	Abs	TConv 5k, 256c, 2s
GDN	IGDN	Conv : 3k, 256c, 2s	Leaky ReLU
Conv : 5k, 256c, 2s	TConv: 5k, 256c, 2s	LeakyReLU	TConv 5k, 384c, 2s
GDN	IGDN	Conv : 5k, 256c, 2s	Leaky ReLU
Conv : 5k, 256c, 2s	TConv: 5k, 256c, 2s	LeakyReLU	TConv 5k, 512c, 2s
GDN	IGDN	Conv : 5k, 256c, 2s	
Conv : 5k, 256c, 2s	TConv: 5k, 3c, 2s		

Table 3.1 Architecture of the modules used in the compression framework

Resnet

The Resnet-101 is used as a baseline backbone for the different vision downstream tasks when going from the RGB images. This will produce a strong and well-known baseline.

Block name	Resnet	Truncated Resnet	SubPixel Resnet
Stem	$\begin{bmatrix} 7 \times 7, 64, \text{ stride } 2 \\ 3 \times 3, \text{ MaxPool, stride } 2 \end{bmatrix}$	-	$\begin{bmatrix} \text{Block(256, 64)} \\ \text{PixelShuffle(2)} \\ \text{Block(64, 16)} \\ \text{PixelShuffle(2)} \\ \text{Block(16, 16)} \\ 1 \times 1, 64 \end{bmatrix}$
layer2	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	-	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
layer3	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 512 \\ \begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 2 \end{bmatrix}$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$
layer4	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$
layer5	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$

Table 3.2 Resnet Architectures

Truncated-Resnet

When going from the compressible representation, we want to use a backbone architecture that is similar in design and complexity to the Resnet-101 backbone. This what is called the *cBackbone* in figure 3.8. The first architecture that we used is the architecture we call the Truncated-Resnet. We cannot use the Resnet stem. The compressible input is of a different dimensionality and is not compatible with the Resnet backbone. Based on the architecture used in [55], we adapt the Resnet backbone by removing the stem, layer 1 and the first two residual blocks of layer 2. The reason we chose to remove those layer is that the spatial resolution of the tensor at this point of the Resnet network matches the spatial resolution of the compressible representation. To match the number of feature maps of the tensor expected at this point in the Resnet architecture, we add a 1×1 convolutional layer to increase the number of feature maps of the compressible input format. We do not change at all the layers 3 and 4 of the Resnet architecture. This way, the number of parameters in the original Resnet architecture and the truncated architecture is very similar, and most of the architecture is the same. This makes it fair to compare performance metrics on the different vision tasks when going from images versus when going from the compressible representation.

SubPixel-Resnet

We developed this Resnet variant, because we obtained subpar result from the Truncated-Resnet architecture. In this new architecture, we keep the layers 1 to 4 the same as in the Resnet architecture, but we propose a new stem that is more adapted to the compressible representation as input. This way, this architecture is mostly the same as the original Resnet architecture, making comparison between them fair.

First, the expected output tensor spatial dimensionality of the Resnet stem is 4 times smaller than the input image. Our compressible representation has a spatial dimensionality that is 16 times smaller than the input image, which means that we need the output representation of the Sub-Pixel Stem must be four times greater than the ones of the compressible representation. The usual solution in this case is to use a deconvolution operation. We instead chose to use the pixel-shuffle operation followed by convolutions. This operation is recent and was originally proposed in the context of single image super resolution (SISR). The goal of that research was to produce a high-resolution image from a low-resolution input and they achieved state-of-the-art results at the time [59]. The process is shown in figure 3.9 and is called a sub-pixel convolution when it is followed by a convolution operation. It was proven

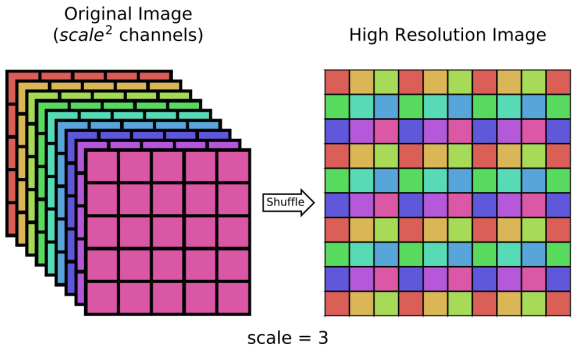


Figure 3.9 Pixel Shuffle operation with a scale of 3

that to me equivalent to a deconvolution. The way it works is by moving the low-resolution channels into the spatial dimension to create an image with a higher spatial resolution. This is a more efficient transformation then the transposed convolution operation. It does not increase the receptive field, this is why it called a sub-pixel convolution. Also, it has been shown [7], that the sub-pixel convolution is better than the deconvolution in a key way that is interesting to this research. Devonvolution creates checkerboard artifacts in a multitude of tasks when it is used to increase the spatial resolution. To alleviate this checkerboard problem a solution that was used what to upsample the representation using bi-cubic inter-

polation and then do a convolution. The sub-pixel convolution does not create checkerboard artifacts and replaces the up-sampling steps. We separate the 4 times spatial up-sampling in two separate operations that will each up-sample by a factor of two. This is done, because preliminary results showed that it was yielding better performance.

Second, we change the convolution in the original stem with a custom residual block, similar to the one used in Resnet architecture, but only composed of convolutional layers with a kernel of size 3. It is presented in the figure 3.10. The three consecutive convolutions with kernel size of 3 have the same receptive field as a single convolution layer with a kernel of 7. This replaces the convolution in the stem of the original Resnet architecture, while having fewer parameters. This type of replacement has been shown to improve performance in vision tasks both in performance metrics and in speed. The normalization function we used is batch normalization, but we also started to make tests using group normalization to mitigate the very low batch size in the vision tasks.

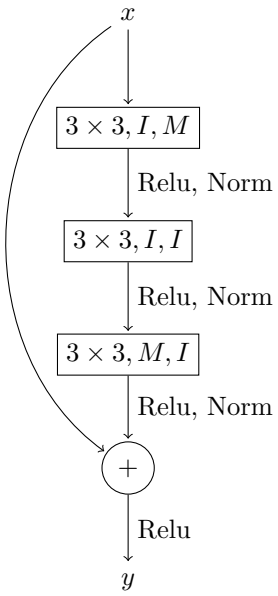
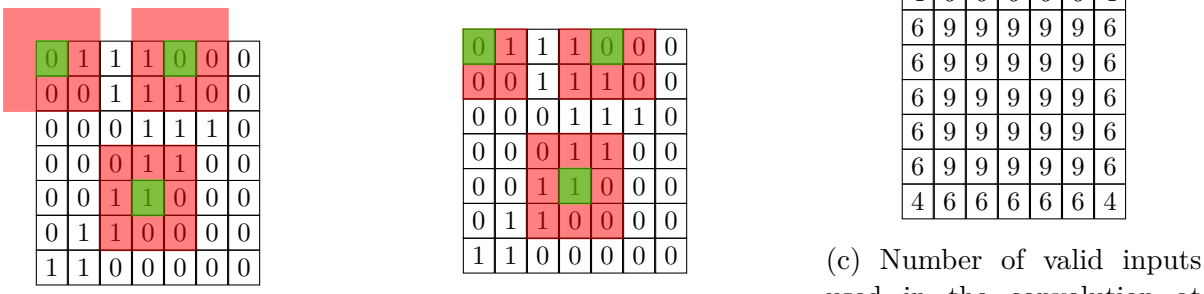


Figure 3.10 Residual Block

Third, we do not use regular convolution layers in the residual blocks. We use rectified convolutions. These convolutions were developed to reduction the introduction of artifacts when padding is used in regular convolutions. Padding is the process of adding elements around the input tensor of a convolution to alleviate the fact that the output of a full convolution has a smaller dimensionality than the input. There exist a multitude of methods to do padding, the main ones are padding with zeros or padding by reflecting the input. Both of these methods give satisfactory results, but introduce signals in the convolution that

were not in the input image. This means that the convolution outputs at the edged might be biased towards the signals that are not in the input tensor. The solution proposed by rectified convolution is to do the convolution with zero padding but to reweigh the output that used padding as input in a manner similar to dropout [60].



(a) Normal Convolution

(b) Rectified Convolution

(c) Number of valid inputs used in the convolution at each position of the output feature map

Figure 3.11 Comparison of a 2d convolution of a 7x7 feature map with a 3x3 kernel with an equivalent rectified convolution. The red squares represent the inputs used in a convolution computation, and the green square the output location.

This means that the residual blocks after the first pixel shuffle do not increase the receptive field of the stem, this means that the receptive field of the sub-pixel stem and the basic stem is the same. The compressible representation being smaller, the receptive field of a single output neuron of the sub-pixel stem covers 19 % of the compressible representation and a single neuron of the basic steam covers 0.1 % of a 256 by 256-colour image, which is the minimum image size we used for training. This gives this architecture potential for better performance which we will see in the results section.

Classification Head

As for the classification head, we use the one proposed in the original Resnet Architecture. This architecture consists of two layers. The first one is a global average pooling layer, that averages all the entries of each activation map. This means that the input can be of any shape because the spatial dimension is removed with the average pooling. The input of this layer had 2048 channels, this means that the output of the pooling layer is a vector with 2048 elements in it. The last layer is a linear layer to the number of classes. The way to measure the performance of this head is by tracking *top-k* accuracy. It measures the expected likelihood of finding the true label in the k classes that the networks predict as the most likely.

In this work, and in the literature in general in classification, the *top-1* and *top-5* metrics are tracked.

Detection Head

For the detection task we tested with two detection heads. The first detection head we used is the detection head proposed in the classic detection paper that presented the Faster-RCNN architecture [61]. The figure 3.12 presents the different modules of this architecture and how they feed each other. The Region Proposal Network (RPN), takes an image representation as input and outputs a set of rectangular object proposals, each with a score to indicate the likelihood of an object being in that rectangle. The Region of Interest (ROI) Pooling modules aggregates the proposals to remove as many duplicate proposals of any given object. The ROI Feature Extractor module extracts the relevant activation maps from the output of the back for each proposal. The box classifier if a very lightweight module that predicts the class of the object inside each proposal. The Box Regressor refines the bounding box coordinate proposals. The backbone used is the Resnet when going from images and Truncated-Resnet or Sub-Pixel Resnet when going from the compressed representation. In this architecture, the output of the backbone that is used is the output of layer 4. In the literature this called a C4. This is the most basic input to a detector in R-CNN family of architecture. When going from the compressed representation, the figure 3.12 it not totally accurate because the backbone takes the compressible representation of the image instead of the image as input.

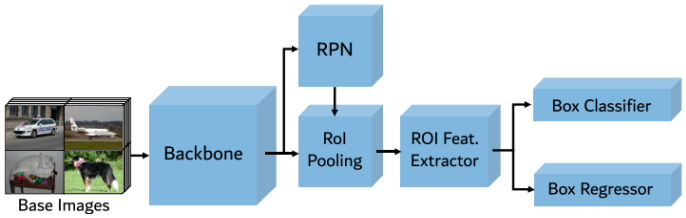


Figure 3.12 Faster R-CNN is a single, unified network for object detection. Replicated from [5]

The second detection head we used is the Feature Pyramid Network (FPN) [6]. It is still a detector in the RCNN family of architecture, but the difference is in the features used by the detection modules. In computer vision, great improvement in almost any task can be achieved when features at multiples spatial resolution/scales are used. This is an old technique [62] that is still used to achieved state-of-the-art results. In this model, instead of using the C4 features, a set of features at multiple scales produced by an FPN is used. This set of features was proven to work really well for detection tasks on a multitude of datasets. These features are produced by using the features outputted by each of the layers 2, 3, 4 and

5 and combining them in a simple way. High-level representations that have a low spatial resolution are mixed with lower-level representation that have higher spatial resolution by using skip connections and upsampling of the feature maps with low spatial resolution. This process is presented in the figure 3.13.

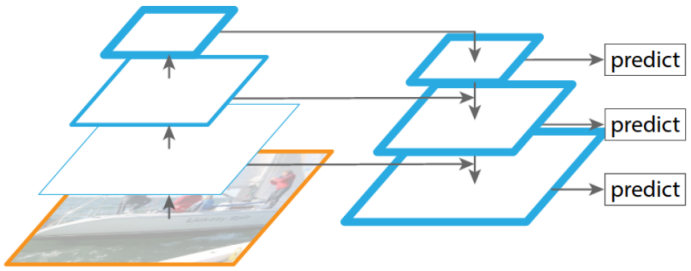


Figure 3.13 Feature Pyramid Network features. Reproduced from [6].

The rest of the network is the identical to the Faster-RCNN detection head. It was used to get better overall results. It was included when major refactoring of the code base was done. The metric used is called the Average Precision (AP) and its variants. Precision measures the percentage of predictions that are correct. In object detection, to consider if a bounding box is correct, the Intersection over Union (IoU) is used. It is defined as the area the Intersection between the predicted bounding box and the truth bounding box divided by the area the union between the predicted bounding box and the truth bounding box. This value is by construction always in the [0,1] range, with 0 being the worst possible score and 1 the best. A threshold on the IoU is used to indicate if a prediction is considered correct. If multiple prediction of the same object is produced, the first one is considered positive and the subsequent one are considered negatives. Average Precision takes its name because it averages the precision obtained at a recall value going from 0 to 1 by 0.1 increments. The recall being the number correct predictions (true positives) divided by the number of correct predictions plus the number or object that was predicted (False Negative). The IoU thresholds range from 0.5 to 0.95, leading to the metrics called AP5 to AP95. In this work we track the AP50. We also track the mAP for mean Average Precision that averages the metrics AP5 to AP95 into a single on. Also, because there are multiple classes, the metrics are averaged over all classes.

Semantic Segmentation Head

We used a well-known class of semantic segmentation architecture called *Deeplab*. It is based on two main ideas. This first one is to use dilated convolution instead of regular

convolution in a Deep Convolutional Neural Network backbone such as Resnet to extract useful features. In this case the backbone is used in a similar way as in object detection. This type of convolution gives a wider receptive field to the convolution operation at a given number of parameters with no increase in computation. The increase of the receptive field is parametrized by a dilation factor. The second idea is the Atrous Spatial Pyramid Pooling (ASSP) module, which computes in parallel the same input with dilated convolution with different dilation factor followed by a concatenation. The different dilation factor makes the AASP a multiscale module. We use the one of the latest iteration of the DeepLab architecture called *DeepLabV3+* [7] which is presented in the figure 3.14.

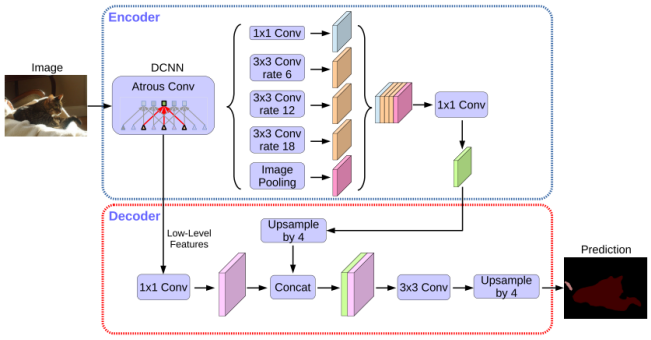


Figure 3.14 DeeplabV3+ architecture. Replicated from [7]

In our case, we adapt the DeeplabV3 for our compressible representation. The input of the Deep Convolutional Neural Network backbone is not an image but the compressible representation, but we cannot use the same Truncated-Resnet and SubPixel-Resnet as described before directly so we simply adapt them a little. We are removing some downsampling that occurs in layers 4 and 5 and by changing the regular convolutions to dilated convolutions. Also, in the case of the Truncated-Resnet, we remove the skip connection between the layer 2 and the DeeplabV3+ because there is no layer 2 in the Truncated-Resnet Resnet Architecture.

A problem in semantic segmentation is the big imbalance in the number of pixels that represent object classes and pixels that represent the background. The way we choose to tackle this problem is by not choosing the usual semantic segmentation loss which is a simple cross-entropy loss averaged over all the pixels. This average does not take into account the fact that some classes are easier than others and the fact that some classes are much more present than others. Instead we choose to use Hard Pixel Mining [63] loss as presented in equation 3.26, where y_i is the true class of pixel i and $\mathbb{1}[p_{i,y_i} < t_K]$ is an indicator function to select only the pixels with the top-K highest loss.

$$l_{ss} = -\frac{1}{K}\sum_{i=1}^{N} \mathbb{1}[p_{i,y_i} < t_K] \log p_{i,y_i} \tag{3.26}$$

This modified cross entropy loss focuses only on the pixels that are harder meaning the ones coming from inherently harder classes and lower count classes. This results in a little gain in performance and we choose to focus on the 15 % of pixels with the highest individual losses.

Few shot Detection

We tested a recent paper [5] that does few-shot detection in a very simple manner. Instead of using meta-learning as presented in the section, 2.2.9 this paper works in a two steps approach. First, a detection network, just like the one presented in the detection head section 3.3.1, is trained on images containing objects from a set of base classes. Second, the whole network except for the output modules that are the *box classifier* and the *box regressor* is fixed. Those two modules are then reinitialized and trained using $K \in \{1, 2, 3, 5, 10\}$ samples from each class including the base classes and the novel classes. This two-step procedure is presented in the figure 3.15. It presented very promising results and we hypothesis that the compressible representation would be a good input for few-shot learning because the representation is smaller and more disentangled that the raw image. We train this task in the same way we do simple detection, meaning that the same hyper-parameters are used. Only the dataset is changed.

3.3.2 For multitask Learning Difficulties

A major difficulty of this work was training in a multitask fashion the different decoders. We present here some techniques that we tried to make the training faster and more stable.

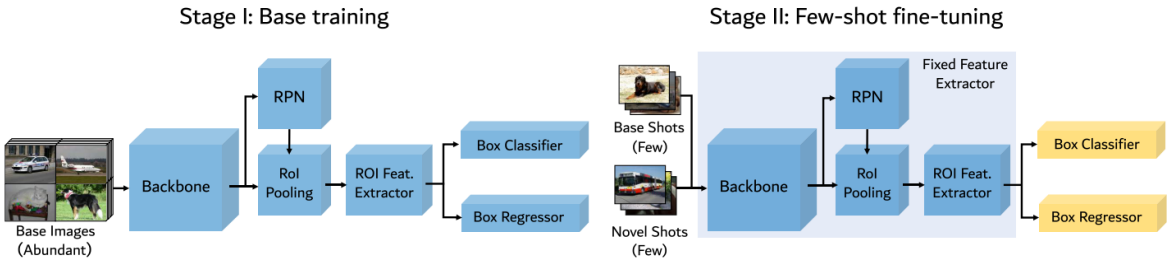


Figure 3.15 Two-stage fine-tuning approach to few-shot object detection. Replicated from [5]

Adaptive Compression objective

A time-consuming problem that became evident when doing joint training was to find the right λ_c, λ_r and the different λ_t to obtain a compression ratio that was close the desired operation points we chose. To mitigate this problem we modify the rate loss to incorporate a target rate R_{target}. The rate loss would be zero if the achieved compression rate is lower than the target rate and would act as ReLU if the achieved rate is bigger than the target rate. This is presented in the next equation.

$$R_{\text{loss}} = \max(\mathbb{E}[-\log_2 p_{\hat{z}}(\hat{z})] + \mathbb{E}[-\log_2 p_{\hat{h}}(\hat{h})] - R_{\text{target}}, 0) \tag{3.27}$$

This is inspired by the hinge loss used in the training of the classical support vector machines (SVM). The target rate is similar to the margin, and the loss is unaffected by how much a correct prediction is good. It doesn't matter if the achieved target rate is close or far below to the target. We motivate this modulated loss on the rate by the intuition that if there is no restriction on the compression rate below the target, the only gradient used to update the weights is coming from the downstream tasks. Then, these tasks will try to pack as much information as possible into the bottleneck, which will increase the achieved rate up to the target. It should lead to a representation that is more semantically guided because it is not guided by anything other than the semantic tasks if the compression is good enough.

CHAPTER 4 EXPERIMENTAL RESULTS

4.1 Implementation details

We implemented this work using the Python programming language and the Pytorch [14] deep learning framework. Choosing this framework over Tensorflow [15] allowed us to use the recent Detectron2 [64] library. It includes optimized CUDA kernels for common detection modules such as ROIalign and Non-Maximum Suppression [65] for object proposals and benchmark implementations of commonly used models. It is currently the fastest detection library available the time of writing across all deep learning frameworks. We also adapted the Detectron2 library to make is suitable for the classification and semantic segmentation downstream tasks. This made it possible to have all tasks working harmoniously because they were based on the same code base. For the few-shot detection task, we adjusted the code that was used in the paper [5] in a small way, because it was also built using Detectron2.

For the compression framework, there was no prior implementation in Pytorch, so we wrote our own. This might be the one of the reasons reason why we obtain slightly lower performance when compared to the Tensorflow implementation [66].

4.2 Datasets

For the image reconstruction task we used the Kodak Dataset [67] as the testing set, which is a collection of 24 natural images with a side length of 768 pixels, stored into a lossless image codec. It is often used a benchmark dataset to measure image quality in a multitude of computer vision tasks including image compression. There is no benchmark training set associated with it and the training set made up of millions of natural images used to train state-of-the-art models at Google [39, 44] is not available so we made up our own. We used 350K images from the Open Images dataset [68], which is set of images collected on the internet. We filtered the images using simple heuristics to only get natural images. These images were originally stored used the JPEG codec, so there are compression artifacts in the images original images. To remove as many compression artifacts as possible, we only used very high resolution images and we downsampled them to be roughly the same size as the images in the Kodak Dataset.

For the downstream tasks, we used the most popular dataset associated with each task. For the classification task, we used the ImageNet dataset [18], which is made of natural images categorized into 1000 categories. It consists of 1.3M training images and 50K test images. It

is considered a landmark dataset on which to perform computer vision experiments. For the object detection task, we used the Pascal VOC Detection [69] dataset which is a detection task over the images of Pascal VOC. The objects are categorized into 20 categories and the dataset consist of 11K images with 27K annotated objects. The few-shot detection was done on the same dataset, and the same shots were used across the experiments. We test on three different splits of shots. For the semantic segmentation task, we use the same dataset as the detection task but it has only 6.9K annotated images. To alleviate the small size, we combined it with another dataset, the Semantic Boundaries Dataset [70], which add another 5.6K annotated image.

4.3 Training procedure

We decided to train all the compressors in this work at two bit rate The first one is 1 bit-per-pixel (bpp). At this compression ratio, the JPEG codec produce on average an image quality of 33 db. At this PRNR the loss in quality in barely perceptible, but small compression artifacts are noticeable. The second operation point we chose is 0.4 bit-per-pixel at which the JPEG codec can produce an image quality of 28 dB, which is clearly noticeable by the human eye. For comparison a lossless image compression codec can compress an image at around 5 bpp. The two operation points are presented in the figure 4.1 using an image from the test dataset of the image reconstruction task.

(a) Uncompressed (b) 1.0 bpp (c) 0.4 bpp

Figure 4.1 JPEG image quality at two different compression ratio on the image Kodim15 of the Kodak dataset

To better see the difference in image quality we zoomed into the original image at the same position, this is because compression artifacts are barely noticeable when the whole image is viewed. Having operation points at high and very high compression ratio let's use observe the

advantages of using our solution over of modern compression codecs in environments where the available bandwidth is limited.

We call **Naive** learned compression when we train the compression architecture, as presented in the figure 3.4, by simply minimizing the equation 3.20. We call **Informed** learned compression, the compressors resulting from joint training of the image reconstruction task with the downstream tasks. For better numerical stability was trained all the compressor in images with values scaled to the [0,1] range.

4.4 Compression results

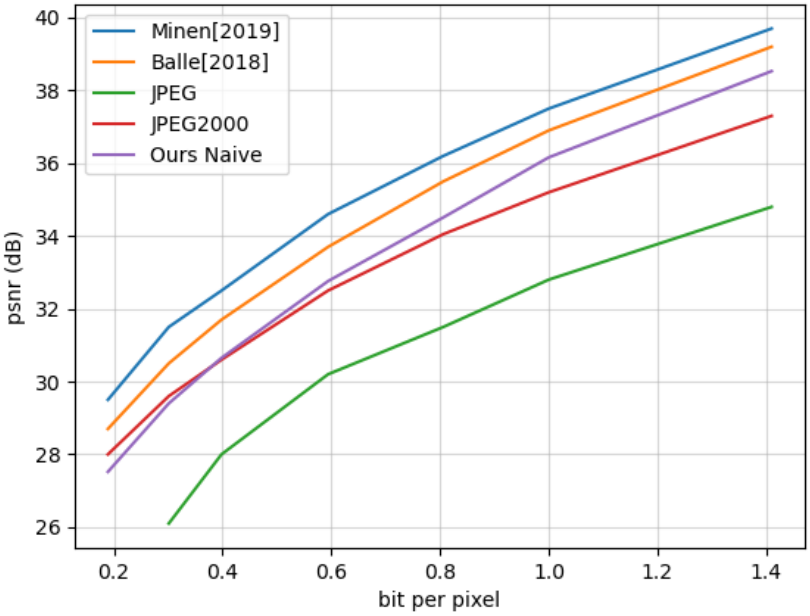
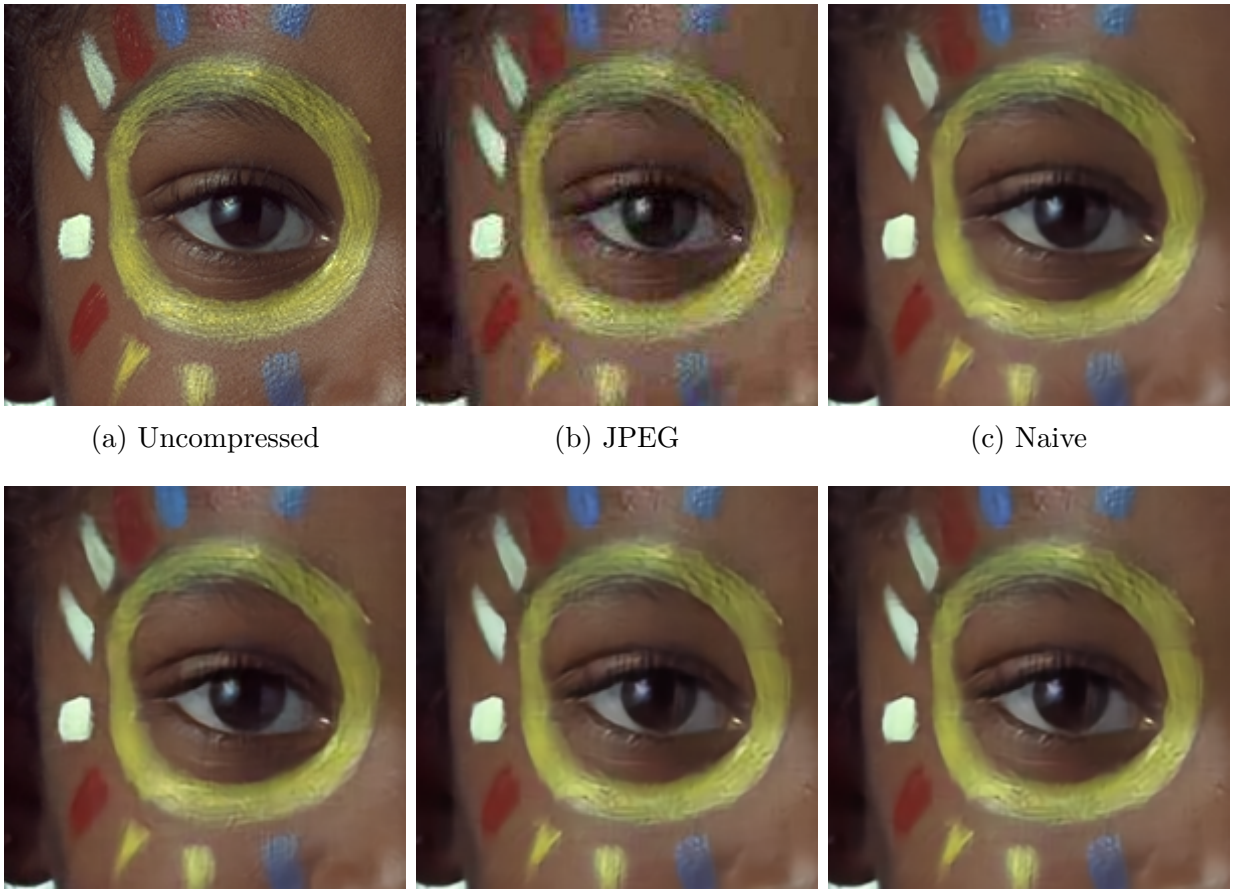


Figure 4.2 PSNR versus bits-per-pixel curve for different methods

To train all the compressors we used the Adam [13] with a learning rate of $5 \times 10^{-}5$ because any higher learning was causing instability. Instability was s major source of difficulty because the system would become unstable and go to a performance worse than random initialization to then recover to a better performance than what was obtained before the instability. This made is very hard to train reliably even more in the case when multiple downstream tasks were connected to the compressor. We use big square patch size of a side length of 512 pixels of the input images because we found that we get better results this way instead of using smaller patches. This is the biggest image size we could use because if we trained with bigger

image sizes, the batch size we obtained was too small, which lead to training instabilities and long training times. The batch size we used was 8. We present the learning curve we obtained in the figure 4.2.



(a) Uncompressed (b) JPEG (c) Naive

(d) Classification (e) Detection (f) Classification and Detection

Figure 4.3 Image quality of our different compressor on the Kodim15 image of the Kodak dataset at a compression rate of 0.4 bit-per-pixel

We note that our model is always better than the JPEG compressor and its successor JPEG2000 at every bit rate. This means that our learned compressor is strictly superior to the most used handcrafted compressor. We can see our model performs a little bit worse than the proposed state-of-the-art hyperprior methods of [43] and [44]. We explain this difference by two factors, the high instability of our training, our insufficient dataset and our insufficient training times. In the state-of-the-art papers, the dataset was said to be constituted of millions of images and that the convergence was obtained in a few weeks. We only trained on 350K images for a few days at a time. At the moment of writing we are investigating the source of the instability, and we are currently thinking that our dataset in

partially responsible for the instability, be we are not sure of this.

For the case of classification informed compression, the only downstream task is classification. We use multi-class cross-entropy for as the classification loss. We train jointly both tasks for 70 epochs dividing the learning rate by a factor of 10 every 30 epochs. In order to keep both tasks on the same schedule, we set a batch size of 72 for image reconstruction, and a batch size of 256 for the classification task. During training we had a lot of instability when multitask learning was involved, especially when detection was involved. On was we found that reduced a lot the instability was to pretrain the decoder weights from the fixed the naive encoder weights and to then train them jointly. For the case of detection informed compression, the batch size for the different decoders was 16 for detection, 72 for the image reconstruction and 256 for classification.

In the figure 4.3, we present the same patch in the image Kodim15 of Kodak datasets at the same compression rate of 0.4 bpp using the different compressor we trained, in comparison to the JPEG compressor. We can clearly see that the learned compressor create a more appealing image because it does not introduce block artifacts. The learned compressors can keep the structure in the images but they tend to have a smoothing effect, erasing small details.

4.4.1 Hinge rate

We were not successful in making the hinge rate loss for the compressor work. We did not make extensive study of this technique because it was showing unpromising performance as shown in the table 4.1. For a similar bit rate, the image quality is much worse. A difference of 6 points in image quality was not worth an easily adjustable compression performance.

Compression Loss	bit-per-pixel	PSNR
Weighted	1.00	36.2
Hinge	0.99	30.5

Table 4.1 Hinge rate performance

4.5 Downstream Tasks results

When training the downstream tasks, the encoder is fixed and only the decoder parameters are trained. To obtain better performance in the downstream tasks of semantic segmentation, detection and few-shot detection we use transfer learning by using classification weights. This technique is wildly used in the machine learning community to obtain better performance

on a small dataset by pretaining on a big dataset. Then, for all the compressor we first trained the ResNet architecture on the learned compressible representation. Then we used the resulting weights to initialize the ResNet backbones of the other downstream tasks.

To train the classification task we use the procedure proposed in [71]. We trained for 100 epochs, with a batch size of 256 with a learning rate of 0.1 the was divided by a factor of 10 every 30 epochs. To train the semantic segmentation task we used patch size of 512, a learning rate of 1×10^{-3} that was divided by 10 every 10 epochs. All downstream task training was done with the SGD optimizer because it was empirically shown to have better generalization properties in classification and detection. All training was done with 4 Nvidia V100-16GB GPUs using mixed precision training.

To compare the performance of the algorithms with JPEG at the two bit rates, we trained the different downstream tasks on images that were first compressed and decompressed using JPEG. For the 0.4 bpp we used a quality level of 12 and for 1.0 bpp we used a quality level of 45. We found these values by compressing the whole Pascal VOC dataset using JPEG and we found the quality level that had the closest bit rate to the target.

4.5.1 Truncated Resnet

The results on the classification and the segmentation tasks are presented in the table 4.2.

	Input	Segmentation mIoU	Classification Top 1	Top 5
	RGB from standard 4 to 5 bpp JPEGs	70.75	77.32	93.33
bpp ~ 1.0	RGB - compressed JPEG at 1.0 bpp	67.52	75.02	92.22
	Naive Learned Compression	64.38	69.80	89.24
	Class Informed Compression	62.73	74.83	91.93
	Detection Informed Compression	64.13	71.66	90.46
	Class & Detection Informed Compression	**68.70**	72.71	90.96
bpp ~ 0.4	RGB - compressed JPEG at 0.4 bpp	57.76	70.73	89.80
	Naive Learned Compression	64.39	70.34	89.75
	Class Informed Compression	61.0	72.60	90.81
	Detection Informed Compression	**64.82**	71.57	90.47

Table 4.2 Performance comparison on the downstream tasks when going from different compressible representation using the Truncated-Resnet as the downstream backbone architecture.

In contrast to what was observed in [52] we see that a classifier trained from the ordinary 1 bpp compressible representation achieves inferior performance (69.80 top-1) compared to

a classifier trained from 1 bpp decompressed images (75.53 top-1 accuracy). However, we show that it is possible to obtain a more semantically structured latent representation that closes this performance gap, while maintaining the performance benefits of training from a compressible representation. Our compressor that has been jointly trained with classification tasks, increases the top-1 classification score to 74.83. Detection informed and jointly informed formats both improve performance, but not as significantly as the classification only jointly trained format. At ~ 0.4 bits per pixel we begin to see the true advantage of training from task informed compressible representation. The naive learned compression is already able to match the results obtained by 0.4 bpp decompressed images. When we use classification or detection informed formats we are able to surpass the results that use images of equal bit rate as input. We believe that this clearly indicates that learned compression formats are not unique and that there is plenty of room to bias them towards structuring representations so that they are better suited for subsequent semantic tasks.

For semantic segmentation, using decompressed images at bpp 1 there is a drop from 70.75 to 67.52 in performance. Training from the naive learned compressible representation yields slightly worse performance of 64.38 mIoU. The class informed compression and detection are not able to improve on that result. However, we observe an interesting boost in the training performance from the compressible representation when using both class and detection informed compression. This indicates that adding multiple classes information to the latent compressible format may improve generalization to other tasks.

We see a similar picture for bpp 0.4, however, in this case there is a more substantial drop in performance when using the decompressed format at a bit rate of 0.4. The naive learned compression already was able to greatly improve on the results at this bit rate, this is similar to what was observed in [52]. Our detection information compression, however, was able to slightly improve on this number.

This architecture does not deliver good performance, this is why we proposed the Subpixel architecture.

4.5.2 Subpixel Resnet

The major difference between the Truncated Resnet and the SubPixel Resnet is the Stem. We did some test to find the best stem we could. We tested if the 4X upsampling was better done in one step or in two 2X upsampling steps interleaved with a few convolution layers. We can clearly see in the table 4.3 that the two smaller upsampling steps greatly outperform the single big step. We also tested if replacing convolution with a kernel size of 7, like in the original ResNet, by 3 convolutions with kernel size of 3 was benefiting. We also tested replacing the

Relu activation function in the stem by the SiLU [72], defined as $f : x \to x\sigma(x)$, and the Mish, defined as $f : x \to x\tanh(softplus(x))$ [73] activation function. These functions are very similar to ReLU, but they are smooth. They have been shown to outperform ReLU in a variety of cases. This is also the case with the proposed stem. After this test we decided to use the stem presented in section 3.3.1. The metric that was used to compare the different architecture choice of the stem was the Top-1 accuracy of the classification task. We chose this metric because the classification weights are used to initialize the other tasks network. We conjectured that better performance on the classification task would lead to better transfer learning, but we did not investigate that claim any further.

One upsample step	Two upsample steps	Residual Block	Mish	SiLU	Top 1
✓					69.2
✓		✓			71.2
	✓	✓			74.8
	✓	✓	✓		75.0
	✓	✓		✓	75.1

Table 4.3 SubPixel Stems on the classification task

When we proposed this architecture, the detection bug was corrected so we included the detection results in the table 4.4.

	Input	Segmentation mIoU	Classification Top 1	Classification Top 5	Detection mAP	Detection AP50
	RGB from standard 4 to 5 bpp JPGs	70.8	77.3	93.3	53.1	80.8
bpp ~ 1.0	RGB - compressed JPEG at 1.0 bpp	67.5	75.0	92.2	52.3	79.8
	Naive Learned Compression	67.5	74.8	92.2	52.8	80.3
	Class Informed Compression	69.5	**75.1**	**92.4**	54.7	81.1
	Detection Informed Compression	70.5	74.9	92.1	54.4	81.4
	Class & Detection Informed Compression	**71.3**	**75.1**	92.2	**54.6**	**81.6**
bpp ~ 0.4	RGB - compressed JPEG at 0.4 bpp	57.8	70.7	89.8	51.4	78.1
	Naive Learned Compression	68.9	74.0	91.7	52.1	79.6
	Class Informed Compression	69.0	**75.1**	**92.3**	52.8	80.1
	Detection Informed Compression	**70.2**	74.8	92.1	**52.9**	**80.3**

Table 4.4 Performance comparison on the downstream tasks when going from different compressible representation using the SubPixel-Resnet as the downstream backbone architecture.

In the 1.0 bpp case, we can see that all the classification results are very close to the 75.0 top-1 accuracy which the performance when going from the compressed JPEG. The value is still lower than when going from the image. This means that our compressor does not offer a

clear advantage at this bit rate for this task. On the contrary, we can see that the semantic segmentation obtains a 71.3 mIoU which is 3.8 points over the base case of 67.5 mIoU. It even surpasses the performance when going from the original image. We see a similar result in the detection case where the best performance of 54.6 AP is 2.3 points then the base case of 53.1 AP and also better than when going from the image. Both these tasks depend on dense labels in the input image. This increase in performance might be due to the fact that the dense pixel-level information in concentrated in a smaller number of elements to look at, making it easier to perform.

For the 0.4 bpp case, all the learned compressor completely outclasses the JPEG compressor at a similar bit rate. We can see a 12.4 mIoU uplift in semantic segmentation, 4.4 uplift in top1 accuracy in classification and 1.5 point in AP detection. This means that in a use case where available bandwidth is extremely limited, like in IoT devices, our learned compressor offers a significant advantage.

We also note that jointly training one a semantic task can help another. As a matter of fact, we can see in the table 4.4 that for both compression ratio, jointly training on the detection task helps the semantic segmentation task. This proves that they learn a representation that is more semantically structured, even for tasks that were **NOT** seen in training.

Few-Shot Detection

Finally, we present the few shot detection results in the table 4.5. We did not include them in the table 4.4, because they would have made the table too big. When lower bpp JPEG images

Method	Bits-Per Pixel	Novel Set 1					Novel Set 2					Novel Set 3				
		1	2	3	5	10	1	2	3	5	10	1	2	3	5	10
YOLO-joint [74]	4 to 5	0.0	0.0	1.8	1.8	1.8	0.0	0.1	0.0	1.8	0.0	1.8	1.8	1.8	3.6	3.9
MetaDet [75]	4 to 5	18.9	20.6	30.2	36.8	49.6	21.8	23.1	27.8	31.7	43.0	20.6	23.9	29.4	43.9	44.1
Meta R-CNN [76]	4 to 5	19.9	25.5	35.0	45.7	51.5	10.4	19.4	29.6	34.8	**45.4**	14.3	18.2	27.5	41.2	48.1
TFA [5]	4 to 5	39.8	36.1	44.7	55.7	56.0	23.5	26.9	34.1	35.1	39.1	30.8	34.8	42.8	49.5	49.8
TFA (Ours)	4 to 5	36.7	28.9	43.2	55.1	56.2	18.0	28.6	33.2	35.1	39.2	27.0	33.2	42.6	48.1	49.5
TFA, very-low bit-rate JPGs	0.4	34.2	25.6	41.8	53.1	53.7	14.9	27.0	30.5	32.8	35.5	23.8	30.1	38.2	44.3	46.3
Ours, Naive Compression	0.4	28.5	21.2	37.0	48.9	50.3	15.7	19.8	25.4	27.8	32.5	20.9	25.3	33.2	40.6	41.0
Ours, Class Informed	0.4	34.9	32.4	41.4	50.6	53.0	18.7	22.8	28.9	30.7	34.2	25.1	29.7	37.9	44.5	44.6
Ours, Detection Informed	0.4	34.3	31.4	40.2	49.3	51.7	17.9	21.7	28.8	29.8	34.1	24.5	29.2	37.3	43.8	44.1
TFA low bit-rate JPGs	1.0	35.6	27.4	42.8	54.2	55.6	16.3	28.0	31.9	33.9	37.8	25.0	31.5	40.6	47.0	48.4
Ours, Naive Compression	1.0	33.5	26.3	40.7	53.5	55.1	20.2	24.1	30.5	31.1	36.8	26.1	30.5	38.7	45.6	45.9
Ours, Class Informed	1.0	38.8	37.2	44.1	55.3	56.7	23.7	27.2	34.5	35.3	39.2	31.2	34.9	42.5	49.4	49.5
Ours, Detection Informed	1.0	39.3	**37.5**	**44.8**	**56.3**	**57.1**	**23.7**	**29.1**	**34.8**	**35.3**	**39.2**	**31.3**	**34.9**	**42.9**	**49.5**	**49.9**
Ours, Class. + Det. Informed	1.0	38.1	37.3	44.7	56.2	57.1	23.6	27.3	34.0	35.1	38.9	31.0	33.9	42.1	48.7	48.9

Table 4.5 Few-shot detection peformance (mAP50) on PASCAL VOC datasets.

are use to train the original few-shot detector (TFA), the performance drops by approximately 1 mAP50 at 1 bpp and 2 mAP50 at 0.4 bpp. Our compression format without semantic training performs lower then TFA from JPEG using similar compression ratio. However, in

sharp contrast, we show that training the few-shot detector from our classification informed compressible representation not just improves overall few-shot performance at 1 bpp, but achieves state-of-the-art results for many of the experimental configurations. This strongly suggests that when jointly trained with tasks, compressible representations can serve as a regularizer in low-data regimes.

4.5.3 Downstream Task Backbone Comparison

In this section we compare the performance of the two backbones we proposed with the results presented in [52], which is the only other work that is comparable that we know of. This work tested its performance on the classification and segmentation task on the same dataset as the one we used in this thesis. We first present the case where the compressible representation is not fine-tuned on the downstream tasks in the table 4.6. We can see that at a lower compression rate, our representation is much stronger than the one used in [52]. In fact, even our weaker Truncated-Resnet shows superior performance.

Input	Segmentation mIoU	Classification Top 1	Top 5
RGB (5 bpp)	**70.8**	**77.3**	**93.3**
cResnet (0.635 bpp) [52]	62.9	67.7	87.9
Truncated-Resnet (0.4 bpp)	64.4	70.3	89.8
Truncated-Resnet (1.0 bpp)	68.7	69.8	89.2
SubPixel-Resnet (0.4 bpp)	68.9	74.0	91.7
SubPixel-Resnet (1.0 bpp)	67.5	75.0	92.2

Table 4.6 Performance comparison on the downstream tasks when the representation is **not trained jointly** with the downstream tasks at low bit-per-pixels

The second case we want to compare is when the encoder is trained jointly with the downstream tasks in the table 4.7. Again we can see that our models are much stronger than the one presented in [52] especially in the semantic segmentation task.

This means that at the moment of writing, our proposed solution is state of the art for vision tasks when going from a compressible representation.

We also want to compare the speed of the different backbones. We compare the inference speed when going from images and when going from the compressible representation in the table 4.8. For Truncated-Resnet and the SubPixel-Resnet we only count the time once the compressible representation is obtained. We skip the encoding time. We do this because,

Input	Segmentation mIoU	Classification Top 1	Top 5
RGB (5 bpp)	70.8	**77.3**	**93.3**
cResnet (0.635 bpp) [52]	64.0	-	88.1
Truncated-Resnet (0.4 bpp)	64.8	72.6	90.8
Truncated-Resnet (1.0 bpp)	68.7	74.8	91.9
SubPixel-Resnet (0.4 bpp)	70.2	75.1	92.3
SubPixel-Resnet (1.0 bpp)	**71.3**	75.1	92.4

Table 4.7 Performance comparison on the downstream tasks when the representation is **trained jointly** with the downstream tasks at low bit-per-pixels

if this method were to be used, the file being used is the compressible representation and not the image, the encoding step has already done. We note that the Truncated-Resnet is on average 20% faster than the original and the Subpixel Resnet is about on par if not a little bit slower. This means that the two proposed models offer a compromise in speed and performance.

Task	Frame-per-second Resnet	Truncated Resnet	SubPixel Resnet
Classification	1716.8	2097.3	1710.3
Semantic Segmentation	73.7	102.0	72.6
Detection	35.7	39.5	34.9

Table 4.8 Computational performance comparison between the forward pass of an RGB image and a compressed latent input.

CHAPTER 5 CONCLUSION

With the growing prominence of computer vision applications, practitioners should consider compression formats that not only enable high-fidelity reconstructions, but can also be used directly for inference for semantic tasks.

5.1 Summary of Works

In this work, we presented the interesting and important field of machine learning research: learned image compression. We presented a simple way of learning an image compressor while taking into account computer vision tasks. The compression far exceeded existing image codecs such as JPEG. We also showed that the computer vision tasks can perform just as well if not better if we present them a semantically structured compressible representation of the images as input. To accomplish this we proposed two different backbone architectures that are used for either speed or raw performance. We also presented a novel way of integrating the vision tasks information into in the learned compressor. We also showed very good results in the low-shot setting when taking the compressible representation as input.

5.2 Limitations

This work was limited by a few factors, namely the compression performance is not quite on par with the state of the art compression methods. This is mainly due to the high degree of instability of our image compression pipeline. This method does not present a way to incorporate a new vision tasks into the training pipeline easily, because the information of the different tasks can interfere with one another.

5.3 Future Research

It would be interesting to see how much better results we could obtain if we used a better architecture. We can note that the encoder and the decoder are very shallow by modern computer vision architecture standards. This is due to the fact that there has not been great success so far with very deep architecture in image compression. In fact, a literature review of modern end-to-end lossy image compression showed that all modern techniques that achieved some level of success used shallow networks for encoding and decoding the image [77]. Developing a better architecture for image compression while using an hyperprior

network is an interesting area of research, that is not touched in this work. More recent work adds an autoregressive component to the hyper-decoder network and obtain slightly higher performance [44]. We used a constant network width at each stage of the network, which is a simple but surely suboptimal choice. Interesting ideas would be to add attention, skip connection, rectified sub-pixel convolutions, mixture of experts or even neural architecture search.

This research project was slowed by the difficulty of training in a multi-task setting, so an interresting way to further the field would be a way to lower the instability. One possible avenue is the use of Meta-Learning for Continual Learning. The training method called La-MAML [78] produces per-parameter learning rate to insure that when a network is traininng on new tasks, in our case downstream tasks, is doesn't not forget about old tasks, in our case compression. It could be a staring block for this.

We conjecture that the combined effects of compact codes provided by compression and the multi-task learning setup, produces representations that are particularly effective in the low shot learning regime. Then we suggest to further investigate this connection. This could be very useful in context of robotics, where samples are expensive to acquire, to enable user penalization.

REFERENCES

[1] I. Higgins *et al.*, "beta-vae: Learning basic visual concepts with a constrained variational framework." *Iclr*, vol. 2, no. 5, p. 6, 2017.

[2] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Advances in neural information processing systems*, 2017, pp. 4077–4087.

[3] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," *arXiv preprint arXiv:1703.03400*, 2017.

[4] Z. Cheng *et al.*, "Learning image and video compression through spatial-temporal energy compaction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 071–10 080.

[5] X. Wang *et al.*, "Frustratingly simple few-shot object detection," *arXiv preprint arXiv:2003.06957*, 2020.

[6] T.-Y. Lin *et al.*, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117–2125.

[7] L.-C. Chen *et al.*, "Rethinking atrous convolution for semantic image segmentation," *arXiv preprint arXiv:1706.05587*, 2017.

[8] H. J. KELLEY, "Gradient Theory of Optimal Flight Paths," *ARS Journal*, vol. 30, no. 10, pp. 947–954, 1960. [Online]. Available: http://arc.aiaa.org

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: https://www.nature.com/articles/323533a0

[10] T. M. Cover, *Elements of information theory*. John Wiley & Sons, 1999.

[11] G. E. Box, "Science and statistics," *Journal of the American Statistical Association*, vol. 71, no. 356, pp. 791–799, 1976.

[12] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[14] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[15] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[16] A. Pinkus, "Approximation theory of the mlp model in neural networks," *Acta numerica*, vol. 8, no. 1, pp. 143–195, 1999.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[18] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[19] H. Touvron *et al.*, "Fixing the train-test resolution discrepancy: Fixefficientnet," *arXiv preprint arXiv:2003.08237*, 2020.

[20] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *Lecture Notes in Computer Science*, p. 818–833, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10590-1_53

[21] K. He *et al.*, "Deep residual learning for image recognition," 2015.

[22] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[23] Y. Gong *et al.*, "Learning compositional visual concepts with mutual consistency," 2018.

[24] N. Tishby, F. C. Pereira, and W. Bialek, "The information bottleneck method," *arXiv preprint physics/0004057*, 2000.

[25] A. M. Saxe *et al.*, "On the information bottleneck theory of deep learning," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2019, no. 12, p. 124020, 2019.

[26] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[27] G. K. Wallace, "The jpeg still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[28] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The jpeg 2000 still image compression standard," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 36–58, 2001.

[29] V. K. Goyal, "Theoretical foundations of transform coding," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 9–21, 2001.

[30] G. W. Cottrell, "Image compression by back propagation: an example of extensional programming," 1988.

[31] J. Jiang, "Image compression with neural networks–a survey," *Signal processing: image Communication*, vol. 14, no. 9, pp. 737–760, 1999.

[32] G. Toderici *et al.*, "Full resolution image compression with recurrent neural networks," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 5435–5443, 2017.

[33] N. Johnston *et al.*, "Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4385–4393.

[34] D. Minnen *et al.*, "Spatially adaptive image compression using a tiled deep network," in *2017 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2017, pp. 2796–2800.

[35] L. Theis *et al.*, "Lossy image compression with compressive autoencoders," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, pp. 1–19, 2019.

[36] T. Dumas, A. Roumy, and C. Guillemot, "Image compression with Stochastic Winner-Take-All Auto-Encoder," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2017, pp. 1512–1516. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01493137

[37] F. Mentzer *et al.*, "Conditional probability models for deep image compression," 2019.

[38] E. Agustsson *et al.*, "Soft-to-hard vector quantization for end-to-end learning compressible representations," *Advances in Neural Information Processing Systems*, vol. 2017-December, pp. 1142–1152, 2017.

[39] J. Ballé, V. Laparra, and E. P. Simoncelli, "End-to-end optimized image compression," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017. [Online]. Available: http://arxiv.org/abs/1611.01704

[40] ——, "Density modeling of images using a generalized normalization transformation," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pp. 1–14, 2016. [Online]. Available: http://arxiv.org/abs/1511.06281

[41] M. H. Baig, V. Koltun, and L. Torresani, "Learning to inpaint for image compression," *arXiv preprint arXiv:1709.08855*, 2017.

[42] O. Rippel and L. Bourdev, "Real-time adaptive image compression," in *International Conference on Machine Learning*. PMLR, 2017, pp. 2922–2930.

[43] J. Ballé *et al.*, "Variational image compression with a scale hyperprior," *arXiv preprint 1802.01436*, 2018.

[44] D. Minnen, J. Ballé, and G. Toderici, "Joint autoregressive and hierarchical priors for learned image compression," *Advances in Neural Information Processing Systems*, vol. 2018-December, no. Nips, pp. 10 771–10 780, 2018.

[45] M. Tschannen, E. Agustsson, and M. Lucic, "Deep generative models for distribution-preserving lossy compression," *arXiv preprint arXiv:1805.11057*, 2018.

[46] A. Vaswani *et al.*, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[47] Z. Cheng *et al.*, "Learned lossless image compression with a hyperprior and discretized gaussian mixture likelihoods," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 2158–2162.

[48] D. Fu and G. Guimaraes, "Using compression to speed up image classification in artificial neural networks," 2016.

[49] S. Singh *et al.*, "End-to-end learning of compressible features," in *ICIP*, 2020.

[50] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[51] J. Masci *et al.*, "Stacked convolutional auto-encoders for hierarchical feature extraction," in *International conference on artificial neural networks*. Springer, 2011, pp. 52–59.

[52] R. Torfason *et al.*, "Towards image understanding from deep compression without decoding," *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pp. 1–17, 2018. [Online]. Available: http://arxiv.org/abs/1803.06131

[53] G. Lu *et al.*, "Dvc: An end-to-end deep video compression framework," 2019.

[54] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE transactions on Computers*, vol. 100, no. 1, pp. 90–93, 1974.

[55] L. Gueguen *et al.*, "Faster neural networks straight from jpeg," in *Advances in Neural Information Processing Systems*, 2018, pp. 3933–3944.

[56] J. Ballé *et al.*, "Variational image compression with a scale hyperprior," *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. [Online]. Available: http://arxiv.org/abs/1802.01436

[57] M. Carandini and D. J. Heeger, "Normalization as a canonical neural computation," *Nature Reviews Neuroscience*, vol. 13, no. 1, pp. 51–62, 2012.

[58] N. Johnston *et al.*, "Computationally efficient neural image compression," 2019.

[59] W. Shi *et al.*, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," 2016.

[60] N. Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[61] S. Ren *et al.*, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016.

[62] T. Lindeberg, "Scale-space for discrete signals," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 3, pp. 234–254, 1990.

[63] T.-J. Yang *et al.*, "Deeperlab: Single-shot image parser," 2019.

[64] Y. Wu *et al.*, "Detectron2," https://github.com/facebookresearch/detectron2, 2019.

[65] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[66] B. J. H. S. J. N. M. D. A. E., "Tensorflow compression," https://github.com/charlespwd/project-title, 2018.

[67] E. Kodak, "Kodak lossless true color image suite (photocd pcd0992)," *URL http://r0k.us/graphics/kodak*, 1993.

[68] A. Kuznetsova *et al.*, "The open images dataset v4," *International Journal of Computer Vision*, vol. 128, no. 7, p. 1956–1981, Mar 2020. [Online]. Available: http://dx.doi.org/10.1007/s11263-020-01316-z

[69] M. Everingham *et al.*, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[70] B. Hariharan *et al.*, "Semantic contours from inverse detectors," in *International Conference on Computer Vision (ICCV)*, 2011.

[71] P. Goyal *et al.*, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[72] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," 2020.

[73] D. Misra, "Mish: A self regularized non-monotonic activation function," 2020.

[74] B. Kang *et al.*, "Few-shot object detection via feature reweighting," 2019.

[75] Y.-X. Wang, D. Ramanan, and M. Hebert, "Meta-learning to detect rare objects," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 9925–9934.

[76] X. Yan *et al.*, "Meta r-cnn: Towards general solver for instance-level low-shot learning," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 9577–9586.

[77] Y. Hu *et al.*, "Learning end-to-end lossy image compression: A benchmark," *arXiv preprint arXiv:2002.03711*, 2020.

[78] G. Gupta, K. Yadav, and L. Paull, "La-maml: Look-ahead meta learning for continual learning," 2020.

APPENDIX A DERIVATIONS

A.1 ELBO

Derivation Jensen's inequality for a convex function: $f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)]$, the inverse is true if f is a concave. log is concave.

$$\begin{aligned}
\log p(x) &= \log \sum_z p(x,z) \\
&= \log \sum_z q_\phi(z) \frac{p(x,z)}{q_\phi(z)} \\
&= \log \left(\mathbb{E}_q \left[\frac{p(x,z)}{q_\phi(z)} \right] \right) \\
&\geq \mathbb{E}_q \left[\log \left(\frac{p(x,z)}{q_\phi(z)} \right) \right] \\
&= \sum_z q_\phi(z) \log \left(\frac{p(x,z)}{q_\phi(z)} \right) \\
&= \sum_z q_\phi(z) \log p(x,z) - \sum_z q_\phi(z) \log q_\phi(z) \\
&= \mathbb{E}_q[\log p(x,z)] + H(q_\phi) \\
&= \mathcal{L}(q)
\end{aligned}$$

Which defines the ELBO and proves that $\log p(x) \geq \mathcal{L}(q) = H(q_\phi) - H(q_\phi, p(x,z))$

Reformulation

$$\begin{aligned}
KL\left[q_\phi \parallel p_{Z|X}\right] &= \sum_z q(z) \log \frac{q(z)}{p(z|x)} \\
KL\left[q_\phi \parallel p_{Z|X}\right] &= \sum_z q(z) \log \frac{q(z)p(x)}{p(z,x)} \\
KL\left[q_\phi \parallel p_{Z|X}\right] &= \sum_z q(z) \log q(z) - \sum_z q(z) \log p(z,x) + \sum_z q(z) \log p(x) \\
KL\left[q_\phi \parallel p_{Z|X}\right] &= \sum_z q(z) \log q(z) - \sum_z q(z) \log p(z,x) + \log p(x) \\
\log p(x) - KL\left[q_\phi \parallel p_{Z|X}\right] &= \sum_z q(z) \log p(z,x) - \sum_z q(z) \log q(z) \\
\log p(x) - KL\left[q_\phi \parallel p_{Z|X}\right] &= \mathcal{L}(x)
\end{aligned}$$

A.2 VAE objective Function

The ELBO can also be reformulated in the following way:

$$\begin{aligned}
\mathcal{L}(q) &= H(q) - H(q, p(x,z)) \\
&= -\mathbb{E}_q[\log q(z|x)] + \mathbb{E}_q[\log p(z,x)] \\
&= -\mathbb{E}_q[\log q(z|x)] + \mathbb{E}_q[\log p(z|x)p(x)] \\
&= \log p(x) - \mathbb{E}_q[\log q(z|x) - \log p(z|x)] \\
&= \log p(x) - \mathbb{E}_q\left[\log q(z|x) - \log \frac{p(x|z)p(z)}{p(x)}\right] \\
&= -\mathbb{E}_q[\log q(z|x) - \log p(x|z) - \log p(z)] \\
&= \mathbb{E}_q[\log p(x|z)] - \mathbb{E}_q\left[\log \frac{q(z|x)}{p(z)}\right] \\
&= \mathbb{E}_q[\log p(x|z)] - KL[q(z|x) \parallel p(z)] \qquad \text{(A.1)}
\end{aligned}$$

Reconstruction loss If we pose the hypothesis that $p_\theta(x|z)$ is the Gaussian distribution $\mathcal{N}(x, I)$

$$\begin{aligned}
\log p_\theta(x|z) &= \log\left(\frac{1}{\sqrt{(2\pi)^d|\Sigma|}}\exp\left(\frac{-1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)\right) \\
&= \log\left(\frac{1}{\sqrt{(2\pi)^d}}\exp\left(\frac{-1}{2}(\hat{x}-x)^T(\hat{x}-x)\right)\right) \\
&= -\frac{nd}{2}\log(2\pi) - \frac{1}{2}||\hat{x}-x||^2
\end{aligned}$$

Because $-\frac{nd}{2}\log(2\pi)$ and $-\frac{1}{2}$ are constants, we can conclude that

$$\max -\log p_\theta(x|z) = \min ||\hat{x}-x||^2 + cst$$

This means that $\min ||\hat{x}-x||^2$ can be used as the objective.

KL The KL divergence between Gaussian functions in dimensions k is the following :

$$KL(\mathcal{N}(\mu_0,\Sigma_0) \parallel \mathcal{N}(\mu_1,\Sigma_1)) = \frac{1}{2}\left(\mathrm{Tr}(\Sigma_1^{-1}\Sigma_0) + (\mu_1-\mu_0)^T\Sigma_1^{-1}(\mu_1-\mu_0) - k + \log\frac{|\Sigma_1|}{|\Sigma_0|}\right)$$

If $\mathcal{N}(\mu_0, \Sigma_0) = \mathcal{N}(\mu, \Sigma)$ and $\mathcal{N}(\mu_1, \Sigma_1) = \mathcal{N}(0, I)$ then

$$KL(\mathcal{N}(\mu, \Sigma) \parallel \mathcal{N}(0, I)) = \frac{1}{2}(\mathrm{Tr}(\Sigma) + \mu^T\mu - k - \log|\Sigma|) \tag{A.2}$$

If we impose that $\Sigma = \mathrm{diag}(\sigma_0, \sigma_1, \ldots, \sigma_k)$ then

$$KL(\mathcal{N}(\mu, \Sigma) \parallel \mathcal{N}(0, I)) = \frac{1}{2}(\sum_{i=1}^{k}\sigma_i + \mu^T\mu - k - \log\prod_{i=1}^{k}\sigma_i) = \frac{1}{2}\sum_{i=1}^{k}\sigma_i + \mu_i^2 - 1 - \log\sigma_i \tag{A.3}$$

Reparametrization trick The expression we want we want to compute is $\nabla_\theta \mathbb{E}_{p_\theta}[f_\theta(Z)]$. The expectation is distributional because p_θ is parametrized by θ and structural because f_θ is also paramtetrized by θ. This is

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{p_\theta}[f_\theta(Z)] &= \nabla_\theta \int_z p_\theta(z) f_\theta(z) dz \\
&= \int_z \nabla_\theta [p_\theta(z) f_\theta(z)] dz \\
&= \int_z f_\theta(z) \nabla_\theta p_\theta(z) + p_\theta(z) \nabla_\theta f_\theta(z) dz
\end{aligned} \tag{A.4}$$

This can be reformuated as the following equation

$$\int_z f_\theta(z) \nabla_\theta p_\theta(z) dz + \mathbb{E}_{p_\theta}[\nabla_\theta f_\theta(Z)] \tag{A.5}$$

In this formulation, the expectation can be appromixatly computed by by a Monte Carlo Simulation, but integral cannot because neither $f_\theta(z)$ or $\nabla_\theta p_\theta(z)$ are density functions. To alleviate this, we use the reparametrization trick.

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{p_\theta}[f_\theta(Z)] &= \int_z f_\theta(z) \nabla_\theta p_\theta(z) + p_\theta(z) \nabla_\theta f_\theta(z) dz \\
&= \int_z [f_\theta(z) \nabla_\theta p_\theta(z) + p_\theta(z) \nabla_\theta f_\theta(z)] \frac{q(z)}{q(z)} dz \\
&= \int_z \left[f_\theta(z) \nabla_\theta \frac{p_\theta(z)}{q(z)} + \frac{p_\theta(z)}{q(z)} \nabla_\theta f_\theta(z) \right] q(z) dz \\
&= \mathbb{E}_q \left[f_\theta(Z) \nabla_\theta \frac{p_\theta(Z)}{q(Z)} + \frac{p_\theta(Z)}{q(Z)} \nabla_\theta f_\theta(Z) \right]
\end{aligned}$$

Choose $p_\theta(z) = q(z)$ just like in the score function in RL

$$\nabla_\theta \mathbb{E}_{p_\theta}[f_\theta(Z)] = \mathbb{E}_q \left[f_\theta(Z) \nabla_\theta \frac{p_\theta(Z)}{q(Z)} + \frac{p_\theta(Z)}{q(Z)} \nabla_\theta f_\theta(Z) \right] \tag{A.6}$$

$$= \mathbb{E}_q[f_\theta(Z) \nabla_\theta 1 + \nabla_\theta f_\theta(Z)] \tag{A.7}$$

$$= \mathbb{E}_q[\nabla_\theta f_\theta(Z)] \tag{A.8}$$

Then change of variable of probability density function where $Z = h(Y, x)$, Y is the new random variable and x is a deterministic input.

$$\nabla_\theta \mathbb{E}_{p_\theta}[f_\theta(Z)] = \mathbb{E}_q[\nabla_\theta f_\theta(Z)] = \mathbb{E}_{p(Y)}[\nabla_\theta f_\theta(h(Y, x))] \tag{A.9}$$

A.3 Hyper-prior

Noise convolution

$$\begin{aligned} p_{\hat{z}}(\hat{z}) &= \left(p * \mathcal{U}(-\frac{1}{2}, \frac{1}{2}) \right)(\hat{z}) \\ &= \int_{-\infty}^{\infty} p(z) \mathbb{1}\left[\hat{z} - \frac{1}{2}, \hat{z} + \frac{1}{2} \right] dz \\ &= \int_{\hat{z}-\frac{1}{2}}^{\hat{z}-\frac{1}{2}} p(z) dz \\ &= F\left(\hat{z} + \frac{1}{2} \right) - F\left(\hat{z} - \frac{1}{2} \right) \end{aligned}$$

where F is the cumulative function of p

KL decomposition

$$KL\left[q \parallel p_{z,h|x} \right] = \mathbb{E}_q \left[\log \frac{q(\hat{z}, \hat{h})}{p_(z, h|x)} \right] \tag{A.10}$$

$$= \mathbb{E}_q[\underbrace{\log q(\hat{z}, \hat{h})}_{=0} - \log p(\hat{z}, \hat{h}|x)] \tag{A.11}$$

$$= \mathbb{E}_q[-\log \frac{p(\hat{z}, \hat{h}, x)}{p(x)}] \tag{A.12}$$

$$= \underbrace{\mathbb{E}_q[-\log p(x|\hat{z}, \hat{h})]}_{\text{Distortion}} + \underbrace{E_q[-\log p(\hat{z}|\hat{h}) - \log p(\hat{h})]}_{\text{Rate}} + \underbrace{\log p(x)}_{\text{Constant}} \tag{A.13}$$

Reason : $\mathbb{E}_q[\log q(\hat{z}, \hat{h})] = 0$ because $q(\hat{z}, \hat{h})$ is always 0 or 1 and $\lim_{x \to 0^+} x \log x = 0$. $\log p(x)$ is a constant so it can be ignored in the minization problem.

Rate decomposition

$$KL\left[q \parallel p_{z,h}\right] = \mathbb{E}_q\left[\log \frac{q(\hat{z},\hat{h})}{p(z,h)}\right] \tag{A.14}$$

$$= \mathbb{E}_q[\log q(\hat{z},\hat{h}) - \log p(z|h) - \log p(h)] \tag{A.15}$$

$$= \mathbb{E}_q[-\log p(z|h) - \log p(h)] \tag{A.16}$$