ABSTRACT

Title of thesis: DEEP NEURAL NETWORKS FOR END-TO-END OPTIMIZED SPEECH CODING

Srihari Kankanahalli, Master of Science, 2017

Thesis directed by: Dr. David Jacobs
Department of Computer Science

Modern compression algorithms are the result of years of research; industry standards such as MP3, JPEG, and G.722.1 required complex hand-engineered compression pipelines, often with much manual tuning involved on the part of the engineers who created them. Recently, deep neural networks have shown a sophisticated ability to learn directly from data, achieving incredible success over traditional hand-engineered features in many areas. Our aim is to extend these "deep learning" methods into the domain of compression.

We present a novel deep neural network model and train it to optimize all the steps of a wideband speech-coding pipeline (compression, quantization, entropy coding, and decompression) end-to-end directly from raw speech data, no manual feature engineering necessary. In testing, our learned speech coder performs on par with or better than current standards at a variety of bitrates (∼9kbps up to ∼24kbps). It also runs in realtime on an Intel i7-4790K CPU.

DEEP NEURAL NETWORKS FOR END-TO-END OPTIMIZED SPEECH CODING

by

Srihari Kankanahalli

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2017

Advisory Committee:
Dr. David Jacobs, Chair
Dr. Ramani Duraiswami
Dr. Carol Espy-Wilson
Dr. Shihab Shamma

Table of Contents

List of Figures iv

List of Abbreviations v

1 Introduction 1
1.1 Context and Motivation . . . 1
1.2 Structure of This Thesis . . . 3

2 Artificial Neural Networks 4
2.1 Overview . . . 4
2.2 Basic Artificial Neurons . . . 7
2.3 Training Neural Networks . . . 9
2.4 Fully-Connected Neural Networks . . . 14
2.5 Convolutional Neural Networks . . . 17
2.6 Residual Neural Networks . . . 20

3 Related Work 24
3.1 Current Speech Coding Standards . . . 24
3.2 A Brief History of ANN-Based Compression . . . 27
3.2.1 Limitations of Prior Work . . . 30
3.2.2 Recent Developments . . . 32

4 Compressive Residual Autoencoders 33
4.1 Architecture and Design . . . 33
4.1.1 Residual Block Types . . . 35
4.1.2 Softmax Quantization . . . 38
4.2 Training Methodology . . . 42
4.2.1 Objective Function . . . 43
4.2.2 Training Process . . . 46
4.2.3 Hyperparameters . . . 47
4.3 Results . . . 48
4.3.1 Objective Quality Evaluation . . . 49
4.3.2 Complexity Evaluation . . . 51

4.3.3 Subjective Quality Evaluation 52
4.3.4 Ablation Study 53

5 Conclusion 54

Bibliography 56

List of Figures

2.1 An example neural network. Taken from [6]. 5
2.2 A simple artificial neuron, with 3 inputs. 7
2.3 Four common activation functions, along with their derivatives. . . . 14
2.4 A small fully-connected neural network for MNIST. 15
2.5 Illustration of a convolution operation. 18
2.6 A small convolutional neural network for MNIST. 19
2.7 A simplified version of Figure 2.6. 20
2.8 A basic residual block, with the shortcut connection in green. 21
2.9 A small residual network for MNIST. 22

3.1 A block diagram of the AMR-WB encoder. Taken from [28]. 26

4.1 Simplified network architecture. 33
4.2 Our full compression pipeline. 35
4.3 The four block types used in our network architecture. 36
4.4 5-step uniform quantizer and its derivative. 38
4.5 Softmax quantization, for both a low and high temperature. 40
4.6 The effect of temperature on our softmax quantizer. 41
4.7 Triangular Mel filterbank with 8 filters. 44
4.8 Mean PESQ of our speech coder, compared with AMR-WB. 50

List of Abbreviations

AMR-WB	Adaptive Multi-Rate Wideband
ANN	artificial neural network
ACELP	algebraic code-excited linear prediction
CNN	convolutional neural network
DCR	Degradation Category Rating
DNN	deep neural network
kbps	kilobits per second
MOS	Mean Opinion Score
MSE	mean-squared error
PESQ	Perceptual Evaluation of Speech Quality
ReLU	rectified linear unit
RNN	recurrent neural network
tanh	hyperbolic tangent function
ITU-T	International Telecommunication Union Telecommunication Standardization Sector

Chapter 1: Introduction

1.1 Context and Motivation

The everyday applications of data compression are ubiquitous: streaming live videos and music in realtime across the planet, storing thousands of images and songs on a single small thumb drive, transferring large amounts of data across satellite channels, and more. In a way, improved compression algorithms were what made these innovations even possible in the first place, and designing better and more efficient compression could help expand them even further (to developing nations with slower Internet speeds, for example, or for much larger transmission distances).

All of the current state-of-the-art standards in data compression are, essentially, hand-designed by researchers. Modern compression algorithms usually require months or years of development, testing, and standardization, involving much effort and manual tuning on the part of the engineers who create them. For example, eight years passed between the release of the original JPEG standard and the finalization of its successor, JPEG2000; similarly, the MP3 standard was a joint effort supported by researchers around the world, which took four years to fully develop (from the MUSICAM project in 1989 to the first MPEG specifications in 1993).

Recently, deep neural networks have shown an incredible ability to learn di-

rectly from data, circumventing traditional feature engineering. This has led to state-of-the-art results in a variety of areas, from speech recognition to computer vision to image generation [1]. These "deep learning" models are not designed to directly perform a task, but rather to *learn* how to perform a task, given a large enough set of data. This naturally leads us to ask: can we use deep networks to *learn* a compression algorithm on par with modern standards automatically from data, obviating the need for hand-designed features and the labor of human engineers?

The specific domain we choose to explore in this thesis is *speech coding*, the compression of speech. Important applications of speech coding include mobile telephony, videoconferencing, and voice over IP (VoIP); all of these mostly rely on *lossy* compression of speech (where the speech signal doesn't need to be transmitted exactly, and some distortion is allowed). Our specific research question then becomes: how can we learn a general, reliable speech coder end-to-end purely from raw speech data, with as little human "intervention" or manual feature engineering as possible?

This end-to-end approach is surprisingly rare in the literature. To the best of our knowledge, this is only the second ever work to learn an audio compression pipeline end-to-end (the previous being an obscure early attempt by Morishima et al. in 1990 [2]), and the first to produce state-of-the-art results. The closest work to ours is that of Cernak et al. in 2016 [3], who created an almost-end-to-end design for a very-low-bitrate low-quality speech coder; however, their pipeline still required hand-designed extraction of acoustic features and pitch at the beginning (and additionally was quite complex, combining many deep and spiking neural networks together). All other related designs we know of employ neural networks as a component of a larger

hand-designed system (e.g. as the predictor in a linear predictive coding system, or as a vector quantizer).

In the domain of image compression, there has been a bit more interest in training end-to-end ANN-based systems since the early 1990s [4], but this paradigm has not yielded state-of-the-art results until fairly recently either (starting August 2016, when Toderici et al. proposed a neural network architecture outperforming JPEG [5]). Thus, we are happy to report that our work seems to be on the cutting edge of both deep learning research and compression research.

We now describe the structure of this thesis.

1.2 Structure of This Thesis

Chapter 2 provides an overview of artificial neural networks and their foundations, including some recent advances which made this research possible. Chapter 3 goes over previous work (in the areas of both machine learning and speech coding) as well as its limitations. Chapter 4 describes the neural network model we built in depth, details how we tested and evaluated it, and gives objective and subjective measures of our speech coder's quality at various bitrates (comparing our approach to current standards). Finally, Chapter 5 gives our concluding remarks.

Chapter 2: Artificial Neural Networks

2.1 Overview

Artificial neural networks (ANNs) are mathematical models consisting of simple "artificial neurons" arranged in interconnected layers. Their structure relies on *compositionality*: individual neurons are simple, but the way they combine is complex. In this way, artificial neural networks take inspiration from human and animal brains, and bear a loose relation to them – hence their name.

Modern *deep* neural networks (DNNs) have many layers of artifical neurons, and can learn high-level hierarchical concepts from data (without an engineer having to explicitly lay out what specific aspects of the data are "important" for a given task). In this vein, DNNs have achieved incredible success over manual hand-engineered features in a variety of areas. As of this writing, deep neural networks have set the state of the art in speech recognition, language translation, natural language understanding, object recognition, automatic image captioning, DNA mutation analysis, handwriting and font recognition, smart image upscaling, text-to-speech synthesis, automatic image and music generation, and a variety of other applications, often beating previous methods by an order of magnitude. [1]

Figure 2.1 shows an example ANN, taken from the Google Brain project [6].

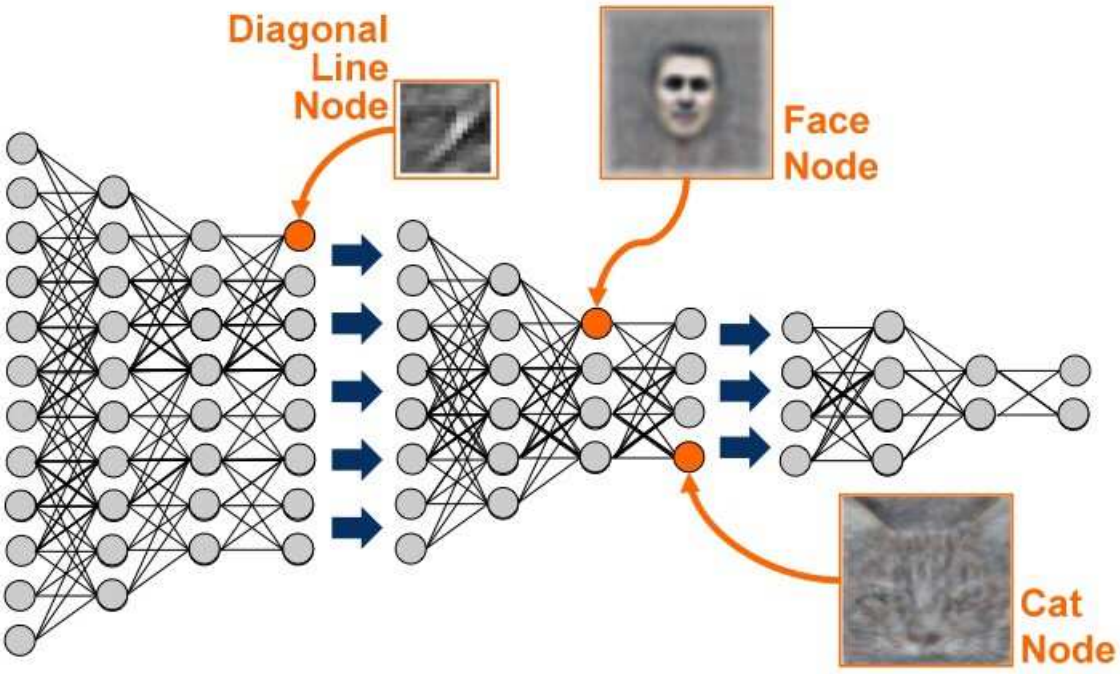


Figure 2.1: An example neural network. Taken from [6].

The gray circles represent artificial neurons, and the black lines represent connections between them; each vertical column of neurons is one *layer* in the neural network's feed-forward structure.[1]

This network was initialized with random parameters and exposed to 10 million still images from unlabeled YouTube videos, with the goal of learning how to represent them as accurately as possible. As the highlighted orange nodes show, after the ANN finished training, it had learnt to capture important concepts about images without any direct human guidance. Neurons in lower-level layers learnt to activate on basic image features such as diagonal lines or edges, and higher-level

[1]This image is not fully representative of the trained Google Brain network; it only shows a few neurons, connections and layers, whereas the real neural net had about a billion parameters.

neurons learnt to compose these simple features into complex concepts that occurred across many images (like human and cat faces: the two things YouTube users love the most).

To better understand the differences between traditional feature engineering and the "deep learning" approach, consider an illustrative example. Say an engineer wants to write a program to recognize faces. How might they go about this? Traditionally, they might have had to first write code to detect edges inside an image, then write a module on top of that specifying which particular types of edges (or combinations of edges) were most likely to correspond to facial features such as eyes, noses, mouths, and so on. Once their program extracted these hand-designed "facial feature" representations from each image, the engineer could finally train some sort of simple classifier that relates these features to a person's identity. This can be a very brittle process, involving a lot of manual work and hand-crafted feature design at each step!

In contrast, the "deep learning" approach would involve designing a neural network which takes in a face image as input and outputs a representation of a person's identity. After the engineer constructs an appropriate network structure for the problem, he or she would then train the network by showing it many example images of faces, each paired with a person's identity, and letting it learn a function mapping faces to identities by itself. The engineering effort is now "transferred" in a sense: the problem is no longer engineering features about faces directly, but rather designing and training a neural network that can accurately *learn* features of faces *and* relate them to a person's identity. The details of this learning process

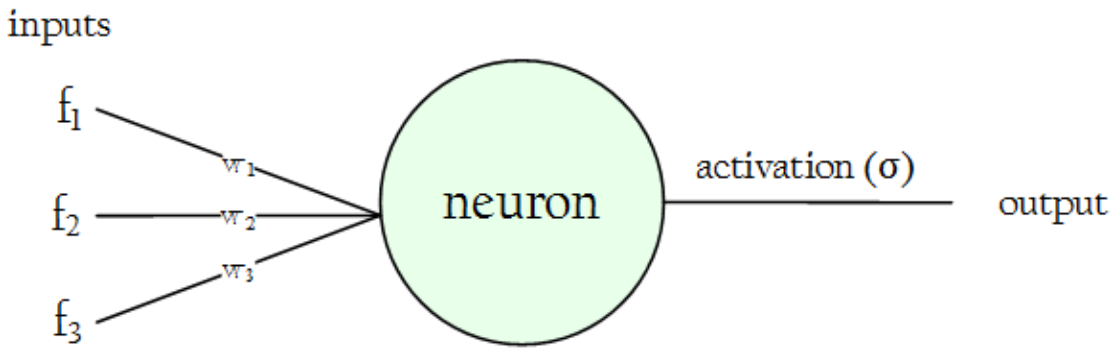


Figure 2.2: A simple artificial neuron, with 3 inputs.

are what this thesis is concerned with.

We now review the specifics of how DNNs work, starting with their most basic unit: the artificial neuron.

2.2 Basic Artificial Neurons

In its simplest original form by Rosenblatt [7], an artificial neuron takes a vector of numeric features as input. Say we have k features. Then, in order to process an input vector, the neuron computes a weighted sum of each feature (plus a bias term) and feeds the output into some predetermined *activation function*, as follows:

$$P(f_{1,2,\ldots,k}) = \sigma([\sum_{i=1}^{k} w_i f_i] + b)$$

where f_i is the input vector (of size k features), w_i are the weights for each individual feature, b is the perceptron's bias, and σ is the activation function. This is illustrated in Figure 2.2.

There are many choices for the activation function σ, but the simplest is the

binary step function (which was the activation function used in Rosenblatt's original formulation). We formally denote the binary step function as Ω, and define it as follows:

$$\Omega(x) = \begin{cases} 1 & \text{if x} > 0 \\ \\ 0 & \text{if x} \leq 0 \end{cases}$$

Then, our neuron will compute the following function:

$$P(f_{1,2,\ldots,k}) = \Omega([\sum_{i=1}^{k} w_i f_i] + b)$$

We can see that the neuron will output 1 if the weighted sum (plus bias) is greater than 0, and 0 if the weighted sum (plus bias) is less than or equal to 0. (Another way of thinking about this: the neuron will output 1 if the weighted sum of features is greater than the bias, and 0 if the weighted sum of features is less than or equal to the bias.) Finally, notice that we can reformulate the above as:

$$P(f_{1,2,\ldots,k}) = \Omega(\sum_{i=1}^{k+1} w_i f_i)$$

where f_{k+1} is always equal to 1, and w_{k+1} is the bias. So the neuron's output is just the activation function Ω applied to the dot product of the weights and inputs (with an extra 1 input added on if we want a bias term): $\Omega(w \cdot f)$.

To give a more concrete example of how this works, consider a simple binary classification problem (basic enough to be solved with a single artificial neuron). Say we have a database of patients, and we want to figure out whether each patient is sick with the flu. We have access to 3 binary high-level features about each patient: fever (does the patient have one?), vaccination status (has the patient been given

the flu vaccine?), and time (did the patient visit the clinic at day or at night?). A neuron taking those features in as input would then compute the following function:

$$P(flu) = \Omega(w_{fever} f_{fever} + w_{vaccine} f_{vaccine} + w_{time} f_{time} + w_{bias})$$

Intuitively, it seems that fever should have a high positive weight, since it's a classic and well-known flu symptom. Vaccination should have a high negative weight, since patients who have been vaccinated are unlikely to develop the flu. Finally, the time of checkin is probably completely irrelevant to whether the patient is sick, so w_{time} should be very close to 0 at the end of training.

Of course, there is still the issue of how to learn these weights from training data in the first place, which we will now examine.

2.3 Training Neural Networks

To start, it turns out single artificial neurons have a glaring theoretical problem: they can only learn very simple linearly separable functions. This limitation was analyzed in depth by Minsky and Papert in their famous book *Perceptrons* [8], where they showed that a basic artificial neuron is incapable of learning even a simple XOR function. To solve problems like this and learn more complex functions, it is necessary to "stack" neurons together and connect them to each other, forming a neural network.

Defining the structure of a neural network for a given problem can be quite complex, but for many problems, the neurons in the network are organized in a *feed-forward* fashion as in Figure 2.1. Each layer of neurons takes in the previous layer's

output, performs some operation, and forwards the result to the next layer. (The image doesn't show an input or an output layer, but to simplify we can assume that the network takes in an image at the first layer, and it outputs some representation of the image at the last layer.)

After the neural network's structure has been defined, training the model usually involves the following steps:

1. *Initializing the weights.* This is a bit more involved than it might seem at first; for example, if the weights of every neuron are all initialized to the same values, every neuron will end up learning the exact same function. In addition, different types of initializations can lead to faster or slower convergence. Discussing the exact nature of this is not important for our purposes, but the standard is to use a certain type of normal or uniform distribution, as specified by Glorot et al. [9] or He et al. [10]. This is the strategy we adopt as well.

2. *Defining an objective function.* Training a neural network involves specifying some sort of objective function (or weighted combination of objectives) to minimize over the training data. For example, in classification problems, the objective function is often the error between the network's predictions and the correct class labels; minimizing classification error amounts to the same thing as maximizing classification accuracy. The objective function also has to be differentiable almost everywhere (so an objective like "minimize the total number of misclassified training examples" is not usable). Some common objective functions for neural networks include cross-entropy and ℓ_2 distance

(also known as mean-squared error).

3. *Training via gradient descent.* Once the network's weights are initialized and its objective function has been defined, we can train the network by showing it pairs of inputs and corresponding outputs. We update its weights in an iterative fashion after each example, based on the objective function. This process repeats until we exhaust the entire training set, at which point we start over. A complete cycle through the training set is known as an *epoch*, and it is standard to train a network for a set number of epochs before stopping.

   In practice, we don't show the network input-output pairs one at a time, but rather in small *minibatches* (common sizes range from 32 to 256). To adjust the weights after each minibatch, we compute the average value of the objective function over the batch. Then, we adjust each weight of every neuron based on that value's gradient, as follows:

$$w := w - \alpha \cdot \frac{\partial}{\partial w} O(W)$$

   where w is the current weight to be adjusted, α is a parameter known as the *learning rate*, and $\frac{\partial}{\partial w} O(W)$ is the partial derivative of the objective function $O(W)$ with respect to the current weight w. The overall effect of this is to take a step towards some minimum of the objective function at each iteration (which may not necessarily be a global minimum). The size of this step depends on α, the learning rate. If α is too low, the weights will only change slightly on each iteration and training won't converge; conversely, if α is too

large, the weight updates will "overshoot" the minimum and training will oscillate or diverge. Thus, the learning rate α is an important parameter to tune when training a neural network.

A significant issue which arises in this process is exactly how to compute the partial derivatives of the objective function with respect to each weight of the network. The standard method is an algorithm known as backpropagation, popularized by Rumelhart et al. in 1988 [11], which utilizes the chain rule to compute partial derivatives throughout the network (so the derivatives of one layer depend on those in all subsequent layers). Another significant issue is how to avoid local minima and guarantee fast convergence; researchers have developed several additions to gradient descent that try to address this, such as momentum and more complex optimizers like Adam [12].

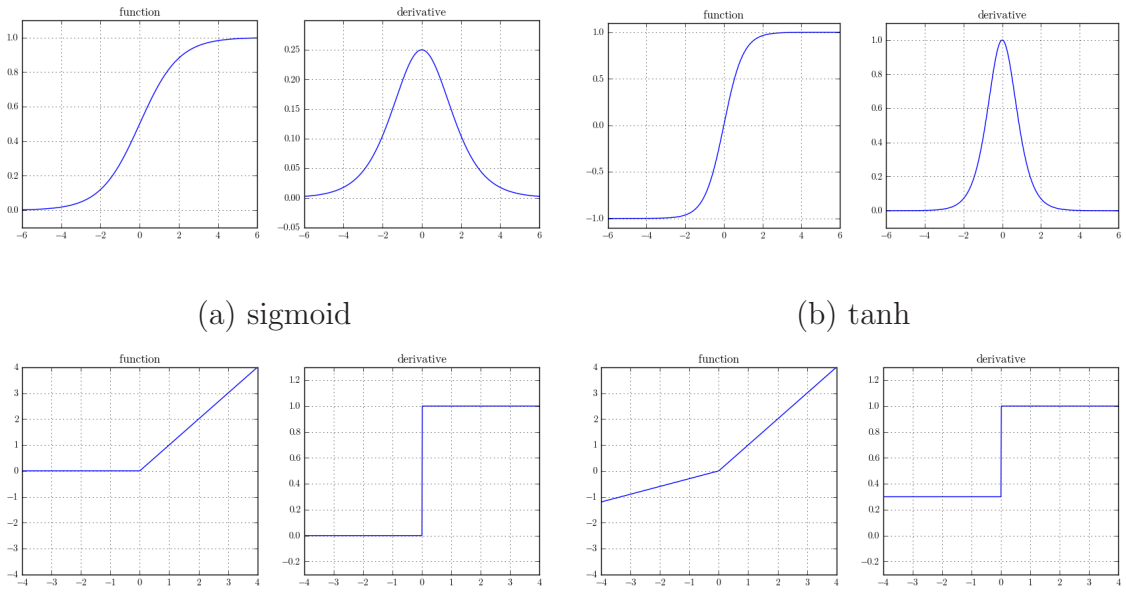
Gradient descent places an extremely important constraint on our network: every operation has to implement a *differentiable function* of its inputs. This means that, given our neuron formulation in Section 2.2, all neurons must implement a differentiable activation function. Thus, our previous binary step function Ω can't be used in a neural network trained with gradient descent, since it has zero or undefined derivative everywhere. Similarly, other non-differentiable operations such as rounding and quantization can't be used with gradient descent in their original form either; as we'll see later on, if we want to implement operations like these, we must find a differentiable approximation.

Four popular choices for a differentiable activation function are the sigmoid,

hyperbolic tangent, rectified linear unit (ReLU), and leaky rectified linear unit (Leaky ReLU) functions, with the latter two finding massive success and popularity in recent years. We show these four functions, alongside their derivatives, in Figure 2.3.

One advantage that ReLU and Leaky ReLU activations provide over the historically popular sigmoid and tanh functions is their immunity to the *vanishing gradient problem*, which is a significant issue that arises in training deep networks. [13] In classical sigmoid and tanh networks, earlier layers are much slower to train than later ones. This is because both of these functions' derivatives are contained in the interval $[0, 1]$; since gradients in earlier layers depend on those in later layers (and are calculated through multiplication), the gradients of earlier layers "vanish" and become almost zero when adjusting weights. Such behavior poses a massive problem, since the later layers in a neural network build off the earlier ones; if the early layers are essentially learning nothing, the network as a whole will learn nothing as well.

The ReLU and Leaky ReLU functions always have a derivative of 1 for $x > 0$, which mitigates the vanishing gradient problem significantly. In addition, their derivatives are simple to compute; since the functions are both piecewise linear, the derivatives are piecewise constant. The Leaky ReLU also shows better convergence properties than the original ReLU, due to having a nonzero gradient in the $x \leq 0$ region.



(a) sigmoid (b) tanh

(c) ReLU (d) Leaky ReLU

Figure 2.3: Four common activation functions, along with their derivatives.

2.4 Fully-Connected Neural Networks

Now that we've established the basic training process for neural networks, a logical next question to ask is: how can we apply ANNs to signals like images and speech? For the purposes of demonstration, we consider MNIST, a simple and classic machine learning benchmark. The MNIST dataset consists of 70,000 28x28 grayscale images of handwritten digits (60,000 for training, and 10,000 for testing and validation). Each image contains one of ten digits, 0 through 9. The goal is to create a classifier that can correctly identify digits with high accuracy.

Using the techniques we've discussed so far, we might try constructing a network like the one in Figure 2.4. We "flatten" the 28x28 image into a 1D vector of length 784, forming our input layer. Then, we connect the input layer to the first

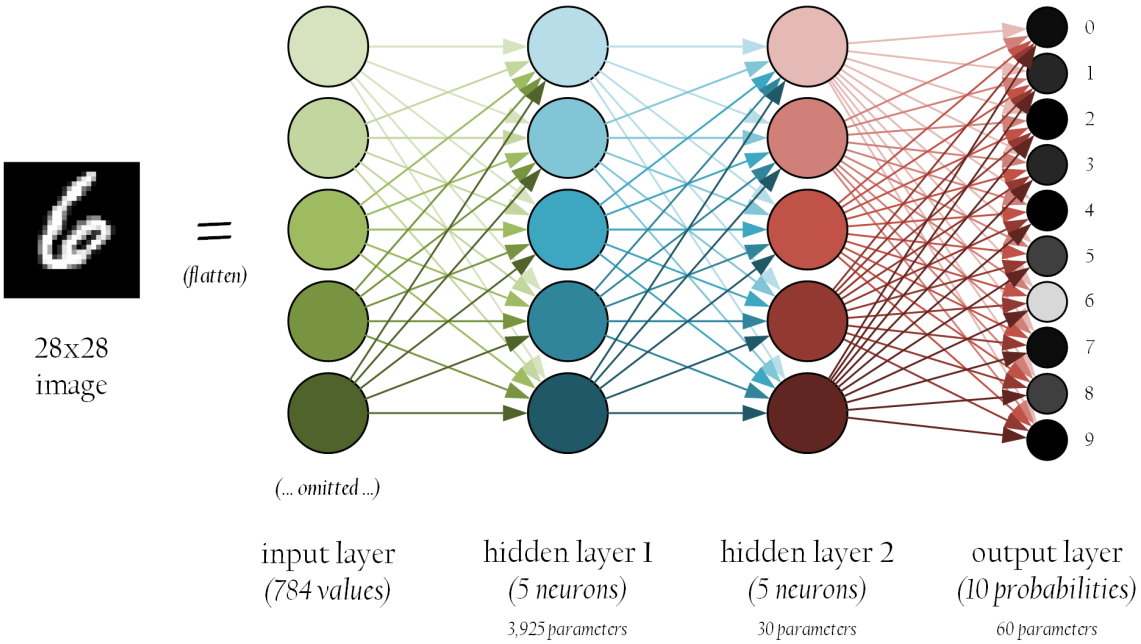


Figure 2.4: A small fully-connected neural network for MNIST.

hidden layer, with 5 neurons. We insert a second hidden layer, also with 5 neurons. Finally, the second hidden layer connects to our output layer of 10 neurons, which each output a probability for one of our 10 digit classes. Each of the neurons implements a Leaky ReLU activation function except for the final layer, which uses a *softmax* function to generate class probabilities. (All Leaky ReLU activations in this paper use a standard "negative slope" of 0.3.)

The specific form of the softmax function is important to note, since it will come up again later in a different context. From a vector v of N arbitrary real numbers, the softmax function generates a vector P of N probabilities (each between 0 and 1, and all summing up to 1) as follows:

$$P(i) = \frac{e^{v_i}}{\sum_{n=1}^{N} e^{v_n}}$$

We can see that the higher a given value in v, the higher the probability that the softmax function will assign it. This property holds over the entire set of real numbers.

The neural network model in Figure 2.4 is called a *fully-connected neural network*, since every neuron in a layer has a weighted connection to every neuron in the next layer. It is also a very small model even for this problem, with only about 4,000 parameters; still, the network achieves 89.83% test accuracy on MNIST (much better than the 10% guaranteed by chance). To put these numbers in context, LeCun et al.'s seminal paper introducing MNIST [14] discusses a similar fully-connected network with about 270,000 parameters, which achieves 96.95% test accuracy. (The main differences are that their first hidden layer has 300 neurons, and the second 100; also, we train using the modern Adam optimizer [12], rather than vanilla gradient descent.) And even the network of LeCun et al. would be considered fairly small by today's standards; recent neural networks trained for general object recognition problems have reached tens of millions of parameters [15] [16].

Fully-connected networks work decently well for small, simple problems such as digit recognition. However, at larger image sizes, or for more complex problems requiring larger networks with more neurons, they suffer from a parameter explosion. For example, if we want to process a modestly sized 128x128x3 RGB image with a hidden layer of 100 neurons, that one layer alone would require almost five million parameters! In addition, fully-connected layers are highly redundant, failing to exploit any of the inherent spatial structure of images or audio. They also have no real spatial invariance, since every part of the entire signal gets a different weight.

2.5 Convolutional Neural Networks

Convolutional neural networks (CNNs) were invented to address these shortcomings. They incorporate spatial structure, are spatially invariant by design, and are generally able to make much more efficient use of their parameters than fully-connected networks can. In addition, CNNs can handle large signals (speech, images, etc.) without succumbing to parameter explosion.

Instead of giving each input a unique weight, a convolutional layer slides *filters* across the signal, recording each filter's response at every position. More formally, the layer computes the *convolution* of each filter with the input signal (hence the model's name). The output of each filter becomes one *channel* of the layer's output. Each channel of a layer's input is filtered simultaneously; for example, a 7x7 filter over an RGB image would really be implemented as a 7x7x3 filter, with a separate sub-filter for each color channel. The convolution filterbanks of each layer are the learnable parameters of the network. Finally, each convolutional layer also has an activation function, as before.

We illustrate the convolution operation in Figure 2.5. The 7x7 filter shown is a basic horizontal edge detector, which essentially looks for the pattern "dark area over light area". When it finds a matching region of the image, it outputs a high response, as in the top rectangle. When it finds a very dissimilar region, it outputs a low response, as in the bottom rectangle (which is "light area over dark area", and thus the exact opposite of what the filter is looking for).

Since the same filters are used over all points of the signal, convolutional lay-

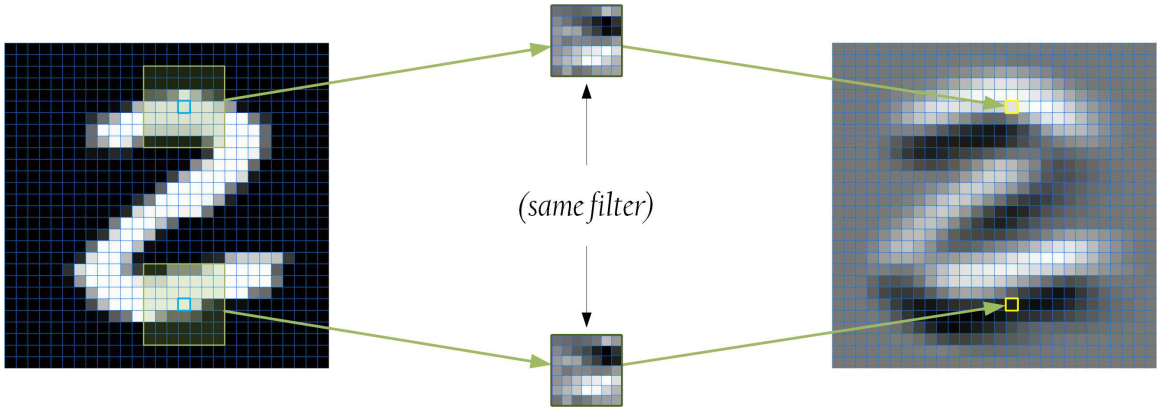


Figure 2.5: Illustration of a convolution operation.

ers have a degree of spatial invariance that fully-connected ones don't. In addition, convolutional layers directly incorporate the spatial structure of the signal, by considering small, local parts of it at any given time. Finally, since the filter size is independent of the signal size (a 7x7 filter remains a 7x7 filter no matter how large the signal it is processing), CNNs do not suffer from parameter explosion on large inputs to the same extent that fully-connected layers do.

Figure 2.6 shows a very small convolutional neural network trained for the MNIST problem. This CNN is even smaller than the fully-connected network in Figure 2.4 (at about 3,000 trainable parameters), yet achieves a much higher 98.66% accuracy on the MNIST test set! The network is constructed by alternating convolution operations with *pooling operations* that reduce each channel's spatial dimension by a half. Each convolutional layer has 4 filters (channels), of size 7x7 per input channel, and implements a Leaky ReLU activation function. After two sets of convolution/pooling layers, the resulting 7x7x4 vector is flattened into a 1D vector of length 196. This 196-length vector is finally linked to a fully-connected layer of 10

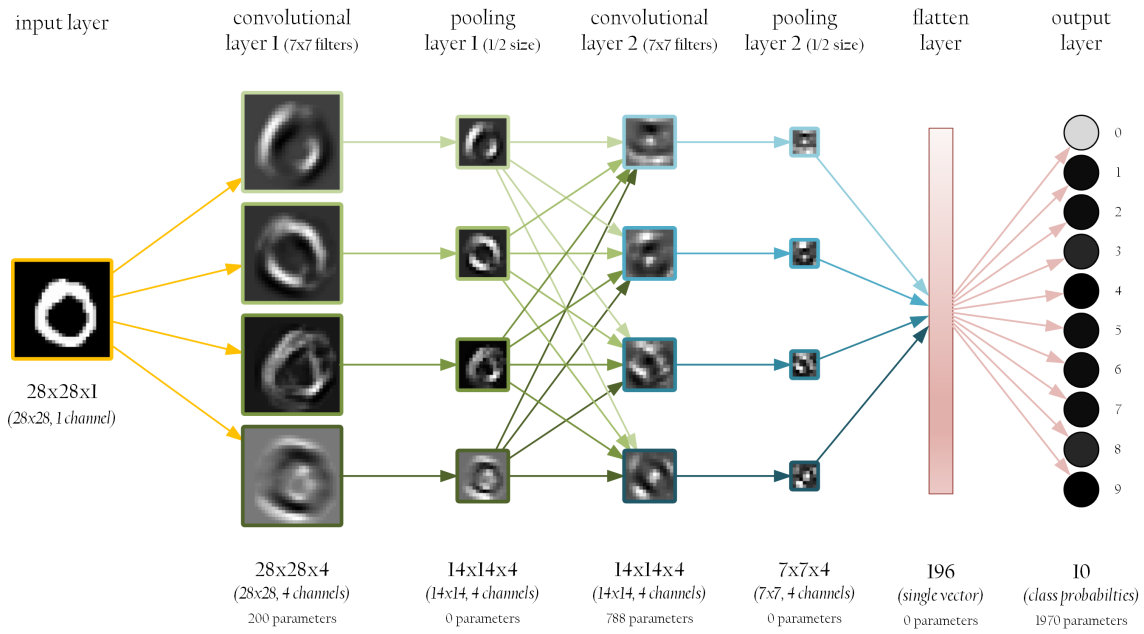


Figure 2.6: A small convolutional neural network for MNIST.

neurons, which outputs class probabilities (using the softmax function as before). (We also show a simplified version of the network in Figure 2.7.)

Because of their alternating convolution and pooling operations, CNNs can examine their input signal at several scales as they get deeper. It has been shown that convolutional layers exhibit compositionality [17]; for example, when trained on images, earlier layers of CNNs will often learn simple problem-agnostic edge and blob detectors, while later layers will learn progressively more specific and complex features (texture, shape, object class) by building off and combining the features of the earlier layers. This is an incredibly appealing property, and also implies that constructing deep networks is of paramount importance if we want to learn the richest features and concepts possible. Empirical findings corroborate this hypothesis, with deeper CNN architectures performing much better than shallow ones [18] [19].

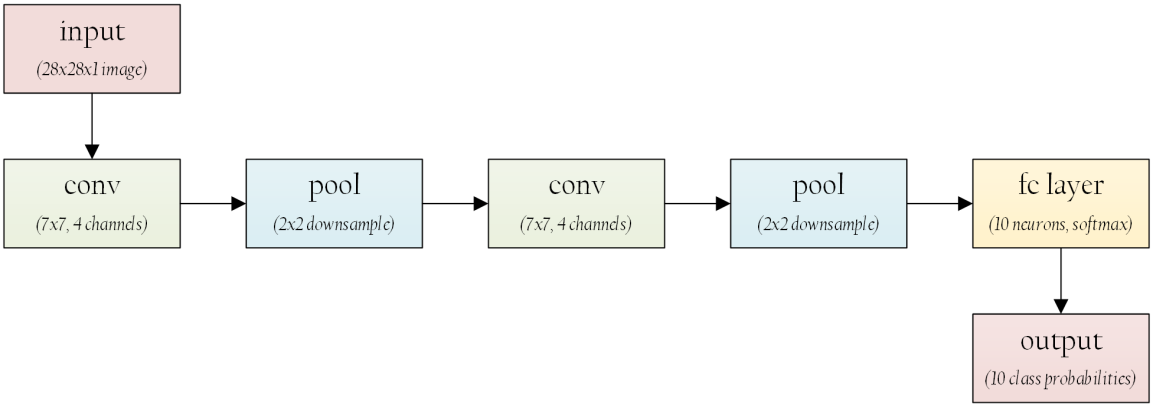


Figure 2.7: A simplified version of Figure 2.6.

2.6 Residual Neural Networks

Deep CNNs are generally quite hard to train past a dozen or so layers, requiring careful architecture engineering such as that seen in the VGG [18] or Inception [19] networks. He et al. [20] found that deep 56-layer CNNs had worse training *and* test error than shallower 20-layer ones, a degradation which could not be explained by either vanishing gradients or overfitting. This is strange, since it's theoretically easy for a deeper network to perform just as well as a shallower one. For example, we could just take the 20-layer network and add 36 more layers that all perform an identity mapping, resulting in a 56-layer one that computes the same exact function. From this observation, He et al. concluded that identity mappings are hard for neural networks to learn and optimize, which led them to propose a new CNN variation known as a *residual neural network*. These models can be trained with hundreds or even thousands of layers, and achieve state-of-the-art performance on image and audio tasks.

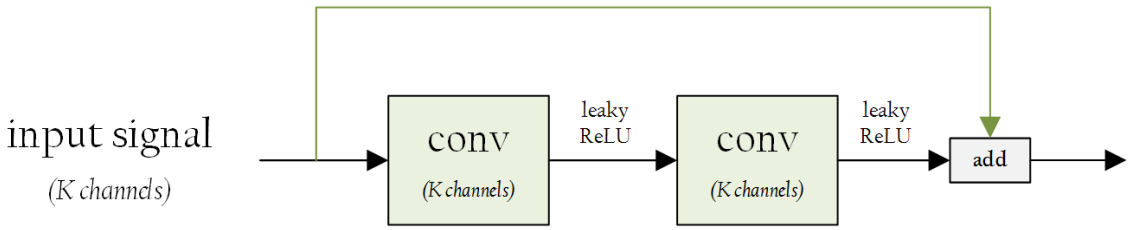
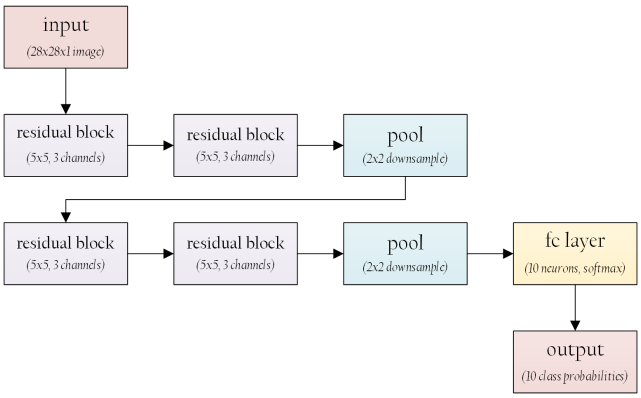


Figure 2.8: A basic residual block, with the shortcut connection in green.

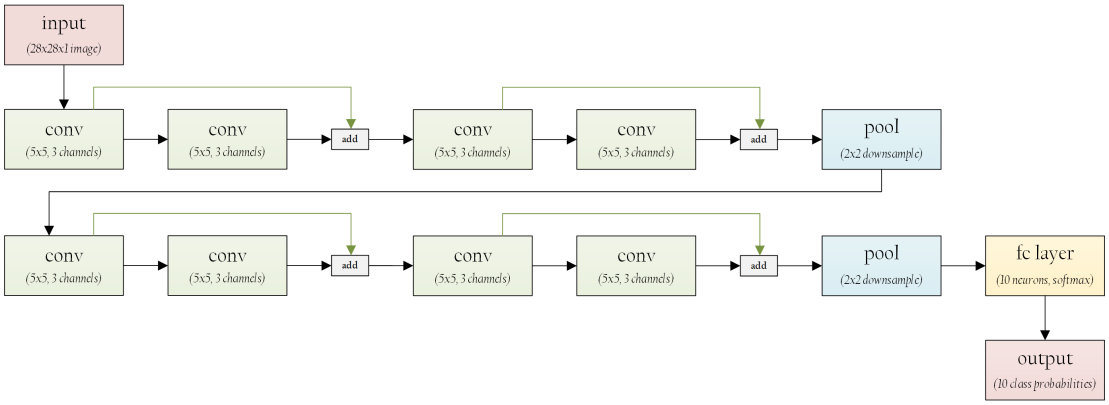
Residual networks reframe the learning process as follows: instead of layers learning a direct transformation $x \to F(x)$, the network consists of *residual blocks*, each of which learn a *residual* transformation $x \to x + R(x)$. In other words, residual blocks refine their input, rather than transforming it to a new level of representation [21]. A residual neural network consists of several *stages* of residual blocks, each stage refining a different level of representation.

We show an example residual block in Figure 2.8, with the *shortcut connection* in green. Note that this is our own slight variation of the original residual block; He et al. used the original ReLU activation instead of Leaky ReLU in [20], and also added a batch normalization operation [22] between the convolution and activation. We found that batch normalization degraded training in our architecture, so we omit it from all residual blocks in this work. In addition, He et al. had the second ReLU activation after the addition; we move it directly before, in the interest of preserving the identity mapping (which is a good property to have, as elaborated in the authors' later further examination of residual networks [23]).

Figure 2.9 shows a very small residual network for MNIST, also around 3,000 parameters (similar to the network in Figures 2.6 and 2.7). We show both a simpli-



(a) simplified



(b) "unrolled"

Figure 2.9: A small residual network for MNIST.

fied version (where the residual blocks are "abstracted away") and, for completeness, an "unrolled" version where all convolutions and additions are visible. (Future figures will show only simplified networks, to save space.) This model is about 4 times deeper than the one shown in Figure 2.7, with 8 convolutional layers, and achieves a slightly higher 98.83% test accuracy on MNIST. We can also confirm He et al.'s observations about depth by removing the shortcut connections from the network (turning into just a regular convolutional network); we observe that both train and test accuracy degrade by about 2%, performing worse overall than the

earlier shallower model.

The encoder/decoder model we later formulate is just a variation of a residual neural network, with some further properties that make it appealing as an end-to-end compression system. Thus, all of the background on ANNs necessary to understand our approach is finally in place.

Chapter 3: Related Work

3.1 Current Speech Coding Standards

While the typical human ear can hear frequencies up to 20,000 Hz, transmitted speech has historically only captured frequencies from 300 to 3400 Hz (at a sampling rate of 8000 Hz, with each sample taking up 16 bits) [24]. This is known as *narrowband* speech, because of the narrow range of frequencies it captures. Narrowband speech is adequate for many purposes, but because of the loss of high-frequency information, it suffers from a characteristic "muffled" sound and lack of intelligibility (especially during fricative consonants) [25].

As a result, researchers have put much effort into the development of *wideband* speech, which can capture frequencies from 50 to 7000-8000 Hz (at a sampling rate of 16,000 Hz). Major mobile providers have started to deploy wideband speech to their end users (marketed as "HD voice") in the last few years. Even though most consumers are still served narrowband speech, this trend is expected to continue [26]. This thesis will deal solely with 16-bit wideband speech sampled at 16,000 Hz.

The two most prominent standards for wideband speech are G.722.1 [27] and AMR-WB [28] (also known as G.722.2). These algorithms both take different underlying approaches to the speech coding problem.

1. *G.722.1.* [27] Provides speech at two moderate bitrates (24 and 32 kbps), and is based on *transform coding*. The idea behind transform coding is to take a small speech window (say, 20ms long), apply some mathematical transform such that most of the signal's energy is concentrated in a few coefficients (rather than spread across the entire time domain like before), and only encode and transmit the most significant ones. G.722.1 uses a process based on the discrete cosine transform (DCT), similar to JPEG.

2. *AMR-WB.* [28] Provides speech at several low to moderate birates (6.6 up to 23.85 kbps), and is based on *linear prediction*. The idea behind linear prediction is to model the speech signal at a particular point in time as a weighted sum of the last few samples before it, plus some excitation term:

$$S_n = [\sum_{i=1}^{K} W_i S_{n-i}] + E_n \tag{3.1}$$

   A sufficiently short speech window can be accurately reproduced just through the parameters of this linear model, without needing to transmit any direct information about the original signal. However, determining what specific weights and excitation patterns best fit a speech signal at a given time is a complex open-ended problem. The exact details of how AMR-WB does this are beyond the scope of this brief review.

Both of these standards were the product of months or years of specialized research, conducted between several different organizations. G.722.1 was created by Picture-Tel (now Polycom, Inc.) and approved by ITU-T after a four-year selection process;

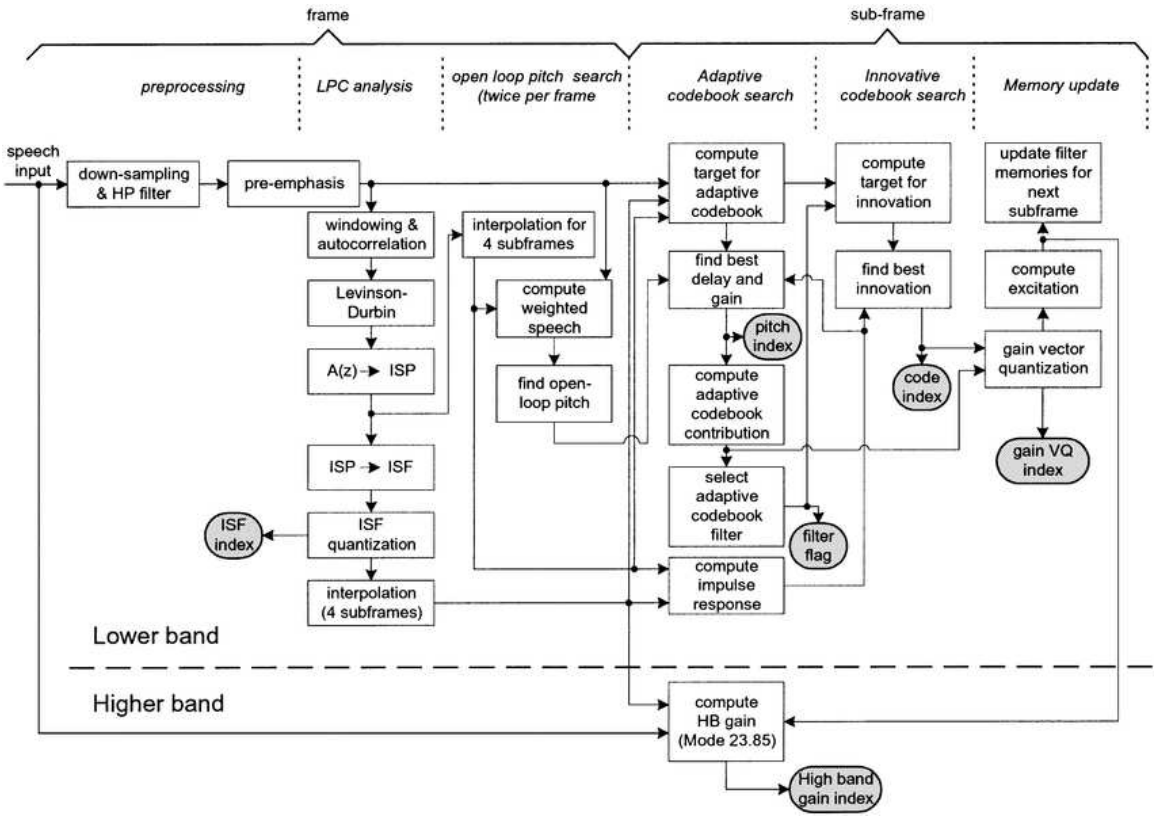


Figure 3.1: A block diagram of the AMR-WB encoder. Taken from [28].

AMR-WB was created by eight speech coding researchers working at VoiceAge Corporation (in Montreal) and the Nokia Research Center (in Tampere, Finland) over a period of two years. The codecs themselves are very elaborate, involving several interconnected layers of processing steps. To give a representative example of what a modern speech coder looks like, we show the AMR-WB encoder in Figure 3.1.

This encoder consists of a preprocessing step followed by five different processing steps; each of these steps can be further broken down into substeps, the outputs of which connect and feed into each other in various complex ways. Designing the AMR-WB encoding pipeline involved the tuning of many parameters,

including the exact forms of the digital filters involved (high-pass, pre-emphasis, perceptual weighting, excitation, etc.), the number of bits allocated at each bitrate for the encoder's different outputs (eight parameters in total for each bitrate), the specific ranges in the pitch search portion of the algorithm, the structure and design of the algebraic codebook, and more – not to mention creating the overall structure and flow of the algorithm, independent of the specific parameters of each step! [28]

We believe this representative example helps put our work in context by clarifying just how much work is involved in creating a modern compression algorithm, and further motivates our research into automated data-driven approaches.

3.2 A Brief History of ANN-Based Compression

Artificial neural networks have seen significant interest from compression researchers, most often as an intermediate pipeline step or as a way to optimize the parameters of an intermediate step [29]. A few select examples of this, from oldest to most recent:

- *1990*: Krishnamurthy et al. [30] extracted traditional features from speech and images, then performed vector quantization using a neural network (to compress said features). They achieved 8x compression on grayscale images with decent quality (and did not provide compression ratios for speech).

- *1994*: Wu et al. [31] used a neural network as one of the steps of a non-linear predictive speech coder. They created a series of narrow-band speech coders ranging from 3kbps to 13kbps, with reasonable quality (from their figures,

seeming to perform significantly worse in subjective tests than the narrowband AMR standard [32]).

- *2001*: Park et al. [33] again combined neural networks and vector quantization for image compression, this time using a competitive neural network to alleviate degradation of edges. They achieved about 11x compression on grayscale images.

- *2008*: Khashman et al. [34] used a neural network to determine the optimal ratio for a Haar wavelet-based compression algorithm.

- *2015*: Cernak et al. [35] used a deep neural network as a phonological vocoder, for low-quality and very-low-bitrate (1kbps) wideband speech. This approach achieved a very low rate of transmission, but significantly degraded quality since many speaker- and environment-specific qualities were lost.

Our proposal is different in nature from all of the above. Rather than using a neural network as a tool in a hand-designed compression pipeline, we want a neural network to constitute the entire compression pipeline itself, from start to finish, and we want to optimize it end-to-end for the type of signal we are compressing. In our case, this means optimizing the network to compress speech, but we can extend the general idea to other types of signals (such as images, video, text, etc.) More specifically, we want to create "encoder"/"decoder" neural networks and optimize them in tandem, so that at the end of training we have a full speech coder operating as follows:

1. Receive a raw speech window, and feed it through the encoder network to

obtain a compressed representation.

2. Transmit the compressed representation to the decoder network, over some channel.

3. Feed the compressed representation through the decoder network, obtaining a perceptually accurate reconstruction of the original speech signal.

This end-to-end approach is a much rarer idea in the literature; however, it is not entirely absent either. In fact, it dates all the way back to 1987 with the work of Cottrell et al. [36], just a year after the "backpropagation" algorithm was popularized and it became possible to train complex multi-layer neural network models in the first place. This is the first paper we know of which attempted to apply neural networks to data compression; the authors achieved 8× compression of grayscale images at decent quality, by training a simple neural network with one hidden layer (for a total of three layers, including the input and output layer) on 16×16 image patches.

Morishima et al. [2] extended this idea to speech coding in 1990, again using a three-layer neural network; as far as we know, this was the only speech coder learned truly end-to-end before this work. Namphol et al. [37] extended the image compression network of Cottrell et al. to five layers (three hidden layers) in 1991. Iterative advances on these works continued through the late '90s, with much interest from researchers and many different ideas being proposed [4]. Many of these proposed neural networks, it is important to note, implemented strictly linear transformations (i.e. they did not incorporate any nonlinear activation functions such as

the sigmoid, hyperbolic tangent, or rectified linear unit).

3.2.1 Limitations of Prior Work

From the late '90s up until very recently, end-to-end compression systems using ANNs seem to have almost vanished from the literature. At most, researchers have utilized ANNs as secondary or tertiary tools to optimize parts of a hand-designed system (as the papers mentioned at the beginning of the previous section did, among others). This is somewhat surprising, since all of the papers mentioned in the last few paragraphs achieved promising results, especially for their time.

In the end, it turns out that the neural network models that researchers were using had several key limitations, and this was becoming clear even by the end of the '90s (not only in compression, but in other areas as well).[1] To wit:

- *The vanishing gradient problem.* As discussed in Chapter 2, the classical sigmoid and tanh activation functions suffer from the vanishing gradient problem, which limits the capacity and depth of neural networks trained with these types of activations. By alleviating this, the ReLU and its variations [38] [39] [40] were responsible for much of the success of modern neural networks.

- *Limited depth and structure.* All of the models covered by Jiang et al.'s review [4] were shallow, encompassing only three to five layers total. This is related to the vanishing gradient problem, which made it hard to train deeper

[1] For a more mathematical analysis of why classical nonlinear neural networks were so hard to train in comparison to modern ones, the reader is encouraged to consult Glorot et al. [9]

networks. Sophisticated network structures like residual networks had also not been created yet; these structural innovations are essential to our proposed model.

- *Linear designs.* Many of these proposed neural networks up through the late '90s were actually fully linear in nature [4], rendering them inherently less capable than a neural network containing nonlinearities. However, deep nonlinear neural networks have historically been hard to train effectively. [9]

- *No perceptual loss.* Traditionally, models were trained to minimize the ℓ_1 or ℓ_2 distance between the original and reconstructed signal. As is now well-known [41] [42], this leads to blurry reconstructions that lose a lot of high-frequency information. Both these losses assume that the data is drawn from a Gaussian or Laplacian distribution; therefore, they are ill-suited to capturing multimodal distributions such as speech or images. Thus, the ℓ_1 or ℓ_2 loss must be further augmented with an additional perceptual loss.

- *Unclear how to control bitrate.* If our end goal is an ANN-based compression system that can target multiple bitrates, then it's unclear how to control the bitrate of the compressed representations (generated by the encoder network), short of hand-designing an entirely different neural network for each bitrate.

These problems effectively killed the community's interest in learning end-to-end compression systems using ANNs, until very recently.

3.2.2 Recent Developments

Recently, there has been a small resurgence of interest in learning compression algorithms directly from data; however, this has been almost entirely limited to the domain of images. In 2015, Toderici et al. reported promising preliminary results applying deep recurrent neural networks to compress 32x32 thumbnails [43], and in 2016, they published an end-to-end recurrent system for full images which outperformed JPEG [5]. In late 2016, Theis et al. [44] and Balle et al. [45] both independently proposed (non-recurrent) ANN systems that outperformed JPEG2000. Finally, early in 2017, Johnston et al. built on the work of [5] to create a system outperforming all state-of-the-art image codecs [46], and Agustsson et al. [47] published an independent (non-recurrent) approach with similar results.

The most similar prior work to ours was published by Cernak et al. [3] in 2016. They propose an almost-end-to-end neural network design for a very-low-bitrate low-quality speech coder, which combines many deep and spiking neural networks together. However, in addition to being quite complex, their design requires extraction of traditional features at the very start of the pipeline; thus, they fall just short of achieving end-to-end training. Our work targets higher bitrates than theirs, so in a sense, our two approaches are orthogonal; still, our model architecture and training process are simplified in comparison to theirs.

The model we propose in the following chapter addresses all of the issues discussed in the previous section, in one form or another.

Chapter 4: Compressive Residual Autoencoders

4.1 Architecture and Design

We now introduce our end-to-end neural network architecture for speech coding, which we call a *compressive residual autoencoder*: compressive because it performs the task of compression, and a residual autoencoder because its structure is derived from residual neural networks [20] and autoencoders [48].

The simplified network architecture is shown in Figure 4.1. The model takes in a vector of 512 speech samples (a 32ms speech window) and outputs another vector of 512 speech samples (the reconstructed speech window after compression and decompression). Its structure consists of 4 different types of blocks (channel change, residual, downsample, upsample), each a variation on the basic residual

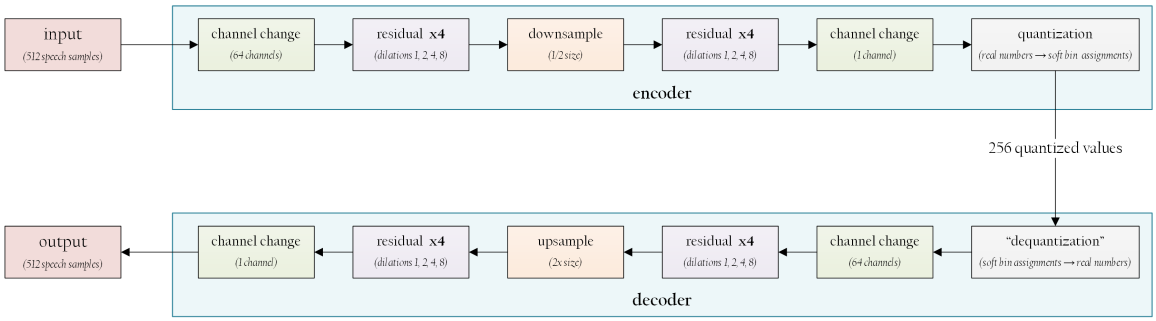


Figure 4.1: Simplified network architecture.

block, as well as a quantizer and "dequantizer".

The encoder is the subnetwork up to and including the quantizer. It takes in a window of 512 speech samples, transforms it into a vector of 256 real numbers, and finally applies quantization, resulting in a vector of shape 256xN (where N is the number of quantization bins). The quantization process will be discussed in depth later, but essentially, each of the 256 real numbers are "softly assigned" among the N quantization bins during training (with a hard one-bin assignment being used at test time).

The decoder is the subnetwork following the quantizer. It takes in the quantized encoder output of shape 256xN, "dequantizes" it back to a vector of 256 real numbers, and transforms this back into a reconstructed window of 512 speech samples. At training time, we train this entire model end-to-end (as in a classical autoencoder), but once the model is trained, we can use the encoder and decoder portions separately.

At this point, our encoded output takes $256 * log_2 N$ bits to represent. For example, with $N = 32$ (32 quantization bins), the encoded output would take 1280 bits, or 160 bytes. 512 raw 16-bit speech samples take up 1024 bytes; thus, this encoder would achieve 6.4x compression. Since raw wideband speech has a bitrate of 256kbps, 6.4x compression results in a bitrate of 40kbps.

We can achieve a better compression ratio by applying entropy coding to the output of the encoder, such that our complete compression pipeline looks like Figure 4.2. Entropy coding is a form of lossless coding which uses fewer bits to code common bitstrings of a message, and more bits to code rare bitstrings. The *entropy* refers

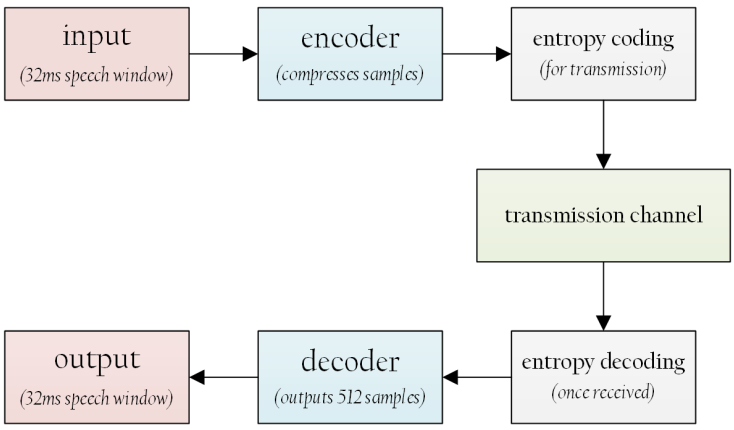


Figure 4.2: Our full compression pipeline.

to the average number of bits required to represent one symbol of a message; with entropy coding, our encoded output takes $256 * entropy$ bits to represent.

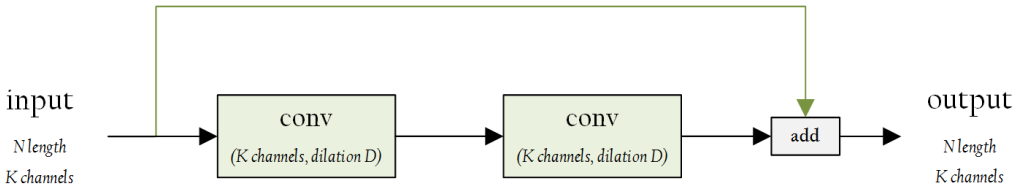
Entropy coding also provides a simple way to train networks at different desired bitrates, without having to engineer different network architectures for each. Depending on our desired bitrate, we can simply constrain the entropy of the encoded speech windows to be lower or higher.

4.1.1 Residual Block Types

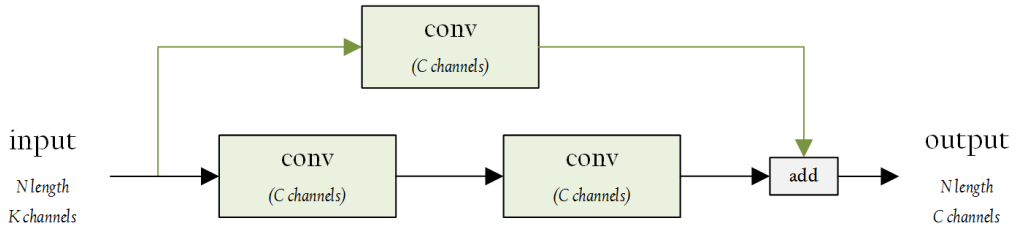
Figure 4.3 shows the four types of residual blocks used in our network architecture, which are all variations of the one shown in Figure 2.8. They are as follows:

1. *Residual block.* Takes an input of K channels with N length each, and outputs K channels with N length each. This is the same as the residual block in Figure 2.8, except that it allows support for dilated convolutions [49].

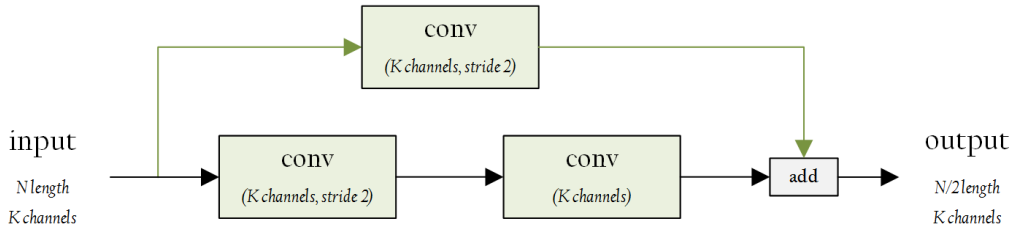
2. *Channel change block.* Takes an input of K channels with N length each, and



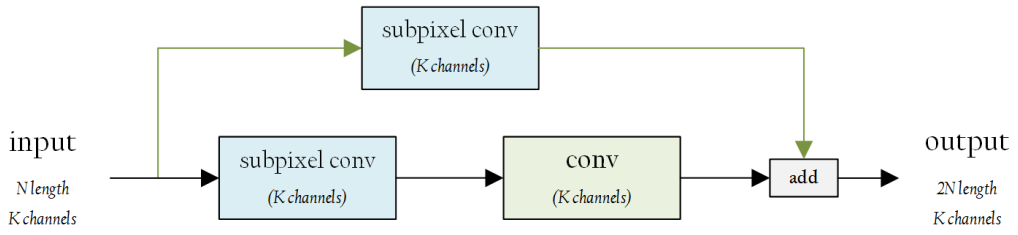
(a) residual



(b) channel change



(c) downsample



(d) upsample

Figure 4.3: The four block types used in our network architecture.

outputs C channels with N length each. This necessitates adding a convolution operation along the shortcut connection. This allows going from 1 input channel to multiple intermediate processing channels, and back down.

3. *Downsample block.* Takes an input of K channels with N length each, and outputs K channels with $\frac{N}{2}$ length each. In other words, it halves the length of each channel, via strided convolution. This allows the network to learn its own downsampling operation, instead of e.g. a hardcoded linear downsampling step.

4. *Upsample block.* Takes an input of K channels with N length each, and outputs K channels with $2N$ length each. In other words, it doubles the length of each channel, via "subpixel convolution" (as introduced by Shi et al. [50]). This allows the network to learn its own upsampling operation, instead of e.g. a hardcoded linear upsampling step.

The activation functions are omitted in the figures, but all convolutions inside these blocks utilize Parametric ReLU activations [51] (a variant of Leaky ReLU where the slope in the $x \leq 0$ region is a learnable parameter of the network). There is unobstructed gradient flow from the beginning of the network to the very end, since every single block has a shortcut connection.

4.1.2 Softmax Quantization

Quantization is the process of mapping real values into discrete "bins". In our network design, we use a quantizer to map the real-valued[1] output of our encoder into discrete values taking up less space. This quantization process is the second aspect of our architecture which must be discussed in depth.

Quantization is inherently non-differentiable. Figure 4.4 shows a (uniform) quantizer with 5 bins, as well as its derivative. The derivative is zero everywhere except for the bin transitions, where it becomes infinity. (This is similar to the binary step function discussed in Section 2.2.) Thus, quantization is fundamentally incompatible with gradient descent; in order to incorporate a quantizer into our encoder, we need to find a good differentiable approximation.

[1]Technically, these are not real values, but rather 32-bit floating point numbers.

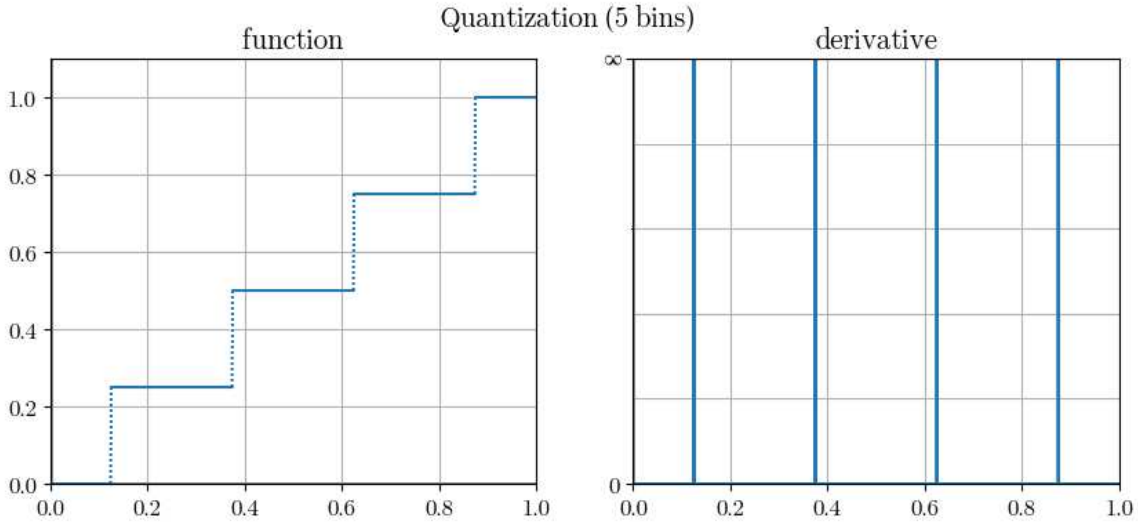


Figure 4.4: 5-step uniform quantizer and its derivative.

The approximation we use is a variation of the one first discussed by Agustsson et al. [47] in 2017. We reframe scalar quantization as nearest-neighbor assignment: given a list B of N bins, we quantize a scalar x by assigning it to the *nearest quantization bin*. Nearest-neighbor assignment still isn't differentiable, but it can be approximated as follows:

1. Compute the distance from x to each of the bins in B, and call this list of distances D.

2. Compute the *softmax* (first discussed in Section 2.4) over $-D$ with temperature σ, and call the resulting probability distribution S. More specifically:

$$S = softmax(-\sigma D)$$

   As $\sigma \to \infty$, S will have a probability of 1 at the bin nearest to x, and a probability of 0 everywhere else. This type of distribution is also known as a *one-hot vector*: a vector with 1 at exactly one index and 0 at every other, identifying a particular value associated with that index. (Recall that the softmax function turns arbitrary vectors into probability distributions, where higher values are assigned higher probabiltiies. Here, we want the *lowest* distance to have the *highest* probability, which is why we negate D when applying the softmax.)

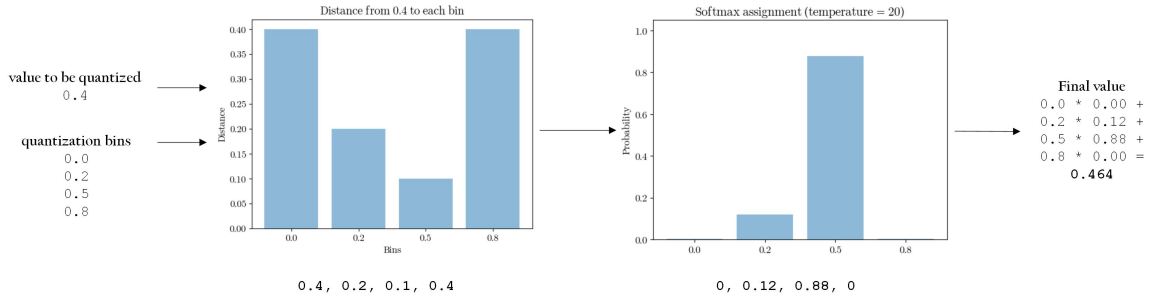
   Thus, during the quantization process, a scalar will get turned into a vector of length N, a vector of length L will get turned into a matrix of size LxN, a matrix of size AxB will get turned into one of size $AxBxN$, and so on. Each sample is replaced by a probability distribution (soft assignment) over quantization bins, and these distributions are what the encoder outputs.

3. On the decoder side, we can "dequantize" the probability distribution S back into a real value $\hat{S}$ by taking the dot product of S and B:

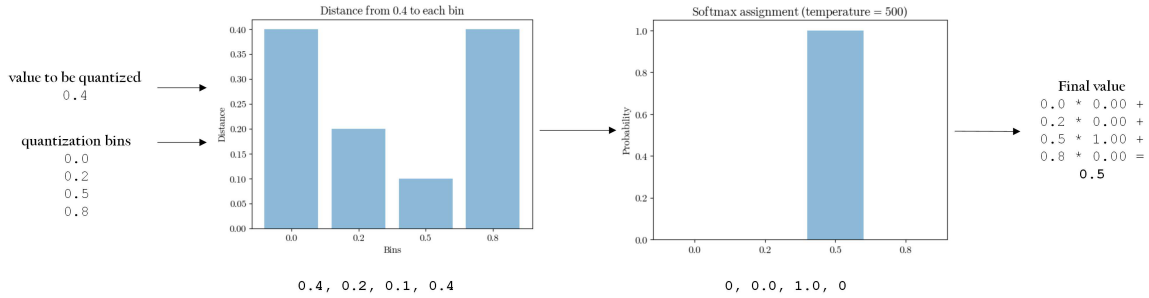
$$\hat{S} = \sum_{i=0}^{N} S_i B_i$$

We refer to this approximation as *softmax quantization*, for the softmax function which we use to approximate nearest-neighbor assignment.[2] Figure 4.5 shows the entire process visually, for both a low and a high softmax temperature. The figure also shows that, at lower temperatures, softmax quantization can fail and not properly quantize the input value.

[2]Agustsson et al. did not give their approximation a specific name.

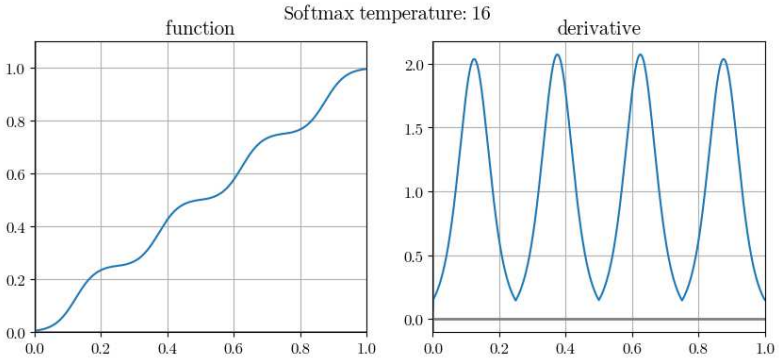


(a) low temperature ($\sigma = 20$)

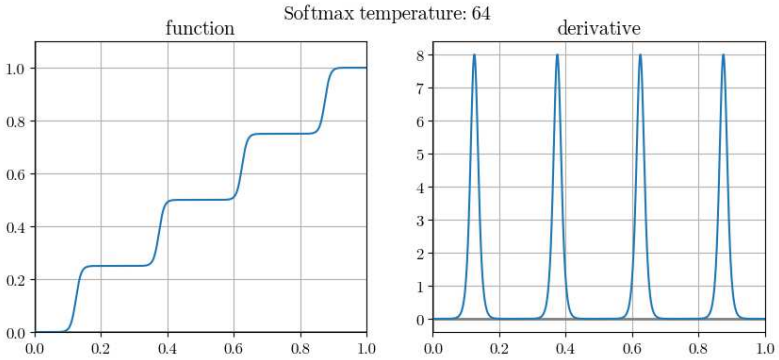


(b) high temperature ($\sigma = 500$)

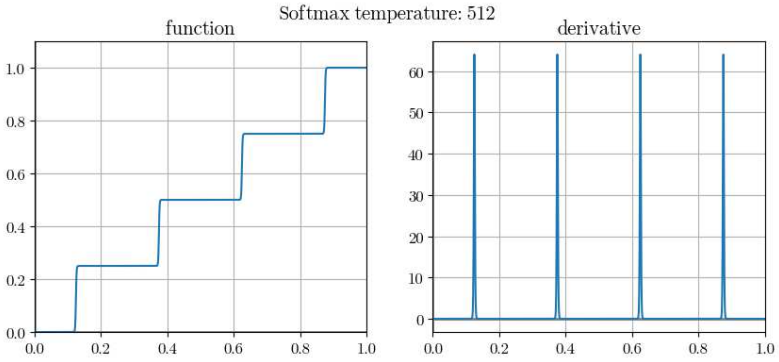
Figure 4.5: Softmax quantization, for both a low and high temperature.



(a) $\sigma = 16$



(b) $\sigma = 64$



(c) $\sigma = 512$

Figure 4.6: The effect of temperature on our softmax quantizer.

We also show the effect of temperature on softmax quantization (and its derivative) in Figure 4.6, for a quantizer with the same bins as Figure 4.4. As σ gets higher, our approximation does indeed approach the original quantizer; however, the derivative also becomes more sparse.

In theory, a function like Figure 4.6(c) with a very sparse derivative should cause optimization difficulties when used with gradient descent; this is ostensibly why Agustsson et al. annealed σ from low to high values during the course of their training process. However, in practice we noticed no problems training with very high temperature values from the start. For all experiments described henceforth, we initialize with $\sigma = 500$ and let σ be a trainable parameter of the network.

4.2 Training Methodology

We train the network on samples from the TIMIT speech corpus [52]. TIMIT contains over 6,000 wideband recordings of 630 American English speakers from 8 major dialects, pre-split into train and test sets with non-overlapping speakers. We create smaller training/validation/test sets from the already-existing train/test split, as follows:

- *Training:* 1,000 files from the original train set
- *Validation:* 100 files from the original train set
- *Test:* 500 files from the original test set

where we make sure the sets contain an even distribution over the 8 dialects, and

that the sets do not share any speakers.

We preprocess each speech file by maximizing its volume. Then, we extract raw speech windows of length 32ms (512 speech samples), with an overlap of 2ms (32 samples) using a Hann window in the overlap region. (This means that each speech window covers a total of 480 unique samples, or 30ms of speech.) For reference, the 1,000 speech files in our training set yield a little over 100,000 individual speech windows.

4.2.1 Objective Function

The network's objective function is multifaceted, incorporating both a perceptual loss over the reconstructed signal and entropy/quantization losses over the encoder's output. Its general form is as follows:

$$O(x, y, c) = \lambda_{mse}\ell_2(x, y) + \lambda_{perceptual}P(x, y) + \lambda_{quantization}Q(c) + \lambda_{entropy}E(c)$$

where x is the original signal, y is the reconstructed signal, c is the encoder's output (the soft assignments to quantization bins), $\ell_2(x, y)$ is ℓ_2 distance (also known as mean-squared error), $P(y, x)$ is a perceptual loss, $Q(c)$ is a quantization loss over the encoder's output, $E(c)$ is an entropy loss over the encoder's output,and λ corresponds to weights for each loss. We now discuss each of the supplemental losses $P(y, x)$, $Q(c)$, and $E(c)$ in more depth:

- *Perceptual loss.* It is well-known that training a neural network solely to minimize ℓ_2 leads to blurry reconstructions lacking in high-frequency information [41] [42]. Therefore, it is helpful to augment training with an additional

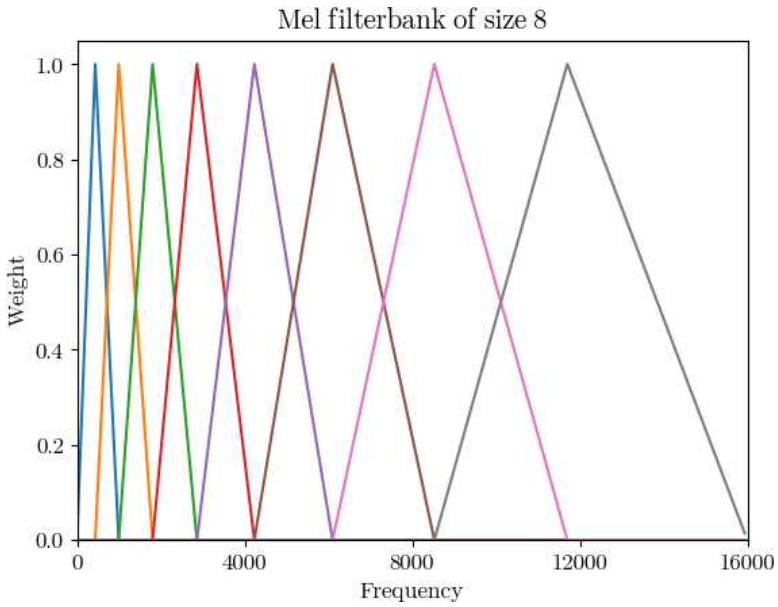


Figure 4.7: Triangular Mel filterbank with 8 filters.

perceptual loss.

Our proposed perceptual loss is based on Mel-Frequency Cepstral Coefficients (MFCCs), which are simple and classic features used in speech processing [53] [54]. MFCCs model the human ear's non-linear frequency response by applying a triangular filterbank (similar to the one seen in Figure 4.7) to the signal's power spectrum. Once we have MFCCs for both the original and reconstructed signals, we can use the ℓ_2 distance between the MFCCs as an effective proxy for perceptual distance.

We use 4 MFCC filterbanks of sizes 8, 16, 32, and 128, to allow for both coarse and fine-grained frequency differentiation. Our final perceptual loss is

the average of the 4 MFCC distances:

$$P(x, y) \;=\; \frac{1}{4} \sum_{i=1}^{4} \ell_2(MFCC_i(x),\; MFCC_i(y))$$

- *Quantization loss.* Because softmax quantization is a continuous approximation to quantization, it is possible for a neural network to learn how to "get around" its constraints. In fact, the network will almost always learn how to generate values outside the intended quantization bins, as long as there is no additional penalty for such behavior. Thus, we define a loss function favoring soft assignments which are very close to one-hot vectors:

$$Q(c) \;=\; \frac{1}{256} \sum_{i\,=\,0}^{255} [\, (\sum_{j\,=\,0}^{N-1} \sqrt{c_{i,j}}) \;-\; 1.0\,]$$

  $Q(c)$ is zero when all 256 encoded symbols are one-hot vectors, and nonzero otherwise.

- *Entropy loss.* The *entropy* of a probability distribution is the average number of bits required to represent a symbol from that distribution. It is computed as follows:

$$entropy(P) \;=\; \sum_{P} -p_i \, log_2(p_i)$$

  where P is a probability distribution and p_i are its elements. At the extremes, a uniform probability distribution with N equally probable outcomes has an entropy of $log_2 N$ bits, and a sparse probability distribution with only one possible outcome has an entropy of 0 bits (since it is entirely predictable). A probability distribution between these two extremes has an entropy between these two bounds; for example, a rigged coin with probabilities 75% heads and

25% tails has an entropy of 0.81 bits (in contrast to a regular coin, which has an entropy of 1 bit).

We can directly compute a histogram h over the quantized symbols by averaging all of the soft assignments the encoder generates. This histogram can be thought of as a probability distribution, specifying how often each quantized value appears. The estimated entropy of the encoder is the entropy of this histogram:

$$E(c) \;=\; \sum_{h\,=\,histogram(c)} -h_i \; log_2(h_i)$$

4.2.2 Training Process

Our training process takes place in two stages:

1. *Quantization off.* The network is trained without quantization. (Because the quantizer is turned off, only the ℓ_2 and perceptual losses are enabled.) After 10 epochs, the quantization bins are initialized using K-means clustering, $\lambda_{entropy}$ is set to some initial value $\tau_{initial}$, and quantization is enabled. We found that this small "pre-training" of the network gave much better and more stable results than turning quantization on from the start.

2. *Quantization on.* The network is trained for a number of epochs, targeting a certain bitrate. At the end of each epoch, the average bitrate of the encoder is estimated (taking into account entropy coding and quantization), using the

following formula:

$$\begin{aligned} bitrate &= (windows/sec) \ * \ (symbols/window) \ * \ (bits/symbol) \ bps \\ &= \frac{16000}{512-32} \ * \ 256 \ * \ E(c) \ bps \end{aligned}$$

If the encoder's bitrate is higher than the target, then $\lambda_{entropy}$ is increased by a small value τ_{change}; if the encoder's bitrate is lower than the target, then $\lambda_{entropy}$ is decreased by τ_{change}. This removes the need to manually find the optimal $\lambda_{entropy}$ for each target bitrate.

During training, we also slowly lower the network's learning rate from an initial value $\alpha_{initial}$ to a final value α_{final}, using cosine annealing [55] [56]. We repeat this training process for each different bitrate we want to target; for example, if we want to target 4 different bitrates, we train 4 total networks.

4.2.3 Hyperparameters

For all experiments, we used the following hyperparameters. We bold the ones which required any significant manual tuning:

Hyperparameter	**Value**
K (number of channels in network blocks)	64
F (convolution filter size)	9

$\sigma_{initial}$ (initial softmax temperature)	500
N **(number of quantization bins)**	32
$\alpha_{initial}$ **(initial learning rate)**	0.025
α_{final} **(final learning rate)**	0.01
λ_{mse} (ℓ_2 loss weight)	30.0
$\lambda_{perceptual}$ **(perceptual loss weight)**	5.0
$\lambda_{quantization}$ (quantization loss weight)	10.0
$\tau_{initial}$ (initial entropy loss weight)	0.5
τ_{change} (entropy loss adjust amount)	0.025
Optimizer	Adam
Batch size	128
# epochs before quantization is turned on	10
# epochs of training total	300

Training a single network takes a little over 13 hours on one GeForce GTX Titan 1080 Ti, using Keras with the TensorFlow backend (which we found to be significantly faster for 1D convolution than the Theano backend).

4.3 Results

We evaluated our ANN-based wideband speech coder against the current AMR-WB standard at several different bitrates, using both objective metrics and subjective listening tests.

4.3.1 Objective Quality Evaluation

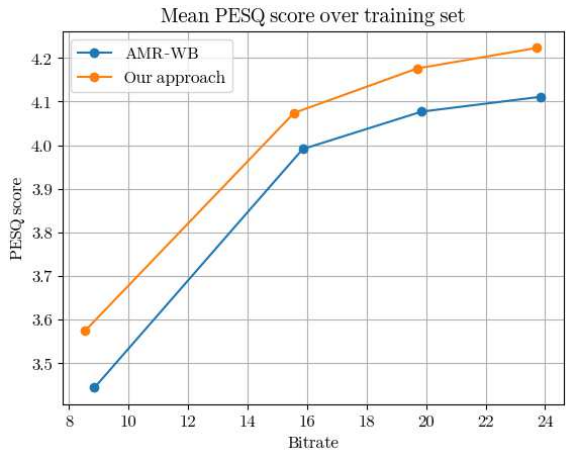
Subjective human listening tests are the preferred way to evaluate audio quality, but comprehensive tests are often very expensive and time-consuming to set up in detail. Thus, researchers have developed *objective* quality measures which determine audio quality computationally. The goal of an objective quality measure is to provide as high a correlation to human listening tests as possible.

The most popular objective speech quality measure is PESQ (Perceptual Evaluation of Speech Quality) [57], developed by ITU-T. PESQ takes in the original speech signal as well as a processed version, and returns an estimated Mean Opinion Score (MOS). An MOS score ranges from 1 to 5, with higher values indicating higher quality:

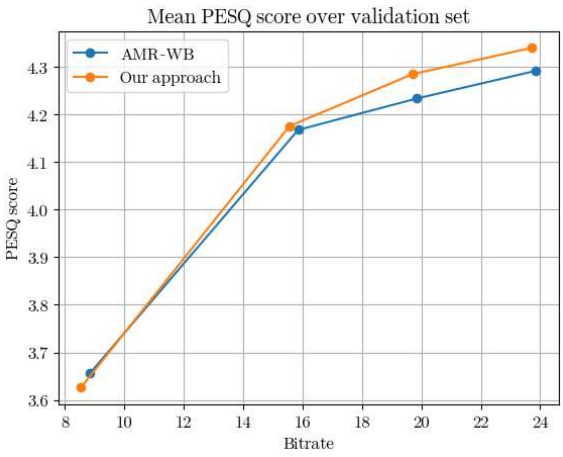
Score	**Meaning**
5	Excellent
4	Good
3	Fair
2	Poor
1	Bad

In our case, the processed version of the signal is simply the speech signal after compression and decompression.

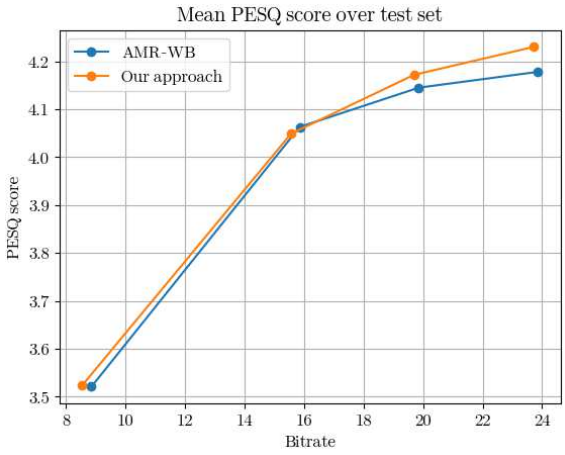
We evaluate the average PESQ of our speech coder versus the AMR-WB standard around 4 different target bitrates, on the full training, validation, and test sets we constructed. The results are shown in Figure 4.8. At higher bitrates, our speech



(a) Training set



(b) Validation set



(c) Test set

Figure 4.8: Mean PESQ of our speech coder, compared with AMR-WB.

coder outperforms AMR-WB by a bit, and at lower bitrates, the two perform essentially on par with each other. We note that the gap is much bigger on the training set than on the validation or test sets, indicating some possible overfitting (note that we did not use any form of dropout or weight regularization). For completeness, we list the specific bitrates and PESQ scores in a table on the following page:

Dataset	AMR-WB		Our approach	
	Bitrate	**Mean PESQ**	**Bitrate**	**Mean PESQ**
Training set	8.85	3.444	8.53	3.574
	15.85	3.991	15.56	4.074
	19.85	4.077	19.70	4.176
	23.85	4.111	23.71	4.223
Validation set	8.85	3.657	8.53	3.626
	15.85	4.167	15.56	4.175
	19.85	4.233	19.70	4.284
	23.85	4.291	23.71	4.339
Test set	8.85	3.521	8.53	3.523
	15.85	4.063	15.56	4.049
	19.85	4.145	19.70	4.172
	23.85	4.178	23.71	4.23

In summary, as far as these objective quality tests go, our learned speech coder is on par with or better than the AMR-WB standard.

4.3.2 Complexity Evaluation

We evaluate the average time taken by our speech coder to encode and decode a single window, on both CPU and GPU, and show the results in the table below. (The computational complexity of our speech coder is independent of bitrate, since the same network architecture is used for every bitrate.)

Processor	Encoder	Decoder	Total
CPU *(Intel i7-4790K @ 3.8Ghz)*	10.52ms	10.90ms	21.42ms
GPU *(GeForce GTX 1080 Ti)*	2.43ms	2.35ms	4.78ms

Our speech coder runs in realtime (under 30ms for combined encode and decode) on both the CPU and the GPU that we tested on. This is without any optimizations beyond those already provided by TensorFlow and Keras, and without any form of model compression or knowledge distillation. Despite this promising result, it's important to note that real speech coders will run on processors much slower than the 3.8GhZ Intel CPU we used; thus, future work in our direction should focus on speeding up computation while preserving the same level of speech quality.

We also give the number of trainable parameters in our model below:

Module	# of parameters
Encoder	742,448
Decoder	816,399
Total	1,558,847

4.3.3 Subjective Quality Evaluation

We were not able to conduct subjective quality tests in the time frame for this thesis. However, we plan to for a future publication.

We will set up a series of small subjective quality tests using the Amazon Mechanical Turk platform. The specific test we will use is a simple preference test.

The listener will be presented with a clean speech signal, and two processed versions: one compressed with AMR-WB, and the other compressed with our method. Then, the listener will be asked to pick which method he or she prefers.

20 speech files will be selected from the test set and compressed with both AMR-WB and our speech coder, at the same 4 bitrates as before. Each of these 80 variations will be shown to listeners on Mechanical Turk, and they will be asked to give their subjective opinions.

4.3.4 Ablation Study

We also plan to perform a simple ablation study, evaluating the effect of our network architecture's individual components. Namely, we will fix the network's target bitrate to about 20kbps, and test the effect each of the following changes has on the average PESQ:

- Removing all shortcut connections
- Removing the perceptual loss
- Replacing dilated convolutions with regular convolutions
- Replacing the downsample/upsample blocks with fixed linear downsampling/upsampling
- Removing intermediate residual blocks

Chapter 5: Conclusion

This thesis has developed a proof-of-concept applying deep neural networks (DNNs) to speech coding, with very promising results. Our wideband speech coder is learned end-to-end from raw speech signals, with no hand-engineered features anywhere to be seen; nevertheless, it manages to be highly competitive with current standards.

The main future direction is obvious: improving quality while decreasing computational complexity. The gap between our approach and the current standards can always be increased further, and while our DNN-based coder already runs in realtime on a modern desktop CPU, this is still a far cry from running on embedded systems or cellphones. Model compression, transfer learning, and clever architecture designs are all valid areas to explore in the future, so we can achieve this.

An observation: except for our model's perceptual loss, *nothing at all* about our approach is specific to speech. Our training process and network architecture are problem-agnostic; the data we feed them is everything. Thus, another future direction arises: feed in different 1D data, such as music, and we can learn a new music format. Change the 1D convolutions to 2D, and we can learn a new image format. The real beauty of the approach is how easily it could extend to different domains; so finally, we urge others to follow up on our work... and extend it.

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Shigeo Morishima, H Harashima, and Y Katayama. Speech coding based on a multi-layer neural network. In *Communications, 1990. ICC'90, Including Supercomm Technical Sessions. SUPERCOMM/ICC'90. Conference Record., IEEE International Conference on*, pages 429–433. IEEE, 1990.

[3] Milos Cernak, Alexandros Lazaridis, Afsaneh Asaei, and Philip N Garner. Composition of deep and spiking neural networks for very low bit rate speech coding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(12):2301–2312, 2016.

[4] J Jiang. Image compression with neural networks–a survey. *Signal Processing: Image Communication*, 14(9):737–760, 1999.

[5] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. *arXiv preprint arXiv:1608.05148*, 2016.

[6] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[7] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[8] Marvin Minsky and Seymour Papert. Perceptrons. 1969.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[11] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[12] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Sepp Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, diploma thesis, institut für informatik, lehrstuhl prof. brauer, technische universität münchen, 1991.

[14] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[15] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[16] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[17] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[18] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[19] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[21] Klaus Greff, Rupesh K Srivastava, and Jürgen Schmidhuber. Highway and residual networks learn unrolled iterative estimation. *arXiv preprint arXiv:1612.07771*, 2016.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[24] CCITT Recommendation. Pulse code modulation (pcm) of voice frequencies. *ITU*, 1988.

[25] L Fernández Gallardo and Sebastian Möller. Phoneme intelligibility in narrowband and in wideband channels. In *Annual German Congress on Acoustics (DAGA)*, 2015.

[26] Global mobile Suppliers Association et al. Mobile hd voice: Global update report. *Information Papers, Mar*, 2014.

[27] Recommendation G.722.1. Low-complexity coding at 24 and 32 kbit/s for hands-free operation in systems with low frame loss, 2005.

[28] Bruno Bessette, Redwan Salami, Roch Lefebvre, Milan Jelinek, Jani Rotola-Pukkila, Janne Vainio, Hannu Mikkola, and Kari Jarvinen. The adaptive multirate wideband speech codec (amr-wb). *IEEE transactions on speech and audio processing*, 10(8):620–636, 2002.

[29] Robert D Dony and Simon Haykin. Neural network approaches to image compression. *Proceedings of the IEEE*, 83(2):288–303, 1995.

[30] Ashok K. Krishnamurthy, Stanley C. Ahalt, Douglas E. Melton, and Prakoon Chen. Neural networks for vector quantization of speech and images. *IEEE journal on selected areas in Communications*, 8(8):1449–1457, 1990.

[31] Lizhong Wu, Mahesan Niranjan, and Frank Fallside. Fully vector-quantized neural network-based code-excited nonlinear predictive speech coding. *IEEE transactions on speech and audio processing*, 2(4):482–489, 1994.

[32] Anssi Rämö. Voice quality evaluation of various codecs. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 4662–4665. IEEE, 2010.

[33] Dong-Chul Park and Young-June Woo. Weighted centroid neural network for edge preserving image compression. *IEEE transactions on neural networks*, 12(5):1134–1146, 2001.

[34] Adnan Khashman and Kamil Dimililer. Image compression using neural networks and haar wavelet. *WSEAS Transactions on Signal Processing*, 4(5):330–339, 2008.

[35] Milos Cernak, Blaise Potard, and Philip N Garner. Phonological vocoding using artificial neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4844–4848. IEEE, 2015.

[36] Garrison W Cottrell, Paul Munro, and David Zipser. Learning internal representations from gray-scale images: An example of extensional programming. In *Ninth annual conference of the cognitive science society*, pages 462–473, 1987.

[37] Aran Namphol, Mohammed Arozullah, and Steven Chin. Higher order data compression with neural networks. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 1, pages 55–59. IEEE, 1991.

[38] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[39] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[40] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[41] Michael Mathieu, Camille Couprie, and Yann LeCun. Deep multi-scale video prediction beyond mean square error. *arXiv preprint arXiv:1511.05440*, 2015.

[42] Alexey Dosovitskiy and Thomas Brox. Generating images with perceptual similarity metrics based on deep networks. In *Advances in Neural Information Processing Systems*, pages 658–666, 2016.

[43] George Toderici, Sean M O'Malley, Sung Jin Hwang, Damien Vincent, David Minnen, Shumeet Baluja, Michele Covell, and Rahul Sukthankar. Variable rate image compression with recurrent neural networks. *arXiv preprint arXiv:1511.06085*, 2015.

[44] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.

[45] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2016.

[46] Nick Johnston, Damien Vincent, David Minnen, Michele Covell, Saurabh Singh, Troy Chinen, Sung Jin Hwang, Joel Shor, and George Toderici. Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks. *arXiv preprint arXiv:1703.10114*, 2017.

[47] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc Van Gool. Soft-to-hard vector quantization for end-to-end learned compression of images and neural networks. *arXiv preprint arXiv:1704.00648*, 2017.

[48] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[49] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[50] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1874–1883, 2016.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[52] John S Garofolo, Lori F Lamel, William M Fisher, Jonathan G Fiscus, David S Pallett, Nancy L Dahlgren, and Victor Zue. Timit acoustic-phonetic continuous speech corpus. *Linguistic data consortium*, 10(5):0, 1993.

[53] Lindasalwa Muda, Mumtaj Begam, and Irraivan Elamvazuthi. Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques. *arXiv preprint arXiv:1003.4083*, 2010.

[54] Vibha Tiwari. Mfcc and its applications in speaker recognition. *International journal on emerging technologies*, 1(1):19–22, 2010.

[55] Ilya Loshchilov and Frank Hutter. Sgdr: stochastic gradient descent with restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[56] Xavier Gastaldi. Shake-shake regularization. *arXiv preprint arXiv:1705.07485*, 2017.

[57] Antony W Rix, John G Beerends, Michael P Hollier, and Andries P Hekstra. Perceptual evaluation of speech quality (pesq)-a new method for speech quality assessment of telephone networks and codecs. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 2, pages 749–752. IEEE, 2001.

[58] Flávio Ribeiro, Dinei Florêncio, Cha Zhang, and Michael Seltzer. Crowdmos: An approach for crowdsourcing mean opinion score studies. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 2416–2419. IEEE, 2011.