Using convolutional autoencoders to improve classification performance

Bachelor's Thesis in Artificial Intelligence

Jordi Riemens
s4243064

July 8, 2015

Supervisors: Marcel van Gerven[†], Umut Güçlü[†]

Department of Artificial Intelligence
Radboud University Nijmegen

[†] Donders Institute for Brain, Cognition and Behaviour, Radboud University Nijmegen

Using convolutional autoencoders to improve classification performance

Jordi Riemens

July 8, 2015

Abstract

This thesis combines convolutional neural networks with autoencoders, to form a convolutional autoencoder. Several techniques related to the realisation of a convolutional autoencoder are investigated, and an attempt is made to use these models to improve performance on an audio-based phone classification task.

1 Introduction

Speech recognition research has long been dominated by research into hidden Markov models (for example, see Lee and Hon (1989); see also Rabiner (1989) for a theoretical review). Hidden Markov models are probabilistic constructs that work on observed time series (in this case, speech recordings), and attempt to retrieve the state variables (actual phones pronounced) that caused these observations. However, in recent years, convolutional neural networks are increasingly taking over hidden Markov models on grounds of classification performance (see Hinton et al. (2012), Sainath et al. (2013b); see Tóth (2014) for state-of-the-art performance). Hybrid approaches are also possible (Abdel-Hamid et al., 2012).

Convolutional neural networks come from the field of image classification, where they are the dominant and best-performing technique (e.g., see Lawrence et al. (1997), Cireşan et al. (2010)), and modified variants of them continue to achieve state-of-the-art performance (e.g., Krizhevsky et al. (2012)). Additionally, they appear biologically plausible to some extent, as some of their characteristic properties are indeed found inside the brain, such as receptive fields that increase in size (Güçlü and van Gerven, 2014) and the use of a hierarchical, feature-based representation (Kruger et al., 2013).

However, convolutional neural networks require supervised training, which in turn requires painstakingly labelled data. Autoencoders, on the other hand, are methods of learning higher-level representations of a data set in an *unsupervised* manner, requiring only the data (which is abundant), and not the labels (which need to be manually matched to the data points). Many variants exist, including the contractive autoencoder (Rifai et al., 2011), the sparse autoencoder (Ng, 2011), the denoising autoencoder (Vincent et al., 2008) and, importantly, the stacked autoencoder (Bengio et al., 2007). The latter can also be combined with the other techniques, such as in a stacked denoising autoencoder (Vincent et al., 2010). Since the autoencoder is an unsupervised network architecture aimed at learning representations, and convolutional neural networks intrinsically learn hierarchical feature-based representations, it seems natural to combine these techniques, to attempt to create an unsupervised hierarchical feature-based representation learner.

The combination between convolutional neural networks and autoencoders has been made before (Masci et al. (2011), Tan and Li (2014), Leng et al. (2015)), though not frequently, as autoencoder research tends to be based on conventional neural networks (without convolution). Most convolutional autoencoders are applied to visual tasks, as that is the origin of convolutional techniques. Hence, the application of convolutional autoencoders to audio data is rare (though it has been done, e.g. Kayser and Zhong (2015)). As this work tries to do exactly this, it does not tread into *completely* new territory, but it is still relatively novel.

Apart from investigating feasibility and techniques for constructing and training convolutional autoencoders, this work also attempts to utilise these models for phone classification, a supervised learning task that is part of speech recognition. Phones are the basic speech sounds that can be found in a language, such as the sounds [i] or [n] found in English (among many other languages). As data, we use the TIMIT data set (Garofolo et al., 1993), a 'classic' within speech recognition. The objective is to correctly classify small audio fragments of phones into the right category.

State-of-the-art performance for this task is in the order of 75-85% accuracy (Hinton et al. (2012), Tóth (2014)), and relative improvements of several percentage points are worthy of publication. However, this work will not focus on 'beating' the state of the art, but will instead investigate whether autoencoders can be utilised to improve classification networks for this task. It has already been shown that autoencoders in general can indeed increase classification performance by pretraining (see e.g. Masci et al. (2011), Tan and Li (2014)), which uses an adequately-trained autoencoder to initialise weights of the classifying network. However, to my knowledge, convolutional autoencoders have not been applied to a phone classification task.

In this thesis I will describe the complete process from data set to autoencoder-aided classification, and detail the considerations made during the course of the project. Section 2 will detail the preprocessing of the TIMIT data set, Section 3 describes how to train a reference classification network, Section 4 discusses the convolutional autoencoder and how to use them for classification, Section 5 describes the main experiments of this thesis, Section 6 discusses the results of these experiments, and it is followed by my conclusions in Section 7 and a discussion section, Section 8. Importantly, there are two appendices. Appendix A contains the full results for the experiments described in Section 5, and Appendix B extensively details the mathematics behind convolutional neural networks, as well as convolutional autoencoders. Mathematical detail is therefore left out of the main text of this thesis, and deferred to this appendix.

2 Data preprocessing

Whether we work on supervised classification tasks or unsupervised autoencoder models, we need data to feed into our network. Therefore, in this section I discuss the preprocessing we used to convert our initial TIMIT data set to a more handleable form, which we thereafter use as input to our networks. (*Disclaimer: the work described in this section (in particular the exchange, realisation and testing of various ideas on preprocessing) was performed in a group of three, consisting of Churchman (2015), Kemper (2015) and myself.*)

2.1 Data selection and reshaping

Given that we use the TIMIT data set from Garofolo et al. (1993), we start with a number of .wav audio files of English spoken sentences recorded by a number of native speakers from different dialects, labelled with timestamps per phone and per word. Given that we are doing a phone classification task, we slice our data on individual phones, i.e. we divide each sentence up into its constituent phones. These will eventually be converted into the actual data points fed into our network.

The TIMIT data set contains both sentences that were recorded by all speakers of the data set, and sentences that were only recorded by one speaker. To avoid over-representation of certain phones or phone combinations, we focus only on the latter category, where every sentence is only present in our data set once (as in Abdel-Hamid et al. (2012)). Sentences that were recorded by all speakers are discarded.

We then apply the mapping proposed by Lee and Hon (1989) to the phone labels. We discard glottal stops ("q") altogether, and several phones are pooled together. This reduces the number of different labels from 61 to 48. Furthermore, within these 48 labels, some labels are put into label groups, where within-group confusions do not count as an error. In other words, our network will have 48 output units, and can therefore assign phones to 48 classes, but effectively there are less distinct phone categories (namely, 39).

Now that we have our data slices and their labels, we ensure all of our data points are the same length, as this is required for a standard convolutional neural network (though 'variable-size' convolutional neural networks exist, see e.g. LeCun and Bengio (1995)). We make our data points the same length simply by zero-padding our slices on both sides, to the largest phone. This prevents any distortion of the sound, and any loss of information or sound quality. This does, however, dramatically increase the size of the data set. This effect is amplified by the fact that, while most phones are quite short, a few phones in the TIMIT data set are extremely long (e.g., certain 'silence' phones), so all other (shorter) phones in the data set are significantly lengthened. The consequence is that the resulting preprocessed data set is far too large to hold in even large amounts of memory. To combat this problem, we discard the 5% longest phones. The largest *remaining* phone, then, is short enough to zero-pad to, such that the whole zero-padded data set *can* readily be preprocessed given a reasonable amount of memory.

2.2 Time-frequency representations

2.2.1 Type of representation

Now, we have equal-sized slices containing phones in .wav format, i.e. their amplitude waveforms. However, it is hard to use the full power of convolutional neural networks for these kinds of 1D signals.

This is the case because the convolutional aspect, which by definition provides translational invariance for learnt features, can only provide *temporal* invariance in these waveforms (simply because time is the only dimension being varied along the axis). In particular, this means that features are *not* pitch-invariant. Therefore, a feature encoding for [a] would need different learnt representations for a low-pitch (e.g., recorded by a male speaker) [a] phone and a high-pitch (e.g., female speaker) one. Given that humans recognise these sounds as being the same phone, we may find a 1D representation that only provides temporal invariance undesirable.

Instead, what we want is a representation in which convolutional layers can give us both temporal and frequency invariance. This naturally brings us towards a 2D representation of sound, with time on one axis, and frequency on the other. There are a couple of major, often-used candidate representations: the short-time Fourier transform (STFT), the mel-frequency cepstrum (MFC) (see e.g. Zheng et al. (2001)) and what we will refer to as the 'gammatonogram', or gammatone-based spectrogram (Patterson et al. (1992), Patterson et al. (1987)). We have tried all three methods and settled for the STFT approach, but I will nevertheless describe the other two methods, and explain why these were not chosen.

Gammatonograms are similar to spectrograms, but are constructed to share certain properties with human audio representation in the ear and nervous system, in particular with the cochlea and basilar membrane in the human ear, and tries to simulate the neural activity of the ear's outgoing auditory neurons (Patterson et al., 1992). Specifically, it uses a so-called gammatone filter bank (Patterson et al., 1987) to convert audio into a number of channels concerning motion of this basilar membrane, and then it uses a 'transducer' simulation that converts this into a pattern of neural activity sent out by the cochlea to the brain. A particularly relevant property of gammatonograms, for our purposes, and the reason we tried it out, is that in the human ear, low- and medium-frequency sounds are represented with higher precision than very high-frequency sounds, and in particular this includes speech sounds, the object of our classification task. In contrast, the STFT represents all frequency ranges equally precisely. Therefore, gammatonograms could have higher precision for speech than STFT, and thereby achieve better phone classification performance. Unfortunately, gammatonograms of decent resolution cost too much memory to fit inside reasonable amounts of memory, and as a result we could not practically test this approach on an actual network.

Mel-frequency cepstra (MFCs) (see Zheng et al. (2001)) are based on the short-time Fourier transforms described next, but transform the found Fourier spectra to MFCs by first mapping the Fourier coefficients to the mel scale, (in some common versions) taking their logarithms, and then taking the (discrete) cosine transform of the resultant list of 'mel frequencies'. For a MFC-based cepstrogram (i.e., including time as a dimension), one divides the audio signal into small, overlapping time windows (often using a Hamming window), and computes the MFC for each window.

The crucial component of these MFCs is the application of the mel scale, which is constructed to mimic subjective human hearing experience, by rescaling the Hertz scale of frequencies to 'mel frequencies'. For mel frequencies, when two notes are perceived to be equally 'far' from each other, they always have a difference of a fixed number of 'mels', regardless of the actual notes in question (only dependent on the perceived distance between the two notes). This does not hold for the Hertz scale, which works multiplicatively: the note one octave up from 440 Hz is 880 Hz, whereas one octave below 440 Hz is the note of 220 Hz. Therefore, a fixed perceived distance of one octave is not an equal distance in Hertz (440 Hz. vs. 220 Hz in the previous example), whereas the difference between these notes in mels is equal. This means, again, that MFCs represent lower and medium frequency ranges, including speech, with more precision than STFTs. Furthermore, MFCs do not have the memory problems of gammatonograms, as MFCs are directly based on the Fourier transform also used by STFTs.

The short-time Fourier transform (STFT; used in e.g., Abdel-Hamid et al. (2012)), similarly to the MFCs, divides the audio fragment into small, overlapping time windows, using a Hamming window function, and simply takes the (discrete) Fourier transform for each window. As mentioned before, this approach represents all available frequency ranges with equal precision, which might not be ideal given that we mostly only use low and medium frequency ranges.

In the decision process, we compared the STFT and MFC approaches using the classification of practical networks. In these tests, STFT-based networks appeared to perform better than MFC-based networks, conflicting with the aforementioned argument. We have also tried removing the final discrete cosine transform from the MFC algorithm, to try if only using the log-mel scale improved performance, but it did not perform better than with regular MFCs. However, it should be noted that it is entirely possible that the apparent performance difference was only due to the particular networks or parameters used in our tests, and that more complex MFC-based networks, or simply versions with a different architecture or different parameter settings, do perform better than STFT-based networks. Regardless of whether this is true or not, our tests pointed out that STFT-based networks appeared to run better, so we settled for the STFT approach.

2.2.2 Parameter settings

Apart from classification performance, another important measure that we can evaluate our audio representations by is reconstruction performance. In other words, if we invert the short-term Fourier transform to reconstruct our original audio signal, then we want that reconstruction to be as good as possible; after all, we eventually want to work with autoencoders that also try to reconstruct the original audio fragment. We evaluate reconstruction quality by calculating the cross-correlation between the original signal and the reconstruction.

To improve reconstruction quality, we can tweak certain parameters related to the STFT transformation: the window size, the amount of overlap be-

Table 1: Cross-correlations between original input and reconstructions, for different numbers of frequency bins, and when discarding or keeping phase information after Fourier transform.

	101 bins	201 bins	301 bins
available	0.67	0.92	N/A
discarded	0.36	0.91	0.98

tween successive windows, and the desired amount of frequency bins. These all directly affect the size of the resulting spectrogram: the former two affect the amount of time windows, the horizontal axis, and the latter equals the number of frequency bins, the vertical axis. There is a significant trade-off related to these dimensions: larger spectrograms yield better reconstructions, but larger spectrograms are also computationally more intensive, thus taking longer to train. Thus, we make the compromise of choosing the parameters leading to the smallest spectrograms that still gives a high-quality reconstruction.

For the time dimension parameters, we base ourselves on the literature (Abdel-Hamid et al., 2012) and then slightly tweak our parameters for better reconstruction. The final parameters used in preprocessing are a 17-millisecond ($\frac{1}{60}$ second) Hamming window size, with 7.5-millisecond overlaps, giving us 16 time windows per phone, which should still contain quite enough time information for our purposes (i.e., leaving enough time points to convolve over) while not being overly large.

For the frequency dimension, we varied the number of frequency bins and looked at the cross-correlation described above (see Table 1). When using a high number of frequency bins, such as 301, the reconstruction becomes near perfect. However, when using a relatively low number of frequency bins, the reconstruction can become deplorable and (when played back) unintelligible. As stated above, we make a compromise between data size (which influences computation time) and both reconstruction quality and classification performance, and chose to use 201 frequency bins for our spectrograms.

2.3 Data transformation

Lastly, we need to consider what exact numbers we put in the matrices that form our data set. The short-term Fourier transform returns a spectrogram containing complex numbers, but convolutional neural networks are not designed to work with complex numbers. As such, we need to convert our complex numbers into one or more real numbers, with which our network *is* able to work.

Complex numbers have two main canonical representations, both in 2D. Firstly, we have polar coordinates, where a complex number is represented as an amplitude, and its phase angle with the real number line. It is possible to feed the network data with multiple channels per unit (cf. RGB images with red, green and blue channels per pixel). Thus, in our context, we can test performance with only amplitude information, but we can also consider adding the complex numbers' phase information to our final data set. Note that using only phase information is not an option, as phase in undefined for zeros (which are definitely present due to zero-padding). Thus, the network would not know how to distinguish between meaningless data (padding) and relevant data. Hence, if we use the polar coordinates of our complex numbers, we need to choose between using only the amplitude, or using both the amplitude and phase.

However, if we look at the improvement in cross-correlations between using only the amplitude, and using both amplitude and phase (see Table 1), we see that phase information only yields a significant performance increase for lower numbers of frequency bins (e.g., 101), but that phase information is largely made redundant by simply increasing the data's resolution. As phase information did not give the network any significant increase in classification performance either, it can be discarded, and using only the absolute value suffices as representation for the complex coefficients from the spectrogram (if polar coordinates are used).

Secondly, we have Cartesian coordinates, where a complex number consists of a real and a complex part. However, using this representation is not convenient to reason with, as a loud sound could, for example, either have a strongly positive, strongly negative or near-zero real (or complex) part, whereas it always has a high amplitude and some 'random' phase. Hence, for representing the whole number, the Cartesian representation is not preferred. However, we find that using only the real parts of the complex numbers improves the reconstruction relative to using the polar representation described above. This comes at a cost of a few percentage points of classification performance, but given that we ultimately want to be able to faithfully reconstruct our original audio inputs, we value the reconstruction performance increase over the slight classification performance decrease, so we settle for simply including the real parts of the spectral coefficients in our preprocessed data set.

A minor, but noteworthy transformation we also apply is data normalisation. From every spectrogram, we subtract the average spectrogram (per element), leading to zero-centred spectrograms. Given that convolutional filters work multiplicatively, they work best with zero-centred data, even though non-zero-centred data can is still usable (e.g., by accordingly adjusting the biases, if the network considers this to be useful).

One final addition we have tried making to our data set was adding Δ and $\Delta\Delta$ values, the frame-by-frame first-order and second-order temporal derivatives, to our data in additional channels.

This cannot influence reconstruction performance, as these values can be readily calculated simply from the regular data, but it can improve classification performance. The reasoning behind this is that, even though these temporal derivatives can simply be found inside the network by learning a simple convolutional filter (or two sequential filters for the second-order ones), by simply adding the deltas to our data these filters would not be necessary, possibly allowing for *more* higher-level layers to reason about this potentially meaningful information. However, after successfully implementing the generation of these deltas, no significant classification performance was found in our experiments, so they were left out.

3 The forward network

To understand how convolutional auto-encoders behave and should be constructed, we first need to know how regular convolutional neural networks are constructed, and how they behave. In particular, it is important to have a reference network architecture with reasonable classification performance, which we can later use for reference when dealing with autoencoders.

State-of-the-art performance on the TIMIT data set, as discussed in Section 1, entails an accuracy in the order of 75-85%, but given that there are as many as 39 effective classes (i.e., chance level is below 3%), we set as our objective a network that reaches 50% accuracy (or more). After all, reaching high classification performance is not the main objective of our forward network.

The reason we need a 'forward' network (i.e., from spectrogram to class label), is that such a forward network will already have learnt some kind of meaningful representation of our data set, with sufficient predictive power to achieve such a level of performance. Given that autoencoders are constructed for the purpose of learning representations of the data, this forward network will provide us with a working example of the type of architecture (e.g., configuration, size, number of filters needed) that can fit these types of representations. Furthermore, it will serve as a reference point for classification performance, with which we can compare the performance of our own networks, when we eventually use autoencoders for classification.

In this section I discuss the chosen architecture of our forward network, as well as several implementational details of our networks. (*Disclaimer: the work described in this section (in particular the exchange, realisation and testing of various ideas on architectures, implementations and efficiency-related issues) was performed in a group of three, consisting of Churchman (2015), Kemper (2015) and myself.*)

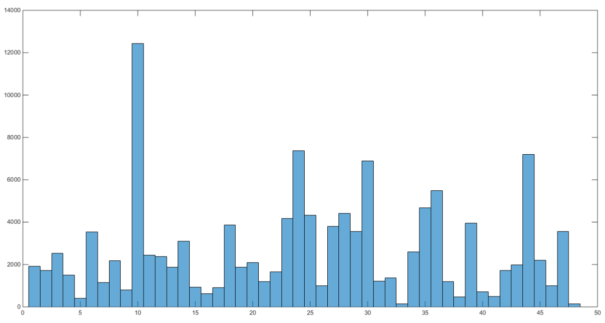


Figure 1: Number of occurrences of different phones in the TIMIT data set. The top peak is a 'silence' phone.

3.1 Implementation

Before we can investigate these forward networks, we first need a way to train convolutional neural networks in general. For this, we use MatConvNet, a convolutional neural network framework in MATLAB from Vedaldi and Lenc (2014). However, we have modified MatConvNet to better fit our purposes (see also Section 3.2 and Section 4).

One rather significant modification to the normal training mechanism followed the observation that the distribution of phone labels within the TIMIT data set is very unbalanced. This is caused by the fact that some phones are naturally abundant in some languages, whereas others are rare (or nonexistent). For example, in English, phones corresponding to E or N might be very common, whereas the phone for J is rare. Since the TIMIT data contains regular English sentences, this imbalance carries over to our data set. As a result, some networks adopt the rather oversimplified strategy of reporting that every phone was a silence (the most common phone in the TIMIT data set), accounting for approximately 7% of the data set and therefore leading to 7% accuracy.

To combat these kinds of strategies, we artificially remove this imbalance in the data set by using stratified (re-)sampling every epoch, such that the phone label distribution becomes uniform. If a class has more than 500 associated data points, 500 of these are randomly selected at the beginning of each epoch, and only these will be included in the training set (the same happens with the test set, but here the number of examples per class is chosen to be 200). If, however, a class has less than 500 associated data elements, we first repeat the entire set of examples for that class as many times as possible (within the assigned 500 or 200 spots), and fill the remainder of space up with randomly sampled data points. For example, at the start of a training phase, if there are 2000 examples of a 'silence' phone, we will randomly select 500 of these to form the 'silence' part of our training set. If, however, there are only 60 examples of a 'J' phone, every spectrogram from this category is represented in the training set 8 times, and 20 of them will be represented 9 times during that epoch. As a result, in the effective training set, we will find an equal amount of 'silence' phones as 'J' phones. This makes the above strategy of mapping everything to 'silence' no longer viable, as this would give the network a training error merely at chance level. It will also give the network more incentive to focus on rarer phones, as opposed to silences.

Another problem with training convolutional neural networks is that it is not always clear when a neural network has 'completed' training. Normally, when the network's training error does not decrease anymore, one decreases the learning rate and continues training. This allows the network, being 'in the right neighbourhood', to 'fine-tune' itself towards even better performance. Instead of doing this manually, we have implemented AdaGrad (Duchi et al., 2011), an adaptive learning rate annealing algorithm that automatically decreases the learning rate per individual filter (or bias) element on the basis of the sum of squares of the magnitudes of all previous updates to that element. In other words, filter elements that have been updated relatively much will be updated less strongly than before, whereas updates to filters that have not been updated as much will be (relatively) boosted. See Section B.9 for a more mathematical description.

3.2 Efficiency

Now that we have a working implementation, however, given the amount and complexity of our data, a problem quickly arises: training non-trivial networks takes a large amount of time. This is especially problematic since there are many different possible architectures, with many parameters that could be tweaked, but there is no time to test any reasonable portion of architectures and settings. Thus, we base ourselves on literature (see also Section 3.3), but even only comparing certain chosen models takes relatively much time, with some larger networks possibly taking weeks to finish training. All in all, it is important to find ways to improve the efficiency of training these networks.

One straightforward approach to increasing the efficiency of our networks is already implemented and supported by default by MatConvNet, namely training our networks on a GPU. This only works for NVIDIA GPUs supporting NVIDIA CUDA, but we have access to one. Unfortunately, though this did increase training speed by an approximated 20-30%, it was not as large an improvement as we had hoped, given the massive parallellism employed by GPUs.

We achieve another significant efficiency upgrade by programming a 'prefetch' feature in C++, enabling us to train our network in one thread, while another thread loads the next batch of spectrograms. This requires a partial rewrite of the actual MATLAB-based training code of MatConvNet. C++ is used because MATLAB's built-in multithreading functionality is not adequate for our purposes, but MATLAB does support running C++ code as MEX functions. The exact efficiency upgrade depends largely on the relative computing times required for normally loading a batch of spectrograms and the actual training by back-propagation (among others, dependent on network size): for large networks where the majority of computing time is spent doing back-propagation, prefetch will not have a profound effect, but for smaller networks (including our final architecture, see Section 3.3) its effect can range from a 25% efficiency increase (for our chosen architecture, approximately) to 50% (in the ideal case) for small networks.

3.3 Architectures

We can now evaluate the performance of networks in a reasonable amount of time, so it is time to choose an architecture for our forward network. There are uncountable possible architectures, so instead of guessing, we base ourselves on the literature.

Given that most convolutional neural network research is concerned with visual tasks, most advances have been made in this area. However, the architectures suited best for these tasks do not carry over to audio tasks. A big reason for this is that, whereas in visual tasks each dimension (vertical, horizontal) within an image has a similar significance, in audio tasks the dimensions (frequency, time) within a spectrogram have a wholly different meaning.

In particular, this implies that convolution with rectangular or square filters, as is common in networks for visual tasks, might be suboptimal; it might be more beneficial to learn filters that only recognise frequency or time patterns.

Thus, there are several possible general architectures. We could do 2D convolution (which is standard in visual tasks), where our filters convolve over both axes. Time convolution, where the whole frequency range is collapsed into a size of 1 by the first convolutional layer (which therefore must span the entire frequency range) and subsequent layers only convolve and pool over the time axis, is another possibility, as well as frequency convolution, where instead the time range is collapsed and the frequency axis convolved and pooled over.

Note that the 'collapsing' convolutional layer need not be 1×16 (if it collapses time) or 201×1 (if it collapses frequency). Indeed, it could be beneficial to have filters such as 8×16, or 201×3, to be able to capture time patterns inside a small frequency range (instead of, initially, only within one bin) and local variations in frequency (instead of initially only being able to reason with 'snapshots'), respectively.

Different axes of convolution provide different benefits to the network. By the nature of convolu-

Network	Depth	Accuracy
Time convolution	2, 2	48.7%
Frequency convolution, large filters	2, 2	56.4%
Frequency convolution	2, 2	59.1%
Frequency convolution, deeper, fully-padded, pooling overlap	4, 3	61.8%
Frequency convolution, deeper, pooling overlap	3, 4	62.6%

Table 2: Classification accuracies for several well-performing architectures. The depth column indicates the number of convolutional and fully connected layers, respectively.

tional neural networks, the network becomes invariant to translations of the input across the axis being convolved over. Thus, frequency convolution learns a representation concerned with certain (combinations of) time patterns within frequency bands, but is invariant to the pitch or base frequency of these patterns. Conversely, time convolution learns certain frequency patterns, and reasons with how these patterns' activations change over time.

All of these techniques are viable and applicable, but literature suggests frequency convolution is the best, followed by 2D convolution (e.g., see Abdel-Hamid et al. (2013)). However, in the process of discovering a concrete architecture that worked, we also ran our own experiments, also indicating that frequency convolution is superior to time convolution (we did not find a well-performing 2D-convolution network architecture). Test results for some of the most well-performing architectures can be found in Table 2.

Full padding, referred to in this table, is described in Section 4.2. The large filters refer to, for example, a 40×16 filter instead of an 8×16 filter in the first layer, which caused a larger error rate (see also Abdel-Hamid et al. (2014)). Pooling overlap refers to using (in our case) a 5×1 pooling region, instead of a (non-overlapping) 2×1 window, causing elements to be in multiple pooling regions simultaneously. In the literature, our choice for a stride-2 max-pooling layer is also supported: Abdel-Hamid et al. (2014) reports that max-pooling outperforms average-pooling, and that the error rate goes up for higher strides for phone classification. As pointwise non-linearity, we use the rectified linear unit (ReLU), as Zeiler et al. (2013) and Sainath et al. (2013a) report that rectified linear units work better than logistic units (e.g., the hyperbolic tangent and the sigmoid).

We see that, in our tests, frequency convolution performs significantly better than time convolution. Additionally, pooling overlap slightly improves performance, whereas full padding decreases it (but only a little). Interestingly, our final architecture uses a minimal amount of distinct filters per convolutional layer (as few as 8 in the first layer), and still achieves very decent performance (over 60% accuracy, whereas 75-85% is the state of the art). This is helpful if we want to use our forward network as a model for our autoencoder, as too many hidden units cause the network to learn the simple identity function, instead of learning a representation of our data. For this work, the network variant with full padding (with 61.8% accuracy) was chosen to be the forward network, for reasons explained in Section 4.2. See Figure 2 for details on our architecture.

4 Convolutional autoencoders

Now that we have an understanding of regular, forward convolutional neural networks, and have found an architecture with very decent classification performance, it is time to turn our attention to our main objective, the convolutional autoencoder.

4.1 Objective

An autoencoder, in general, is a type of neural network aimed at the unsupervised learning of higher-level representations of data. A supervised neural network attempts to learn to produce certain target responses $\mathbf{T}$ (e.g., a class assignment) for certain inputs $\mathbf{X}$. For an autoencoder, $\mathbf{T}$ is equal to $\mathbf{X}$. In other words, an autoencoder is a model trained such that its output mimics its input as closely as possible, where closeness is commonly defined by the Euclidean error.

This seems like a rather trivial task. However, the difficulty is caused by the fact that the network's hidden layers (in most cases) grow successively smaller in size, before they grow larger once again so that the representation size returns to the original input size. In other words, an autoencoder must learn a compressed representation of the data. It has been found that for regular autoencoders with at most one layer using the sigmoid activation function, and all (other) layers linear, the optimal resulting network is strongly related to the principal components analysis, or singular value decomposition, of the data set (Bourlard, 2000). In other words, regular autoencoders are able to correctly learn a meaningful representation of the data.

4.2 Inverting layers

Regarding autoencoders as a representation learner gives rise to the concept of the encoder-decoder view of autoencoders. Essentially, the autoencoder consists of two parts: the 'encoder', which converts the data into a meaningful, compressed representation, and the 'decoder', which reconstructs the original input from the output of the encoder. The encoder learns an encoding function from our phones to these smaller representations, which the decoder attempts to invert as well as possible. For regular

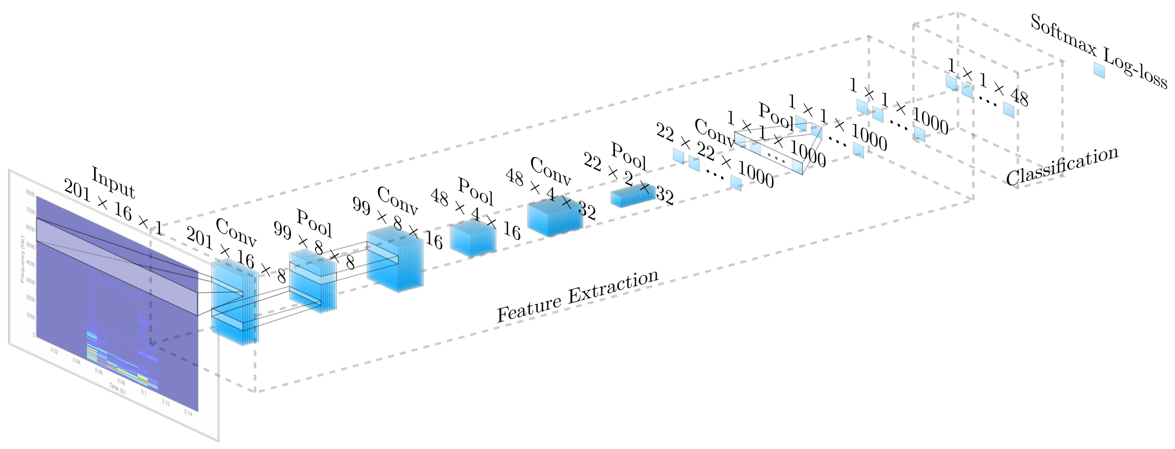


Figure 2: Our finalised full-padding architecture with convolution over the frequency axis. Denoted sizes refer to the *feature map* sizes (i.e., output sizes) of each layer. Each pooling layer and fully-connected layer is implicitly followed by a relu unit, and there is a dropout layer with dropout rate 0.9 between the 'feature extraction' and 'classification' parts. Adapted from Churchman (2015) with permission.

(fully-connected) neural networks, this boils down to using fully-connected layers that invert the *dimension change* of the encoder's layers, one by one: if the an encoder layer has connections from 128 inputs to 1000 outputs, then its corresponding decoder layer is fully-connected with 1000 input units and 128 output units. Weights are then simply learnt with back-propagation.

Note that, in many autoencoder models, there is a certain symmetry between the encoder and decoder. If we can approximately invert every encoding layer in isolation, we can arrange the layer-wise decoders in the reverse order as the encoders they attempt to invert, and append this sequence of decoders to our encoder model. Given that every single encoder-decoder layer pair is approximately reduced to the identity function, it is expected that the entire network also approximates the identity function, or in other words, that is will form a decent autoencoder.

Now, we might be able to simply do this using regular neural networks, as above, but convolutional neural networks work differently than regular neural networks. To be precise, convolutional neural networks are a special case of regular neural networks, so they bring extra limitations with them (in particular, not all regular neural network layers can be used in convolutional neural networks, as in the latter weights are shared and operators are (mostly) applied locally, placing extra constraints on the possible connections).

In general, convolutional neural networks can have convolutional and pooling layers, as well as layers that compute activation functions, layers that implement dropout, etcetera. 1×1 fully-connected layers can simply be decoded using other 1×1 fully-connected layers, but the problem with convolutional autoencoders is that this does not hold in a straightforward way for convolutional and pooling layers.

The reason convolutional and pooling layers cannot be decoded by other instances of themselves is that, whereas they generally decrease the 'image size' (i.e., size of one spectrogram or feature map), they cannot increase it, and neither can other types of layers allowed in convolutional neural networks. Furthermore, the only cases when the image size is *not* decreased by such a convolutional or pooling layer are when its 'stride', or subsampling rate, is 1 (i.e., no subsampling is performed) and when its filters or pooling regions are fully padded (i.e., the total horizontal padding is 1 less than the filter width, and similarly for the vertical direction). However, for pooling layers a subsampling rate e of 1 somewhat defeats their purpose (which, after all, *is* subsampling), so for all practical purposes, we can assume that the representations shrink as we go through the network, but then it cannot grow back to its original size, as no supported type of layer is capable of this. Therefore, our output will have different dimensions than our input, causing the Euclidean error to be undefined and thus the task absolutely impossible without further work.

4.2.1 Convolutional layers

Therefore, we first need to understand how to approximately invert convolutional and pooling layers. Firstly, for convolutional layers we ought to limit ourselves to fully-padded layers with stride 1, as described above. The lack of stride for convolutional layers is not particularly constraining, as most networks use pooling layers for their subsampling. Full padding increases the number of base positions from where convolution can take place, which makes the network train slightly more slowly as higher-level representations are simply larger.

Care must be taken, however, in simply changing all convolutional layers to be fully padded, as what was previously a 1×1 representation fed into fully connected layers is now larger (e.g., 22×2), since the size decrease of convolutional layers was blocked out. This is a problem, because now the (previously) fully-connected units (as they use 1×1 filters) are no longer fully connected.

As all these units in essence code for the same features as in not-fully-padded networks, and we (for classification) are only interested in whether these features are *present* or not (and not in *where* they were present, as the remaining base positions were only artificially introduced by padding), we simply insert another pooling layer before the 1×1 fully-connected units, which pools the entire feature map (in this case, 22×2) back to a 1×1 size per channel. In our tests (see Section 3.3), this conversion from unpadded convolutional layers to their fully-padded versions did not significantly impact classification performance of the forward network. See Figure 2 for the architecture used.

The advantage of using stride-1 fully-padded convolutional layers is that, as mentioned above, this allows for convolutional layers that learn to invert another convolutional layer. Thus, we do not need to invent and implement a completely new layer to be able to invert convolutional layers.

However, we do need to think about how we attempt to invert convolution. It is possible to simply learn these filters with back-propagation, but it might be beneficial to consider other options in which we can be 'smart'. In particular, we should consider the 'convolutional transpose' (Zeiler and Fergus, 2014) as an option, which is aimed specifically at inverting convolution (though within the context of 'deconvolutional networks' (Zeiler et al., 2010), not autoencoders). The convolutional transpose, apart from permuting the input and output channel dimensions of all filters (simply to make the two mathematically able to be combined), flips all filters in the horizontal and vertical directions. This is equivalent to traversing the filter elements in reverse order in every input-output channel pair. The hope is that this sufficiently approximates the identity function, i.e. that convolution using 'transposed' filters approximately inverts regular convolution. See Section B.8.1 for a more mathematical description.

Note that, for non-square filters, the term convolutional transpose is a slight misnomer, since taking the regular transpose of the filters changes the dimensions of our filters, but this might not be legal. For example, applying a 5×1 filter on 20×1 data is quite possible, but if the transpose gives us a 1×5 filter, it is mathematically not allowed or possible to apply this transposed filter to our data. Nevertheless, for ease of reading we will sometimes refer to it simply as the transpose, or filter transpose.

Now, we can apply this convolutional transpose in different ways in autoencoders. For example, we can initialise decoder filters to the transpose of the corresponding encoder filters before training. We could even *fix* the decoder to use the encoder's transposed filters, so that after every update of an encoding convolutional layer, the corresponding decoding layer is again set equal to the transpose of the updated encoding filters. These methods, and more, are described in more detail in Section 4.3, and I compare these approaches in the experimental part of this thesis; see Section 5.

4.2.2 Pooling layers

Next, we want to invert pooling layers. As here, subsampling is involved, we cannot invert pooling layers with pooling layers, or any other supported type of layers, because we need to be able to increase the size of our data points. This means we need to define a new unpooling layer of some sort, that attempts to invert the pooling layer as well as possible.

For this, we use a technique called switch unpooling (Ranzato et al. (2007), Zeiler and Fergus (2014)), aimed specifically at invert max-pooling layers (the most common type of pooling layers). In max-pooling, we examine all input elements inside the pooling region (which is shifted over all possible base positions as in convolution), and set the output element for that base position to be equal to the maximum of these input elements. In other words, we map the elements of each pooling region to their maximum, and shift the pooling region over the input in a convolutional manner.

Now, to use switch unpooling, we additionally save *which* element was the maximum (breaking ties in some consistent way) when traversing the pooling layer. We call these locations of maximum elements the 'switches'. Then, in switch unpooling, the input data contains the (decoded) value of these maximum elements, so together with these switches, in theory we can perfectly invert max-pooling, *for these maxima*, by setting the unpooled value of these maximum elements to the input of the unpooling layer. However, we have no knowledge of any other elements than the recorded maximum, except that their value is lower than our maximum. Therefore, we simply set all unpooled non-maximum elements to zero, since our data is centred around zero. See Section B.8.2 for a mathematical description.

Whereas switch unpooling works well for autoencoder performance, it may do so in an unintended way. Information containing the switches is 'leaked' from the first pooling layer (the second layer of the encoder, and of the whole network) to the corresponding decoding layer (the second-last layer of the decoder, and thus of the entire network) without any intermediation by other layers. Put in another way, we are not purely decoding the encoded *representation* of the data, but we are making use of information internal to the encoder not normally available to 'the outside'.

To illustrate that this is indeed the case, and that this is unintended, consider the following experiment, due to Churchman (2015). Convert a non-

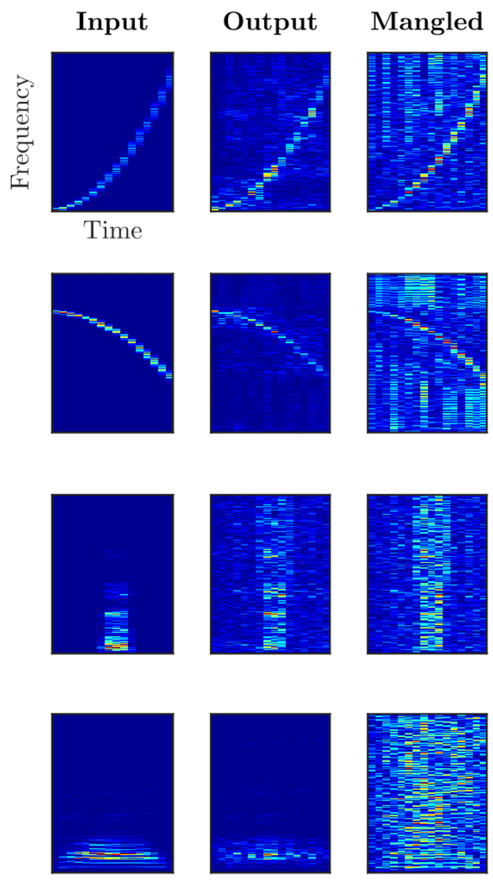


Figure 3: Results for the 'chirp' experiment, for an example autoencoder. The 'mangled' output refers to the condition where the encoded representation is zeroed out. Used with permission from Churchman (2015).

speech sound (in his case, a 'chirp') to the learnt representation, using the encoder. During this process, the switches are saved, and sent to the decoder. Now, before starting the decoding process, zero out the output of the encoder, thereby leaving only the switch information available to the decoder. It would be expected, and perhaps desired, that this would dramatically decrease the quality of the reconstruction, as the output of the encoder 'should' be the main body of information for the decoder. However, instead we find a remarkably good reconstruction of the input 'chirp', even though the only information we have about our input is contained in these switches. This generalises, to some extent, to speech sounds from our actual data set, as well, though for those especially there were problems in the frequency bands where no input signal was found. See Figure 3.

The amount of information kept in these switches may be disadvantageous for the representation learnt in the encoder, as the autoencoder could overly rely on this information for performance, instead of maximising the information content of its output. This is especially bad for our main goal, classification, as the classifier only has access to the encoder's output, and not to the switches (see Section 4.4 for how exactly a trained autoencoder can be used for classification).

As such, we may want to reconsider our use of switch unpooling in our autoencoders. However, if we want to remove the switch information from the equation, that leaves only the value of the maximum element per pooling region available, and the resulting unpooling layer has no way of knowing where this maximum element resided in its pooling region. Furthermore, we cannot make any approximations such as simply mapping everything to zero (or some other fixed value). Especially in the case of zero, this would be disastrous for further decoding due to the multiplicative nature of convolution (after all, $0 \cdot c = 0, \forall c \in \mathbb{R}$), which would leave only biases to work with. For any other fixed constant, we cannot even know whether it is in the desired range of values or not (e.g., a fixed output value of 1 is a horrible choice if the network happens to output values between -0.1 and 0.1).

These considerations give rise to the blind upsampling method, where we simply give all elements in a pooling region the recorded value of their maximum (as this is the only value other than zero we know is of the correct order of magnitude). As some elements may be part of multiple pooling regions, we have to slightly adjust this definition to make this well-defined: the unpooled value of an element Y_{ijd}, is the average of the input values corresponding to all pooling regions which Y_{ijd} is a part of. In practice, this is a fairly terrible inversion, but we simply do not have any more knowledge available. Blind upsampling is mathematically described in Section B.8.3.

I have implemented switch unpooling in C++/CUDA inside the existing MatConvNet toolbox, so that max-pooling layers save their switches, which can be used by the all-new switch unpooling layers in our networks. Additionally, I have implemented blind upsampling in a similarly new unpooling layer. Thus, we can rely on switch unpooling and blind upsampling to invert max-pooling layers in our autoencoder experiments in Section 5, and their performance will also be compared there.

4.3 Training methods

Now that we can approximately invert all necessary layers, it is possible to train autoencoders. However, how exactly is it possible to learn meaningful representations with these models, without knowing anything about the data we are seeing (specifically, without knowing the class each data point belongs to, or having access to an already trained classification model)? The answer is very simple: train everything with regard to the reconstruction error. More specifically, we can randomly initialise both our encoder as our decoder filters and simply use back-propagation to learn both of these from scratch, minimising the Euclidean error between the reconstruction and input.

However, apart from random initialisation, we also have the convolutional transpose as an option to invert convolutional layers. One straightforward way of using it, is by initialising the decoding filters to the transposed filters of the corresponding encoding layer, and then training the network as usual. Since the transposed layer is expected to be a good approximation of the inverse of the encoding layer, this should be a fairly decent initialisation of our decoding filters, given the (in this case) randomly-chosen encoding filters.

There is also a more extreme version of using the convolutional transpose, which relies on this technique even more. In essence, instead of merely initialising the decoder using the filter transpose and allowing them to diverge during training, we fix the decoding layer's filters to always be the transpose of the corresponding encoding layer's filters. Thus, if the encoder is updated during training, the decoder is correspondingly updated, such that the decoder again uses the current filter transpose of the encoder's filters.

This requires not only the ability to merely turn off training for convolutional layers, but it also requires a layer that is constantly updated to be the transpose of the corresponding encoding layer. I have modified MatConvNet's Matlab code accordingly, adding the ability to turn off learning of filters and biases (separately) in specific layers, if needed, and adding a convolutional transpose pseudo-layer, which uses the existing code of convolutional layers, but which is constantly updated to be the encoding layer's convolutional transpose as described above.

Given that we now have the ability to turn off training for specific layers, we can experiment with this as well. For example, by randomly initialising the encoder's weights, and turning off learning, it is possible to evaluate the performance of a decoder given a randomly-chosen encoder. It is, of course, also possible to train the encoder that is best inverted by a randomly-chosen decoder, but this is not of any particular use, given that we are trying to learn meaningful representations.

All of the above training methods are relatively simple: we lay down a network architecture, possibly fix certain filters to certain values (e.g., by turning off learning or fixing a decoder filter to the transpose of an earlier encoder filter), and start training. In contrast, the stacked autoencoder is an autoencoder that is trained step by step, successively higher into the autoencoder hierarchy. For better understanding stacked autoencoders, we introduce the term 'autoencoder unit' (or just 'unit'). An autoencoder unit consists of an encoder and decoder part, and represents one 'level' in the autoencoder hierarchy. Its encoder is comprised of one convolutional layer, one max-pooling layer, and finally a rectified linear unit, a non-linear activation function that maps negative numbers to zero, and leaves non-negative numbers unchanged. Conversely, its decoder starts with a rectified linear unit, followed by a switch-unpooling layer and a convolutional layer. See Figure 4.

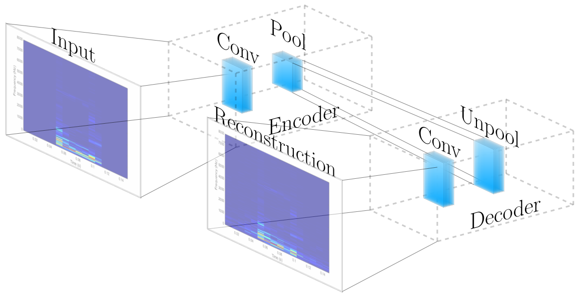


Figure 4: One autoencoder unit. Its encoder consists of a fully-padded stride-1 convolutional and a pooling layer, implicitly followed by a ReLU layer. Its decoder consists of an implicit ReLU layer, an unpooling layer and a fully-padded stride-1 convolutional layer, all three aimed at inverting the corresponding encoder layer. Made using material from Churchman (2015) with permission.

Now, using our forward network's architecture as a model for an autoencoder, we get an autoencoder with four units if we train up to the point where the data size is down to 1×1. The 'regular' training method would be to train the entire network right from the start, which means we train these four units simultaneously. However, the stacked autoencoder works differently. First, train only the first (i.e., outermost) unit until convergence. Then, between the trained encoder and decoder of the first unit, insert the (randomly initialised) second unit, and train until convergence again. Repeat until all layers have been trained.

There are two alternative versions of this training method, which differ in how the units are trained when there are multiple units. In the original stacked autoencoder, previously-trained units are fixed, so units cannot be altered after they have been trained until convergence. This corresponds to the first unit learning some fixed representation $\mathbf{R}_1(\mathbf{X})$ of the data $\mathbf{X}$, and the next unit learning a higher-level representation $\mathbf{R}_2(\mathbf{R}_1(\mathbf{X}))$ of $\mathbf{R}_1(\mathbf{X})$. However, it is possible that the representation $\mathbf{R}_1$ is useful for a single-unit autoencoder, but not optimal for a two-unit system. Thus, a variant of the stacked autoencoder could *initialise* the outermost unit to the learnt representation $\mathbf{R}_1$, instead of *fixing* it. This way, both units can be trained together until convergence. The newly-added unit can still make use of the previously-learnt representation $\mathbf{R}$, but if there is a first-unit representation $\mathbf{R'}_1$ that is more useful for two-unit autoencoders (i.e., $\mathbf{R}_2(\mathbf{R'}_1(\mathbf{X}))$ is more informative), the system can still favour $\mathbf{R'}_1$ over $\mathbf{R}_1$, instead of being forced to use $\mathbf{R}_1$.

4.4 Classification

Now that we are able to train convolutional autoencoders, it is time to use them for our final objective: classification. With the convolutional autoencoder, we should be able to learn meaningful representations of our data set, but how do we utilise these to decide to which class a certain spectrogram belongs?

One straightforward way is to take only the encoder part of a trained autoencoder, and use it to initialise the filters of part of a forward network before training it. For example, for our forward architecture, four-unit autoencoders can be trained, and its filters used to initialise the first four convolutional layers of the forward network. After initialisation, all layers are trained. This technique is called pretraining, and is a common application of autoencoders (e.g., see Masci et al. (2011), Tan and Li (2014)).

However, there is also a method that makes even more use of the learnt representation. Similarly to the stacked autoencoder, instead of merely initialising the first units to those of the trained encoder, we can fix them to that value, not training them any further. This enforces that the learnt representation is used, while the fully connected layers (which are not part of the autoencoder) are still able to learn to properly classify the data.

5 Experiments

Until now, we have gathered many ideas about inverting the convolutional layer, the max-pooling layer, and about how to train an autoencoder. It is now time to put these ideas to the test, in several series of experiments.

5.1 Inversion and small autoencoders

The first series of experiments is aimed at understanding how convolutional layers can best be inverted. Specifically, the value of the convolutional transpose as an inverter of convolutional layers is investigated, as well as small-scale autoencoder experiments intended both to serve as a baseline for the inversion experiments and to examine how different styles of autoencoders compare to each other, and to the inversion experiments. Switch unpooling was used in these experiments.

We test the decoding power of three different techniques for inverting the convolutional layer. Firstly, we take for the decoding convolutional layer a non-learning filter, which is fixed to be the convolutional transpose of the corresponding encoding layer at all times (1). Secondly, instead of fixing the decoder's filters to the transpose, we could instead only initialise these filters to the encoder filters' transpose, and subsequently train the decoder (2). This, crucially, allows the decoder to diverge from the transpose if this would be beneficial to the autoencoder's performance. A third option, used as a baseline, is to not use the transpose at all, and to simply train randomly-initialised filters (3).

The convolutional transpose requires filters to take the filter transpose of, so we used the trained filters of our forward network for our encoder filters, as we already know these learn some kind of meaningful representation. However, it is not certain if this representation is anywhere near optimal. Therefore, apart from training a decoder for this fixed encoder representation (a), one condition also allows the encoder to be trained (b).

As with the type of decoder, this 'variable' in our experiment also deserves a baseline. This immediately brings us into the realm of autoencoders, as we will no longer use the filters trained via supervised learning, but instead we will use a randomly-initialised encoder.

I tested a number of autoencoder techniques. Firstly, as a baseline for autoencoders, both the encoder and decoder are randomly initialised and not trained. It is important to have a completely random baseline, as random representations already lead to decent performance, when followed by fully connected layers. Secondly, another baseline was used, where the encoder was randomly initialised and not trained, but the decoder is instead fixed to be the convolutional transpose. A third baseline was used, where the encoder is fixed to the trained encoder from our forward network, but the decoder is randomly-initialised and not trained.

Then, akin to the combination (1b) above, we evaluate how well the convolutional transpose can be used by an encoder, by fixing the decoder to the transpose of the encoder, and then training a randomly-initialised encoder. Conversely, and similarly to combination (3a) above, we evaluate well a random representation can be decoded by randomly initialising and fixing an encoder, while training a randomly-initialised decoder freely. Lastly, we try to obtain an estimate for optimal performance, by randomly initialising both the encoder and decoder, and training both freely (similar to combination (3b) above).

The above discussion only describes how the encoder and decoder filters are initialised and, possibly, related. By doing so, it left out the concrete network architecture, because this is another experimental parameter, used to find out how certain techniques scale, and interact with pooling and ReLU layers in a network. Since we have a large array of experimental conditions to test, we keep these networks as small as possible, while still relevant, so it is still feasible to train all experimental networks until convergence or a fixed boundary (50 epochs), whichever comes earlier. Note that exact filter sizes and pooling regions were taken from our forward network.

The first architecture we used is concerned only with convolution, as it consists of one encoding convolutional layer and a second, decoding convolutional layer (I). For the second architecture, we look at a one-unit network, where the encoder consists of a convolutional layer followed by a pooling layer and a ReLU, and the decoder consists of a ReLU, a switch unpooling layer, and a decoding convolutional layer (II). The third architecture (III) is a two-unit system, consisting of two of the encoders that were used in (II), in a row, followed by the decoder, which contains (II)'s decoder twice.

5.2 Full-scale autoencoders

The second series of experiments focusses mainly on two issues raised in Section 4. Firstly, we compare blind upsampling and switch unpooling in autoencoders. The outcome is very likely to be in favour of switch unpooling, since more relevant information can be used. However, the definitive judgement will have to wait until after the next series of experiments, described in Section 5.3, where representations learnt using these unpooling techniques are compared. Secondly, the stacked autoencoder is compared against the regular training method (i.e., initialising and training all layers together). Two versions of stacked autoencoders were described in Section 4.3, a 'strong' version which fixed the previously learnt representations and a 'weak' version where the representations are used in initialisation, but then further trained; they are briefly compared. A third question is also addressed (for the regular training method), namely whether the decoder, attempting to recreate the forcedly positive (because of the encoder's ReLU layers) inputs to the corresponding encoding layer, should also contain a ReLU of its own, or if alternatively a decoding unit should consist only of an unpooling layer followed by a convolutional layer.

As baseline, networks were randomly initialised and not trained, in all combinations of using switch unpooling or blind upsampling, and keeping or discarding the ReLU layers in the decoder.

As network architectures, we start with a regular one-unit system and build our way up, adding units in the middle as with the aforementioned two-unit system. We run these tests for systems of one, two, three and a maximum of four units, as after the fourth unit the fully-connected part of the network starts, and it reasons only with 1×1 data (which is even harder to decode).

5.3 Classification

The third and last series of experiments is intended to provide insight into autoencoders, as used for classification. Specifically, here we test the *classification* performance of several networks and training methods, instead of looking at the Euclidean error between reconstruction and input. The classification performance of a network that uses autoencoder should also logically be linked to the quality of the representation learnt by the autoencoder, as high-quality, meaningful representations of the data will be 'easier' to reason about for the networks' fully-connected layers than a low-quality representation with less informative value.

These experiments answer some of the same questions as in the previous section, but now in the context of a classifier. Specifically, we compare the classification performance achieved when using a regular autoencoder, versus a stacked autoencoder, and furthermore we compare the classification rate when blind upsampling versus switch unpooling was used while training the autoencoder. Additionally, we compare the two major methods of using an autoencoder for classification, namely by using the autoencoder's encoder as a fixed representation, and by pretraining, where the forward network's filters are merely initialised to the autoencoder's encoder, but are allowed to be trained further (see also Section 4.4).

As baseline, a randomly-initialised encoder is used in both conditions. Note that using pretraining with a randomly-initialised encoder is equivalent to using no autoencoder at all (i.e., simply training the classification network with randomly-initialised weights).

6 Results

As the tables containing experimental results are too large to included here, they can be found instead in Appendix A.

6.1 Inversion and small autoencoders

The first thing we notice is the enormous error when we fix the encoder to the one from our trained network, and simply use the fixed convolutional transpose as decoding filters. Given that, on average, a spectrogram lies 3.1 Euclidean distance from the null spectrogram (which is also the average spectrogram, as we use zero-centred data), and even a completely randomly-initialised fixed network achieves a Euclidean error of 8.81 on the small convolution-followed-by-transpose network, the average error of 107 for this small network, only by using the fixed transpose, is gigantic.

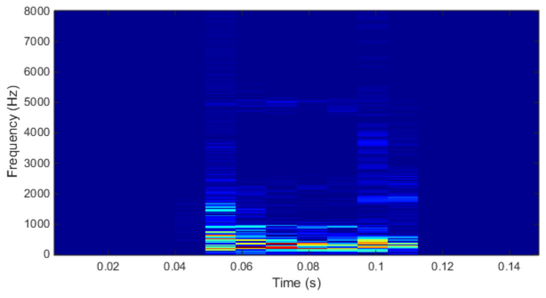
Thus, clearly, something is wrong with this particular set-up. Given that even this simple set-up experienced these large errors, either the encoding filters taken from our forward network employ a 'bad' representation, or the transpose is not actually a good inverse for convolution. Given that either allowing the encoder to be trained, or allowing the decoder to deviate from the transpose during training already returns performance to a more normal order of magnitude, the case could be made that both the encoder is 'bad', *and* the transpose is a poor approximation for the inverse of convolution.

However, other experiments point out that, while the forward network's representation is indeed suboptimal, the transpose not approximating the inverse of convolutional layers well is the main cause of the abysmal performance of the simple convolution-followed-by-transpose set-up. There is a slight indication of this when we initialise our experiment to this network, and then train either the encoding or decoding filters. If we train the encoder, but leave the decoder fixed to the encoder's transpose, the result is a Euclidean error of 3.75 on the small network. If we instead train the decoder, however, and leave the encoding filters fixed to the forward network's trained ones, we obtain a Euclidean error of 2.88. More conclusive evidence is obtained when not merely initialising filters to the trained forward filters and their transpose, but when they are randomly initialised and then trained. Randomly initialising the encoder while keeping the decoder fixed to the transpose leads to a Euclidean error of 2.90 after training, whereas training a randomly-initialised decoder for our fixed forward encoder gives us a network that performs significantly better, with a Euclidean error of 0.39.

More conclusive evidence that the transpose is not a good approximate inverse comes from our baselines. If we randomly initialise an encoder without training it, while using the (fixed) transpose for the decoder, we get a Euclidean error of 28.7. If we replace the transpose by a completely random decoder, we get the (significantly better) Euclidean error of 8.81.

All this evidence suggests the convolutional transpose is somehow rather worthless for reconstruction. However, a closer look at what the network *actually* outputs reveals the truth: the network's reconstructions actually *do* look remarkably similar to their input for lower layers (see Figure 5). The reason the Euclidean error skyrocketed was not due to the convolutional transpose not being useful, but it was caused by the fact that the network's output varied from the input in *intensity*. To illustrate this, consider the arrays [1, -4, 3, 7] and [2, -8, 6, 14]. It is easy to see that the second array is twice the first, and thus we would find this a very good reconstruction: it captured the variations in the input in perfect detail. However, taking the Euclidean error tells us something different: the Euclidean error here is $\sqrt{75} \approx 8.7$, which is not particularly good. To summarise, the Euclidean error concerns itself with the specific numbers, whereas the transpose (and, plausibly, us human judges of our networks' performance) cares more about the 'shape' of the output. The transpose, in this case, serves as an example to suggest that the Euclidean error might not be the best choice of error functions for autoencoders.

If we look further into the results table, we also find information about the representation as trained by the forward network. Specifically, we see that it is clearly not optimal for reconstruction.



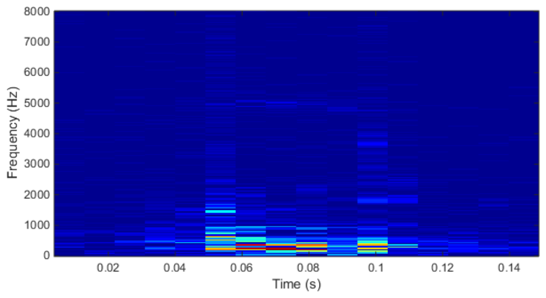


Figure 5: An example reconstruction of a one-unit network using the transpose as decoder

Without training and using a randomly-initialised decoder, our trained encoder leads to a Euclidean error of 17.8, whereas a correspondingly random encoder gives an error of 8.81. For networks with a decoder fixed or initialised to the encoder's transpose, allowing the encoder to train its representation, instead of fixing it to that of the forward network, significantly increases performance, indicating the learnt representation is not optimal.

However, more definitive results can be obtained by looking at when the encoder is randomly initialised, and then trained. We see that, in these cases, the resulting networks ("real" autoencoders, as they do not make use of the network trained in a supervised manner) perform significantly better than their equivalents when the encoder is initialised or fixed to the forward network's.

This is a more general trend in the table. Not only does randomly initialising the encoder improve performance compared to when it is initialised to the trained, classification-purpose representation, randomly initialising the decoder's filters also significantly improves performance compared to when they are initialised with the transpose. It is to be expected that allowing the network to learn makes it perform better than keeping parts of it fixed, but these results seem to suggest that random initialisation is superior (in terms of Euclidean-error performance) over more informed initialisation.

Lastly, there is one general but conflicting trend related to the network architecture used. It is only expected that 2-unit networks perform less well than smaller networks, as they have more parameters to learn and they throw away more data. However, in about half the examined networks, the full-unit architecture (i.e., encoding convolution, max-pooling, ReLU, switch unpooling, decoding convolution) performs noticeably better than the smaller architecture, which only has an encoding convolutional layer followed by a decoding one. Further tests (not shown) have indicated that a larger pooling 'stride', or subsampling rate, enhances this effect.

Looking closer, we see that in fact, all networks for which the pooling layer seems to increase performance (or at least not decrease it) either prominently feature the convolutional transpose, or use a completely random, untrained encoder or decoder. Judging by our earlier observations, this means the 'bad' networks seem to benefit from pooling. This has a rather simple reason: pooling followed by switch unpooling pulls all values but the pooling regions' maxima towards zero, which is fixed to be the average for all input elements (since they are zero-centred). Thus, given a network that performs badly, pulling part of the output towards the average could increase performance over relying on 'wild guesses' or (in the case of the convolutional transpose) a technique suffering from an incompatibility between its main benefit (giving an output that 'looks' like the original input) and the measurement device (the Euclidean error, which is concerned with absolute numerical differences). We also see that the better networks among the examined ones, including mainly the autoencoders and the experiments with more 'randomness' (either in learning or in initialisation) in them, do not benefit from the pooling layer, and indeed they become worse, presumably because their fairly-correct reconstruction is being pulled back towards the average (which lies farther away from their input than the reconstruction they generated).

6.2 Full-scale autoencoders

Here, the most noticeable effect is that autoencoders using blind upsampling are dominated virtually everywhere by their equivalents using switch upsampling, including when a random representation is used and not trained. This is the expected outcome, as switch upsampling has access to relevant information that blind upsampling cannot make use of. Switch upsampling is a perfect inverse for all maximum elements, and outputs the average for all non-maximum elements, whereas blind upsampling also only inverts the maximum elements correctly, but greatly overestimates all non-maximum elements.

This can also clearly be seen if we examine the actual reconstruction. If we look at the reconstructions given by networks using blind upsampling, we notice that the output is always the same, and does not actually depend on the input: these networks have simply learnt to output the average spectrogram, regardless of input. Networks with switch upsampling perform better, and commonly produce recognisable reconstructions (though definitely not always).

Whether the poor performance of blind-upsampling networks is caused by a bad encoder (which uses a null representation), or by a decoder that has not properly learnt to make use of the information in the encoder's representation, will be partially answered in Section 6.3: if a classification network fails to find information with predictive value in the encoder's output, then the encoder clearly has not learnt a proper representation of the data. If, however, a classification network can make use of the encoder, even though the autoencoder's decoder could not, then clearly the autoencoder's decoder must be at fault, possibly due to blind upsampling being inadequate for decoding.

In the results, we can also see that strongly-stacked autoencoders seem to be outperformed by regular autoencoders, while weakly-stacked autoencoders achieve a lower (or at least similar) Euclidean error than both for smaller network sizes, when switch unpooling is used. Four-unit networks proved significantly harder than three-unit variant for all but the non-stacked, switch-unpooling networks, slightly weakening this conclusion.

Again, we can examine the actual reconstructions made by the networks. Here, we see a slightly different sight: strongly-stacked autoencoders, whose reconstructions are mostly decent, actually perform better than weakly-stacked autoencoders, whose reconstructions have, as it seems, activations randomly spread over the spectrogram with no clear relation to the input. However, regular autoencoders beat both types of stacked autoencoders, with reconstructions that can be very good.

No significant difference was found between keeping and discarding decoder ReLU layers, except for the baselines where no actual training took place. Here, discarding the ReLU layers negative impacted performance, which can be easily explained by the fact that ReLUs pulled the (random) feature maps more towards zero, by zeroing out all negative values. Since zero is the average, keeping the ReLU layers therefore improved performance, but not in a way that provides insights into the inner workings of normally-functioning autoencoders.

6.3 Classification

Unfortunately, we see that the baseline of random initialisation has not been surpassed. However, one of the tested networks with pretraining did end up with only 5% higher train error, and only 2% higher test error, which is quite close.

Expectedly, we that networks with pretraining mostly perform better than those with only a fixed encoder, as pretraining allows the network to find an even better encoder representation than the one trained from the autoencoder. There are two exceptions to this, but they are both networks that evidently do not have a good representation of the data, as they are the two worst-performing networks. Therefore, they do not form particularly strong evidence against this conclusion.

However, it is interesting that the difference between the fixed-encoder and pretraining conditions can vary between zero (or slightly negative) and nearly a massive 30% of classification performance. This indicates that, when a given representation may be fairly bad for reconstruction, it can still be a good initialisation point.

Apart from this expected conclusion, no clear and consistent patterns can be found. Blind upsampling is outperformed by switch unpooling on a regular autoencoder, but beats switch unpooling when using a weakly-stacked autoencoder. The strongly-stacked autoencoder achieves a lower error than the weakly-stacked variant for switch unpooling, but there is not enough data (in particular, no strongly-stacked autoencoder that uses blind upsampling) to conclude that this pattern is robust.

7 Conclusion

The main objective of this thesis was to assess the feasibility of convolutional autoencoders, and (if found feasible) use them to improve classification performance. Convolutional autoencoders were indeed found to be feasible, and experiments have been performed to better understand how it performs. However, using them for forward networks has not shown to be an improvement over random initialisation.

Nevertheless, this thesis is not one without any useful results. Importantly, it was found that the Euclidean error is, despite being standard, a bad measure of autoencoder performance. It places too much emphasis on getting the numbers exactly right: it may assign a very large error to reconstructions humans would judge as being near-perfect (e.g., that of the transpose, which led to outputs that looked strikingly similar to the input, but returned output elements in a different number range than the input), and it may assign a (relatively) small error to an autoencoder that always outputs the average spectrogram, regardless of its input.

It has also proven to be a bad predictor of 'usefulness' of the learnt representation. Apart from the above point, where the Euclidean error does not predict how much the reconstruction (subjectively) 'looks like' the input, it is also not very informative for the classification performance when the encoder is used in a forward network. Networks with similar average Euclidean errors led to radically different classification performances when used, even when excluding the random initialisation of a regular forward network (which has a similar Euclidean error to some 'real' autoencoders) from the evidence for this conclusion.

Additionally, how much an autoencoder's reconstructions 'look like' the input does not seem to be a robust indicator of subsequent classification performance either. For example, one network (blind upsampling, weakly-stacked) output the average spectrogram regardless of the input, but still proved of some use for a classifier. Between two networks (switch unpooling, weakly-stacked and blind upsampling, regularly-trained) whose reconstructions both looked fairly random and dissimilar to the input, one had a representation so bad the forward network hardly exceeded chance level even when the encoder was only used for initialisation (and thus, subject to training), whereas the other learnt slowly and was still terrible compared to other networks, but still significantly exceeded chance level.

We can conclude from this last point, as well as the absence of performance improvement when using autoencoders for classification, that representations that are good (or optimal) for reconstruction are not necessarily good (or optimal) for classification. The converse can be concluded from the inversion experiments, where random initialisation consistently beat the trained encoder. Thus, we conclude that good autoencoder representations and good classifying representations are distinct, and one cannot be used to improve another. Instead, random initialisation beats the more informed initialisations considered in this work in all of the experiments described here. However, if the results are genuine (and not caused by possible issues in the experiment itself, see Section 8), this is not too worrying for (convolutional) autoencoders and pretraining in general, as Saxe et al. (2011) already notes that random weights can work remarkably well for good architectures, and as such the good performance of random weights was already known in literature.

More conclusions can be drawn within the separate 'stages' of this thesis. The convolutional transpose looks like a good way to invert convolutional layers, but when measured by Euclidean error, it does not perform well because its reconstruction had scaling issues (its elements were not in the same range of numbers as those of the input). Nevertheless, its reconstructions do look noticeably similar to the original input. Switch unpooling works well in autoencoders, but it does so by 'leaking' information through channels other than the actual representation. The leaked information is so predictive that chirps could be recognisably reconstructed from deep inside the network even when the encoded representation was zeroed out. Blind upsampling does not have this problem, but instead leads to terrible autoencoder performance and not uncommonly to the network not learning a proper representation of the data (instead opting to always output the average).

An example of the unpredictiveness of the Euclidean error is that stacked autoencoders are outperformed by regularly-trained networks when measured by Euclidean error, but outperform regularly-trained networks when used in pretraining for classifying networks. This example also shows that a representation can be a good initialisation point without being particularly predictive itself, because we also see that these stacked autoencoders do not do nearly as well when the autoencoder's representation is fixed. This is slightly expected: random filters make for good initialisation points, as consistently proven in all experiments in this work, but a randomly-initialised encoder is still far from optimal, as training the encoder can greatly improve performance.

In summary, the main conclusions of this work are as follows. Good autoencoder representations do not make good classifier representations and vice versa, but random initialisation is beneficial for learning both. The Euclidean error is a bad predictor of autoencoder performance and subsequent classification performance, and whether an autoencoder's reconstructions 'look like' the original input is not indicative of classification performance either. The convolutional transpose works well as a decoder of convolution, but has scaling issues. Switch unpooling works optimally, but leaks vital information an autoencoder can and will rely on; blind upsampling does not leak information, but performs badly. Stacked autoencoders are superior to regular networks when used in pretraining, but random initialisation is still better than using any of the autoencoders tested.

8 Discussion

Looking back at the work reported in this thesis, the conclusion that using convolutional autoencoders for classification does not improve performance over a random initialisation, is slightly disappointing, as this was one of the main objectives of this thesis.

As already mentioned, there are two main explanations for this result. The first is that randomly-initialised weights, by themselves, already make for a good and predictive representation. This has been acknowledged in the literature (Saxe et al., 2011), and would imply that our autoencoders' representations are not necessarily terrible, they might just not be good enough than the apparently-informative random ones. However, there are alternative explanations for why the random-initialisation baseline was not beaten in this project (despite getting quite close with pretraining), and therefore we can formulate new directions of future research.

Firstly, as mentioned in Section 7, the Euclidean error proved to be rather unhelpful in predicting both human-judged autoencoder performance and the usefulness of the learnt representation for classification. However, it is not only used in the selection of architectures and techniques to experiment with (especially those chosen for classification experiments), it is also used in the autoencoders' learning rule during training, thereby influencing the representations learnt. Thus, the effect of using different error functions, or autoencoder techniques which change the objective function, should be investigated.

Specifically, part of the experiments could be repeated using sparse autoencoders Ng (2011), forcing a more compact representation (and therefore, hopefully, leading to the development of more informative features) that could be easier to reason with for subsequent classification. Contractive autoencoders (Rifai et al., 2011), which are trained to use a more stable and invariant representation, might also perform better when used for classification. These techniques may also help compensate for the lack of dropout or a similar mechanism in the autoencoders, which might have caused similar features to be learnt across different units.

Another method that could be tried out is the denoising autoencoder (Vincent et al., 2008), which artificially adds noise to the training data, so the learnt representation becomes invariant to it. Similarly, data augmentation (e.g., adding noise to the input, or slightly shifting it) during training might help the network build more robust representations.

Ultimately, it is impossible to try out all architectures. Just as our results indicate good classifier representations may not make good autoencoder representations, it is possible architectures that work well for classification do not always work well for autoencoders. Then, it is also possible that this held for the architecture of our forward network, which was, after all, only selected based on classification performance. Given that only this forward architecture was used as a reference for all autoencoder architectures tested here, it is possible the number of filters per layer, the filter sizes, or the number of layers, was adequate for our classifier, but not for autoencoders (since they do require more information than classifiers).

It is also possible that our preprocessing caused problems. Specifically, if a working set of parameters for mel-frequency cepstra was found, with a corresponding network that performed better than the STFT version, the preprocessed data could have been 'easier' to reason with for our autoencoders and classifiers based thereon.

Some research also suggests that using rectified linear units as activation function is suboptimal. Thus, the reconstruction error and possibly the subsequent classification rate could improve when they are replaced by hyperbolic tangent function (as in Kayser and Zhong (2015), also on speech data) or a sigmoid function (as in Ranzato et al. (2007)). However, sources disagree, as Kayser and Zhong (2015) reports the hyperbolic tangent improving performance over rectified linear units, whereas Zeiler et al. (2013) and Sainath et al. (2013a) report that rectified linear units work better than logistic units, such as the hyperbolic tangent and the sigmoid.

In addition, it is possible that the networks considered here actually *do* perform better than regular convolutional neural networks, but that this was not seen in the 50-epoch experiments considered here. This would mean that initialisation with autoencoder filters (or using these as a fixed representation) leads to a lowered convergence point of the test error, for example by combating overfitting or preventing the network to tread into a local minimum it cannot get out of. This can easily be seen by letting the considered networks run until test-error convergence, but was not possible within this project due to time constraints.

Also, it is possible using ensemble strategies could have improved performance. For example, Churchman (2015) reports a decrease in reconstruction error when averaging over several stochastic models, or when only one deterministic model is used, when averaging over reconstructions of the noise-augmented spectrogram (cf. denoising autoencoder). It is not unthinkable ensemble strategies can be used to benefit classification performance, as well (e.g., classification by majority vote).

As a result of the convolutional transpose causing a large Euclidean error when used for inverting convolution, representations learnt using this technique have not been examined for full-scale autoencoders, and have not been used for classification. It would be interesting to see if, despite having a reconstruction in the wrong range (and order of magnitude) of numbers, representations learnt with this technique are of sufficient quality to surpass random initialisation.

In virtually all experiments described in this work, including both autoencoder and classification experiments, the recorded test errors were consistently lower than the training errors. This cannot be explained by the fact that, in the training phase of an epoch, the network is still improving, causing the average error of that epoch to include errors made by worse, slightly earlier versions of the network: not only is the magnitude of this difference larger than the amount of improvement the network makes in one epoch, this effect even persists for experiments where no learning takes place at all. Given that the TIMIT data uses a fixed partition to divide its data into a training and test set, a plausible explanation is the two sets do not have the exact same phone distribution, and that the test set is somehow 'easier' than the training set. However, a more interesting effect could be at play, so this is open to investigation.

References

Abdel-Hamid, O., Deng, L., and Yu, D. (2013). Exploring convolutional neural network structures and optimization techniques for speech recognition. In *INTERSPEECH*, pages 3366–3370.

Abdel-Hamid, O., Mohamed, A.-R., Jiang, H., Deng, L., Penn, G., and Yu, D. (2014). Convolutional neural networks for speech recognition. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, 22(10):1533–1545.

Abdel-Hamid, O., Mohamed, A.-R., Jiang, H., and Penn, G. (2012). Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 4277–4280. IEEE.

Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., et al. (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153.

Bottou, L. (1991). Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nîmes*, 91(8).

Bourlard, H. (2000). Auto-association by multilayer perceptrons and singular value decomposition. Technical report, IDIAP.

Churchman, T. J. (2015). Reconstructing speech input from convolutional neural network activity. Unpublished bachelor's thesis, Radboud University Nijmegen, the Netherlands.

Cireşan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Garofolo, J. S. et al. (1993). *TIMIT: acoustic-phonetic continuous speech corpus.* Linguistic Data Consortium.

Graham, A. (1981). *Kronecker Products and Matrix Calculus: With Applications.* Ellis Horwood series in mathematics and its applications. Ellis Horwood Limited.

Güçlü, U. and van Gerven, M. A. (2014). Deep neural networks reveal a gradient in the complexity of neural representations across the brain's ventral visual pathway. *arXiv preprint arXiv:1411.6422.*

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-R., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97.

Kayser, M. and Zhong, V. (2015). Denoising convolutional autoencoders for noisy speech recognition. Unpublished.

Kemper, D. (2015). Understanding the features of a convnet trained for phone recognition. Unpublished bachelor's thesis, Radboud University Nijmegen, the Netherlands.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Kruger, N., Janssen, P., Kalkan, S., Lappe, M., Leonardis, A., Piater, J., Rodriguez-Sanchez, A. J., and Wiskott, L. (2013). Deep hierarchies in the primate visual cortex: What can we learn for computer vision? *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1847–1871.

Lawrence, S., Giles, C. L., Tsoi, A. C., and Back, A. D. (1997). Face recognition: A convolutional neural-network approach. *Neural Networks, IEEE Transactions on*, 8(1):98–113.

LeCun, Y. and Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10).

Lee, K.-F. and Hon, H.-W. (1989). Speaker-independent phone recognition using hidden Markov models. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(11):1641–1648.

Leng, B., Guo, S., Zhang, X., and Xiong, Z. (2015). 3d object retrieval with stacked local convolutional autoencoder. *Signal Processing*, 112:119–128.

Masci, J., Meier, U., Cireşan, D., and Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. In *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 52–59. Springer.

Ng, A. Y. (2011). Sparse autoencoder. Unpublished.

Patterson, R. D., Nimmo-Smith, I., Holdsworth, J., and Rice, P. (1987). An efficient auditory filterbank based on the gammatone function. In *a meeting of the IOC Speech Group on Auditory Modelling at RSRE*, volume 2.

Patterson, R. D., Robinson, K., Holdsworth, I., McKeown, D., Zhang, C., and Allerhand, M. (1992). Complex sounds and auditory images. In *Auditory Physiology and Perception: Proceedings of the 9th International Symposium on Hearing Held in Carcens, France on 9-14 June 1991*, number 83, page 429. Pergamon.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Ranzato, M. A., Huang, F. J., Boureau, Y.-L., and LeCun, Y. (2007). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE.

Rifai, S., Vincent, P., Muller, X., Glorot, X., and Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 833–840.

Sainath, T. N., Kingsbury, B., Mohamed, A.-r., Dahl, G. E., Saon, G., Soltau, H., Beran, T., Aravkin, A. Y., and Ramabhadran, B. (2013a). Improvements to deep convolutional neural networks for lvcsr. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 315–320. IEEE.

Sainath, T. N., Mohamed, A.-R., Kingsbury, B., and Ramabhadran, B. (2013b). Deep convolutional neural networks for LVCSR. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8614–8618. IEEE.

Saxe, A., Koh, P. W., Chen, Z., Bhand, M., Suresh, B., and Ng, A. Y. (2011). On random weights and unsupervised feature learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1089–1096.

Tan, S. and Li, B. (2014). Stacked convolutional auto-encoders for steganalysis of digital images. In *Asia-Pacific Signal and Information Processing Association, 2014 Annual Summit and Conference (APSIPA)*, pages 1–4. IEEE.

Tóth, L. (2014). Combining time-and frequency-domain convolution in convolutional neural network-based phone recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 190–194. IEEE.

Vedaldi, A. and Lenc, K. (2014). MatConvNet - convolutional neural networks for MATLAB. *CoRR*, abs/1412.4564. Accessed in January 2015 (v1.0-beta8).

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*, pages 818–833. Springer.

Zeiler, M. D., Krishnan, D., Taylor, G. W., and Fergus, R. (2010). Deconvolutional networks. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2528–2535. IEEE.

Zeiler, M. D., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q. V., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J., et al. (2013). On rectified linear units for speech processing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3517–3521. IEEE.

Zheng, F., Zhang, G., and Song, Z. (2001). Comparison of different implementations of MFCC. *Journal of Computer Science and Technology*, 16(6):582–589.

A Full experimental results

This appendix contains the full array of results obtained for the experiments described in Section 5 (included as an appendix because the tables are too large to be included in the main body). Both the encoder and decoder are trained, unless it is explicitly stated that they are fixed. n stands for the number of epochs the network was trained. In cases where this number is marked with the dagger symbol†, the network was manually stopped, as it either fundamentally had no capability to learn (due to only having fixed filters), or showed that it did not learn for an extended number of epochs (at least 5). For the other networks, the training phase was capped at the fiftieth epoch.

For reference, the average Euclidean distance from a spectrogram to zero (also the average spectrogram, as we have zero-centred our data), is 3.1, and the achieved Euclidean errors for random initialisation without training can be found at the top of each table.

A.1 Inversion and small autoencoders

Table 3: Results of the experiments described in Section 5.1.

Euclidean error	conv. + inverse	1 unit	2 units
Baseline: no training, encoder fixed and initialised randomly, decoder fixed and initialised randomly	train 9.45	train 3.46	train 3.61
	test 8.81	test 3.29	test 3.37
	$n = 8^{\dagger}$	$n = 17^{\dagger}$	$n = 17^{\dagger}$
Baseline: no training, encoder fixed and initialised randomly, decoder fixed to transpose	train 30.2	train 4.13	train 3.40
	test 28.7	test 3.85	test 3.20
	$n = 2^{\dagger}$	$n = 11^{\dagger}$	$n = 5^{\dagger}$
Baseline: no training, trained fixed encoder, decoder fixed and initialised randomly	train 18.7	train 4.13	train 13.7
	test 17.8	test 3.85	test 12.8
	$n = 2^{\dagger}$	$n = 5^{\dagger}$	$n = 4^{\dagger}$
Fixed trained encoder, decoder fixed to transpose	train 114	train 28.8	train 426
	test 107	test 27.1	test 400
	$n = 6^{\dagger}$	$n = 7^{\dagger}$	$n = 5^{\dagger}$
Fixed trained encoder, decoder initialised to transpose	train 3.06	train 1.40	train 4.75
	test 2.88	test 1.29	test 4.41
	$n = 44$	$n = 50$	$n = 34$
Fixed trained encoder, decoder initialised randomly	train 0.39	train 1.28	train 3.00
	test 0.36	test 1.20	test 2.82
	$n = 39$	$n = 12$	$n = 47$
Initialised to trained encoder, decoder fixed to transpose	train 4.02	train 3.21	N/A
	test 3.75	test 3.04	N/A
	$n = 49$	$n = 50$	N/A
Initialised to trained encoder, decoder initialised to transpose	train 1.85	train 1.83	train 3.06
	test 1.73	test 1.73	test 2.88
	$n = 50$	$n = 50$	$n = 42$
Initialised to trained encoder, decoder initialised randomly	train 0.29	train 0.82	train 3.06
	test 0.27	test 0.78	test 2.86
	$n = 33$	$n = 50$	$n = 43$
Encoder initialised randomly, decoder fixed to transpose	train 3.16	train 3.14	train 3.18
	test 2.90	test 2.92	test 3.02
	$n = 29^{\dagger}$	$n = 11^{\dagger}$	$n = 8^{\dagger}$
Encoder fixed and initialised randomly, decoder initialised randomly	train 0.26	train 1.47	train 2.32
	test 0.24	test 1.36	test 2.21
	$n = 50$	$n = 26^{\dagger}$	$n = 15^{\dagger}$
Encoder initialised randomly, decoder initialised randomly	train 0.14	train 0.73	train 1.89
	test 0.13	test 0.69	test 1.77
	$n = 35$	$n = 50$	$n = 47$

A.2 Full-scale autoencoders

Table 4: Results of the experiments described in Section 5.2. Note that, for single-unit networks, neither stacked vs. regular training (as there is nothing to stack on), nor the decoder ReLU layers (the encoder ReLU still exists) actually change anything, so they are only listed once, in the uppermost row with an equivalent set-up.

Euclidean error	1 unit	2 units	3 units	4 units
Baseline: no training, random initialisation, switch unpooling, with decoder ReLUs	train 3.46	train 3.37	train 3.35	train 3.38
	test 3.29	test 3.21	test 3.16	test 3.15
	$n = 17^\dagger$	$n = 2^\dagger$	$n = 2^\dagger$	$n = 3^\dagger$
Baseline: no training, random initialisation, switch unpooling, without decoder ReLUs		train 3.45	train 3.43	train 4.44
		test 3.20	test 3.19	test 4.11
		$n = 3^\dagger$	$n = 2^\dagger$	$n = 3^\dagger$
Baseline: no training, random initialisation, blind upsampling, with decoder ReLUs	train 5.26	train 3.46	train 3.37	train 3.40
	test 4.73	test 3.26	test 3.18	test 3.22
	$n = 4^\dagger$	$n = 5^\dagger$	$n = 3^\dagger$	$n = 8^\dagger$
Baseline: no training, random initialisation, blind upsampling, without decoder ReLUs		train 3.82	train 3.40	train 7.06
		test 3.51	test 3.19	test 6.40
		$n = 3^\dagger$	$n = 4^\dagger$	$n = 20^\dagger$
Regular training, switch unpooling, with decoder ReLUs	train 0.73	train 1.89	train 2.49	train 2.67
	test 0.69	test 1.77	test 2.22	test 2.48
	$n = 50$	$n = 47$	$n = 25^\dagger$	$n = 20^\dagger$
Regular training, switch unpooling, without decoder ReLUs		N/A	train 2.39	train 2.61
		N/A	test 2.22	test 2.41
		N/A	$n = 32$	$n = 50$
Regular training, blind upsampling, with decoder ReLUs	train 3.24	N/A	N/A	train 3.34
	test 3.03	N/A	N/A	test 3.16
	$n = 19^\dagger$	N/A	N/A	$n = 12^\dagger$
Regular training, blind upsampling, without decoder ReLUs		N/A	N/A	train 3.38
		N/A	N/A	test 3.14
		N/A	N/A	$n = 15^\dagger$
Stacked (strong), switch unpooling, with decoder ReLUs		train 2.30	train 2.97	train 3.33
		test 2.20	test 2.77	test 3.18
		$n = 19^\dagger$	$n = 9^\dagger$	$n = 7^\dagger$
Stacked (weak), switch unpooling, with decoder ReLUs		train 1.63	train 2.48	train 3.59
		test 1.52	test 2.32	test 3.41
		$n = 17^\dagger$	$n = 48$	$n = 50$
Stacked (weak), blind upsampling, with decoder ReLUs		train 3.20	train 3.35	train 3.35
		test 3.03	test 3.11	test 3.17
		$n = 18^\dagger$	$n = 17^\dagger$	$n = 9^\dagger$

A.3 Classification

Table 5: Results of the experiments described in Section 5.3.

Classification error (%)	fixed encoder	pretraining
Baseline: randomly-initialised encoder[1]	train 69.1	train 50.2
	test 66.3	test 48.2
	$n = 45$	$n = 50$
Regularly-trained autoencoder, switch unpooling	train 79.4	train 79.3
	test 74.4	test 66.6
	$n = 44$	$n = 39$
Regularly-trained autoencoder, blind upsampling	train 97.0	train 97.2
	test 97.9	test 95.8
	$n = 8^{\dagger}$	$n = 43$
Strongly-stacked autoencoder, switch unpooling	train 84.6	train 55.7
	test 73.4	test 50.5
	$n = 47$	$n = 50$
Weakly-stacked autoencoder, switch unpooling	train 87.5	train 88.2
	test 83.3	test 80.0
	$n = 43$	$n = 30$
Weakly-stacked autoencoder, blind upsampling	train 79.7	train 61.1
	test 75.4	test 56.1
	$n = 49$	$n = 50$

[1]Note that a classifying network that uses pretraining with a randomly-initialised encoder is completely equivalent to a regularly-trained forward network.

B Mathematical descriptions and derivations

In this section I will describe in precise mathematical detail what a neural network consists of, how it is represented and how a given network computes its output. Furthermore, I will describe how a neural network can be trained using gradient descent, and derive formulas to calculate the derivatives that are necessary to execute this algorithm for convolutional neural networks. Finally, I will give a mathematical description of convolutional auto-encoders, and extra layers and derivatives necessary for these types of models.

The definitions used herein are all standard definitions from the field, but the relevant derivations described in this appendix are all my own work (except where indicated). The relevance of this appendix is both so we have a solid mathematical account of what happens in convolutional neural networks and autoencoders, and to clear up several ambiguities and (occasionally) mistakes in Vedaldi and Lenc (2014).

B.1 Representation of a neural network

Very generally, a neural network in our context is a function $\mathbf{f}$ that maps an input matrix $\mathbf{X} \in \mathbb{R}^{\#U \times D}$ to some output matrix $\mathbf{Y} \in \mathbb{R}^{\#U' \times K}$, given a collection of weights $\mathbf{w}$. Here, $\#U$ and $\#U'$ stand for the number of units that encode the input and output of $\mathbf{f}$, and D and K are the number of dimensions each unit can encode (this is needed for e.g. complex-number or multidimensional input). This mechanism is displayed in Figure 6.

Instead of referring to input and output as these matrices, we will also refer to them in their vectorised forms $\mathbf{x} = \text{vec}(\mathbf{X}) \in \mathbb{R}^{\#UD}$ and $\mathbf{y} = \text{vec}(\mathbf{Y}) \in \mathbb{R}^{\#U'D}$, which makes it easier to define certain derivatives described later. Note that the vec operator is invertible given fixed dimensions, which allows us to switch freely between these representations. For a precise definition of vectorisation, see Graham (1981).

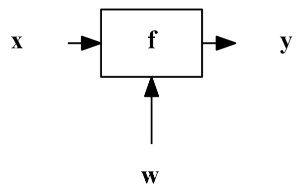


Figure 6: Generic view of a neural network

However, this generic view of a neural network is oversimplified. In reality, a (deep) neural network is not a single arbitrary mapping, but it consists of L layers. Each layer i computes its own function $\mathbf{f}_i$ given respective inputs $\mathbf{x}_i$, weights $\mathbf{w}_i$ (however, note that not all types of layers actually make use of weights) and output $\mathbf{y}_i$. Here, the output of one layer is fed to the next, so we set $\mathbf{x}_{i+1} := \mathbf{y}_i$ (barring border cases). Thus, we have that the total function computed is the composition of the individual functions $\mathbf{f} = \mathbf{f}_1 \circ \mathbf{f}_2 \circ \ldots \circ \mathbf{f}_L$, where the concrete individual functions implemented $\mathbf{f}_i$ depend on their weights $\mathbf{w}_i$. The total weight vector $\mathbf{w}$ from the previous paragraph is an array containing all layers' weights $\mathbf{w}_i$, making the layered view a special case of the previous, more generic view. This layered configuration can be seen in Figure 7.

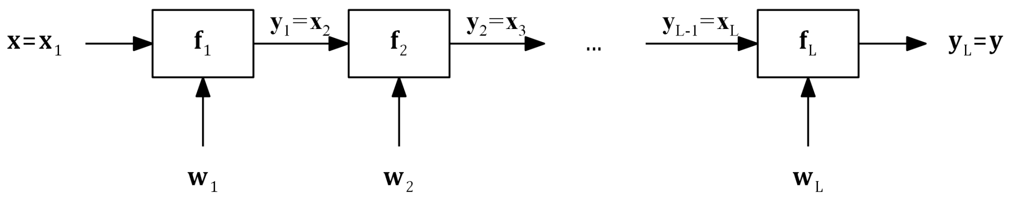


Figure 7: More specific, layered view of a neural network

B.2 Training by gradient descent

Gradient descent is a general optimisation technique that attempts to minimise some error function by slightly changing a certain set of parameters in the direction where the error decreases the most. To use gradient descent, we need to know not only the data set's inputs $\mathbf{x}^n$, so that we can calculate the network's outputs $\mathbf{y}^n := \mathbf{f}(\mathbf{x}^n, \mathbf{w})$, but we also need to know the desired outputs of the network $\mathbf{t}^n$. This necessity makes this a supervised learning technique.

Given the above data, we can calculate the gradient of $E(\mathbf{w})$:

$$\nabla E(\mathbf{w}) = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = (\frac{\partial E(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial E(\mathbf{w})}{\partial w_m}),$$

where m is the number of weight elements in $\mathbf{w}$. This gradient tells us the direction in which an update of $\mathbf{w}$ will result in the strongest increase in the error function, and $-\nabla E(\mathbf{w})$ tells us the direction of strongest error decrease. We can use this in the update rule of gradient descent,

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E(\mathbf{w}), \text{ or}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla E(\mathbf{w}^{(t)}),$$

for a small positive learning rate η and time step $t \geq 0$. Repeating this update rule until (approximate) convergence has been reached, guarantees that with the resulting network weights we are in a local minimum of the error function, i.e. we have locally reached top performance. Note that it is not guaranteed the network ends up in a global minimum.

The above is known as a batch update rule, or as a whole batch gradient descent. The error function $E(\mathbf{w}) = \sum_{n=1}^{N} E^n(\mathbf{w})$ sums over the element-wise errors for all N data elements, and therefore calculating its value and its gradient requires examining the entire data set. This means the network has to process every single data point to execute one update. This is computationally very expensive for large data sets such as TIMIT. Thus, for our purposes it is easier to use stochastic gradient descent. In stochastic gradient descent, we do not compute the gradient of the total error function $E(\mathbf{w})$, but we use only the element-wise error function $E^n(\mathbf{w})$, as defined above. The update rule then becomes

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E^n(\mathbf{w}), \text{ or}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla E^n(\mathbf{w}^{(t)}),$$

where n is randomly circulated through the data elements. This is easier to compute, and usually converges faster (Bottou, 1991).

For practical purposes, a hybrid approach is often preferable, since basing updates on single data points leads to rather jittery updates. In this hybrid approach, the above batch update rule is used, but instead of summing over the entire data set (which is computationally expensive), we sum over a mini-batch that contains a reasonable number of randomly chosen data points (e.g., 64 or 128 points). This way, the jitter is mostly removed from the updates, since it will be averaged out over the elements of the mini-batch, but batches are still small enough to be computationally efficient. The mini-batch approach is also a form of stochastic gradient descent, and we will use that term to refer to the mini-batch approach from here on.

B.3 Back-propagation of errors

As noted above, to use stochastic gradient descent we need to be able to compute the gradient of the element-wise error function, $\nabla E^n(\mathbf{w})$. However, it is obviously not feasible to fully expand ℓ and $\mathbf{f}$, and "simply" compute the derivative of the expanded function, since the resulting function will be rather large and complex. Instead, we will use the (multivariate) chain rule to decompose the gradient into smaller, more handleable parts.

First of all, we use the fact that all interaction between E^n and $\mathbf{w}$ is mediated by the value of $\mathbf{y}^n$, so we can focus on these parts separately (using the chain rule):

$$\frac{\partial E^n}{\partial \mathbf{w}} = \frac{\partial E^n}{\partial \mathbf{y}^n} \frac{\partial \mathbf{y}^n}{\partial \mathbf{w}}.$$

The first part, $\frac{\partial E^n}{\partial \mathbf{y}^n}$ is simply the derivative of the error function given the network output, given the network output. This depends on the particular choice of error function, and thus can be found in Section B.4.

For the second part, $\frac{\partial \mathbf{y}^n}{\partial \mathbf{w}}$, we have the same problem as for the original gradient we wanted to find, namely that $\mathbf{y}^n$ does not depend directly on $\mathbf{w}$, but that this dependency first goes through intermediate layers. Say we want to compute the partial derivative of $\mathbf{y}^n$ with respect to $\mathbf{w}_i$, the weight vector of layer i. The data point's index n is left out of the equations for readability from this point onwards, as it is not particularly relevant in these derivatives.

We can apply the multivariate chain rule to our partial derivative, given that $\mathbf{y} = \mathbf{y}_L = \mathbf{f}_L(\mathbf{x}_L, \mathbf{w}_L)$. This gives us that

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{w}_i} + \frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_L}\frac{\partial \mathbf{w}_L}{\partial \mathbf{w}_i}.$$

Here, $\frac{\partial \mathbf{y}_L}{\partial \mathbf{x}_L}$ and $\frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_L}$ can be obtained by examining the function implemented by layer L (see Section B.6). Furthermore, $\frac{\partial \mathbf{w}_L}{\partial \mathbf{w}_i}$ is easily determined. In the case where $i = L$, $\frac{\partial \mathbf{w}_L}{\partial \mathbf{w}_i}$ will be equal to the identity matrix. Furthermore, $\mathbf{w}_i = \mathbf{w}_L$ does not depend on the input of that layer, $\mathbf{x}_L$, so $\frac{\partial \mathbf{x}_L}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{x}_L}{\partial \mathbf{w}_L} = 0$. Thus, we simply end up with the trivial equation $\frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_L}$ in this case. As mentioned before, the value of this derivative depends on the function layer L implements.

However, it could also be that $i \neq L$. In this case, given that weights are independent from one another, we get that $\frac{\partial \mathbf{w}_L}{\partial \mathbf{w}_i} = 0$ and we end up with

$$\frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{w}_i}.$$

Note that, per construction, $\mathbf{x}_L = \mathbf{y}_{L-1}$, giving us

$$\frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{x}_L}\frac{\partial \mathbf{y}_{L-1}}{\partial \mathbf{w}_i}.$$

Now, we can repeat exactly the same argument as before. If $i = L-1$, then we have

$$\frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_{L-1}} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{x}_L}\frac{\partial \mathbf{y}_{L-1}}{\partial \mathbf{w}_{L-1}},$$

which is determined by the functions implemented by layers $L-1$ and L. If $i \neq L-1$, we again apply the chain rule to give us

$$\frac{\partial \mathbf{y}_L}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{x}_L}\frac{\partial \mathbf{y}_{L-1}}{\partial \mathbf{x}_{L-1}}\frac{\partial \mathbf{x}_{L-1}}{\partial \mathbf{w}_{L-1}}.$$

We can repeat this argument several times, applying the chain rule until we arrive at layer i. The general formula we arrive at is

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}_i} = \frac{\partial \mathbf{y}_L}{\partial \mathbf{x}_L}\frac{\partial \mathbf{y}_{L-1}}{\partial \mathbf{x}_{L-1}}\cdots\frac{\partial \mathbf{y}_{i+1}}{\partial \mathbf{x}_{i+1}}\frac{\partial \mathbf{y}_i}{\partial \mathbf{w}_i}.$$

Combining the previous results, we see that

$$\frac{\partial E^n}{\partial \mathbf{w}_i} = \frac{\partial E^n}{\partial \mathbf{y}^n}\frac{\partial \mathbf{y}^n}{\partial \mathbf{w}_i} = \frac{\partial E^n}{\partial \mathbf{y}^n_L}\frac{\partial \mathbf{y}^n_L}{\partial \mathbf{x}^n_L}\frac{\partial \mathbf{y}^n_{L-1}}{\partial \mathbf{x}^n_{L-1}}\cdots\frac{\partial \mathbf{y}^n_{i+1}}{\partial \mathbf{x}^n_{i+1}}\frac{\partial \mathbf{y}^n_i}{\partial \mathbf{w}_i},$$

where the remaining partial derivatives are determined by layers i and up. As we will later see, it can also be convenient to collapse this back to

$$\frac{\partial E^n}{\partial \mathbf{w}_i} = \frac{\partial E^n}{\partial \mathbf{y}^n_L}\frac{\partial \mathbf{y}^n_L}{\partial \mathbf{x}^n_L}\frac{\partial \mathbf{y}^n_{L-1}}{\partial \mathbf{x}^n_{L-1}}\cdots\frac{\partial \mathbf{y}^n_{i+1}}{\partial \mathbf{x}^n_{i+1}}\frac{\partial \mathbf{y}^n_i}{\partial \mathbf{w}_i} = \frac{\partial E^n}{\partial \mathbf{y}^n_i}\frac{\partial \mathbf{y}^n_i}{\partial \mathbf{w}_i}$$

Gradient descent using these resulting rules is called back-propagation of errors, or simply back-propagation, because the value of a partial error derivative in layer i influences (propagates through) the partial error derivatives of the layers lower in the network (closer to the input, hence 'back').

B.4 The error function

Furthermore, we need to define an error function $E(\mathbf{w})$ that we want to minimise. For classification problems, including our TIMIT phone classification task, the logarithmic loss function is often used for our error function. This function takes the summed negative logarithm of all input elements in the feature channel that belongs to the *correct* class. Formally, if $c^n \in \{1, 2, \ldots, K\}$ is the correct class of data point n out of N, and $\mathbf{Y}^n \in \mathbb{R}^{\#U' \times K}$ the output of the network with weights $\mathbf{w}$ and input $\mathbf{X}^n$, then we could say

$$E(\mathbf{w}) = \sum_{n=1}^{N} E^n(\mathbf{w}) := \sum_{n=1}^{N} l(\mathbf{Y}^n, c^n),$$

where the logarithmic loss function l for a single data point is defined as

$$l(\mathbf{Y}, c) = -\sum_{i=1}^{\#U'} \log Y_{ic}.$$

Note that the dependency of the element-wise error, E^n, on $\mathbf{w}$ lies in the fact that $\mathbf{Y}^n = \mathbf{f}(\mathbf{X}^n, \mathbf{w})$ depends on $\mathbf{w}$.

There is, however, a problem with using the plain logarithmic loss function as error function: given that it computes the logarithm of elements of output matrix $\mathbf{Y}$, it may be undefined (or infinite) if elements of $\mathbf{Y}$ are negative or zero, which they might be, given that weights (and thus, the precise behaviours of function $\mathbf{f}$) are initialised randomly. Obviously, updating our weights by an infinite amount is not a good idea, so we need an alternative error function that can handle both negative and positive numbers. For this, we combine the logarithmic loss function with the softmax function σ. $\sigma(\mathbf{Y})$ is defined by

$$[\sigma(\mathbf{Y})]_{ik} = \frac{e^{Y_{ik}}}{\sum_{d=1}^{D} e^{Y_{id}}},$$

where $k \in \{1, \dots D\}$ and where $i \in \{1, \dots, \#U\}$ (and thus, the dimensions of $\mathbf{X}$ and $\mathbf{Y}$ are the same). The outputs of the softmax function are in the range $(0, 1)$, and per unit i, the output elements sum to one (i.e., $\sum_{k=1}^{D} \sigma(Y_{ik}) = 1$ for every unit i). Therefore, every output unit of the softmax function can be said to encode a categorical probability distribution over its dimensions.

The softmax function and the logarithmic loss can be combined into what we will call the softmax-loss function $\ell = l \circ \sigma$, which gives us the following error function (note that the dependence of E on $\mathbf{w}$ lies in the fact that $\mathbf{Y}^n = \mathbf{f}(\mathbf{X}^n, \mathbf{w})$ is a function of $\mathbf{w}$):

$$E(\mathbf{w}) = \sum_{n=1}^{N} E^n(\mathbf{w}) := \sum_{n=1}^{N} \ell(\mathbf{Y}^n, c^n)$$

$$\ell(\mathbf{Y}, c) = -\sum_{i=1}^{\#U'} \log \frac{e^{Y_{ic}}}{\sum_{d=1}^{D} e^{Y_{id}}} = -\sum_{i=1}^{\#U'} \left(Y_{ic} - \log \sum_{d=1}^{D} e^{Y_{id}} \right).$$

As stated in Section B.3, to use the back-propagation algorithm we need to know the derivative of this softmax-loss function ℓ with respect to the network output $\mathbf{Y}$.

For an output element Y_{ik}, there is a case distinction to be made between the case $k = c$ and $k \neq c$, because of the special mention that Y_{ic} gets in the formula for ℓ. However, this case distinction is conveniently expressed using the Kronecker delta, making sure we do not need a case distinction in our final formula:

$$\frac{\partial Y_{ic}}{\partial Y_{ik}} = \delta_{ck} \equiv \begin{cases} 1 & \text{if } c = k \\ 0 & \text{if } c \neq k. \end{cases}$$

The derivative of the log term is independent of the question whether $c = d$, and thus can be expressed conveniently in the following form:

$$\frac{\partial}{\partial Y_{ik}} \left(\log \sum_{d=1}^{D} e^{Y_{id}} \right) = \frac{1}{\sum_{d=1}^{D} e^{Y_{id}}} \cdot e^{Y_{ik}} = \frac{e^{Y_{ik}}}{\sum_{d=1}^{D} e^{Y_{id}}} = [\sigma(\mathbf{Y})]_{ik}$$

We can now put these two expressions together to find our final error derivative, in matrix form:

$$\frac{\partial \ell}{\partial \mathbf{Y}} = -\left(\mathbf{1}^{\#U'} \cdot (\mathbf{e}_c^D)^T - \sigma(\mathbf{Y}) \right) = \sigma(\mathbf{Y}) - \mathbf{1}^{\#U'} \cdot (\mathbf{e}_c^D)^T,$$

where $\mathbf{1}^{\#U'} \in \mathbb{R}^{\#U'}$ is the all-ones vector and $\mathbf{e}_c^D \in \{0,1\}^D$ the c-th indicator vector ($[\mathbf{e}_c^D]_k = \delta_{ck}$, i.e. the c-th element is a 1 and the rest are zero). Note that, therefore, $\mathbf{1}^{\#U'} \cdot (\mathbf{e}_c^D)^T$ is a matrix with the same dimensions as $\mathbf{Y}$ that has a 1 in all elements of the c-th column, and a 0 everywhere else.

B.5 Convolutional neural networks

Convolutional neural networks form a specific subclass of the neural networks described in Section B.1. These are characterised by having one or more convolutional layers (see Section B.6), as well as having an image-like structure in their input. They are especially fit for perceptual tasks, such as object classification and facial recognition. Inspired by visual tasks, but also applicable to other domains, convolutional neural networks employ in specific a rectangular grid structure (cf. an image) in the configuration of its units (including, crucially, the network's input and output).

Formally, their input is a matrix $\mathbf{X} \in \mathbb{R}^{H \times W \times D}$, and the output is $\mathbf{Y} \in \mathbb{R}^{H' \times W' \times D}$. For example, for an RGB image, we have $H \times W$ take on the image's height and width, and $D = 3$ to allow for each one 'pixel' (input unit) to process each of the R, G and B channels separeately. As stated before in Section B.1, we will also refer to the vectorisations of these matrices, $\mathbf{x} = \text{vec}(\mathbf{X}) \in \mathbb{R}^{HWD}$ and $\mathbf{y} = \text{vec}(\mathbf{Y}) \in \mathbb{R}^{H'W'K}$.

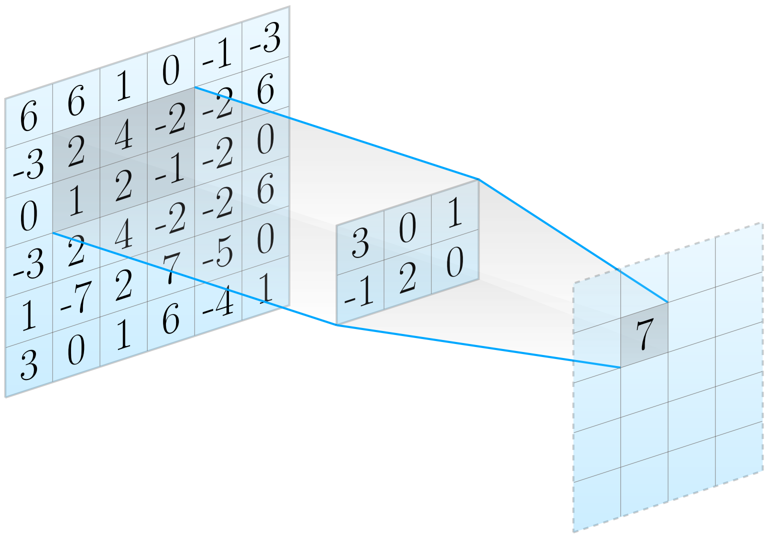


Figure 8: An example of a convolutional layer. The output is computed as $3 \cdot 2 + 0 \cdot 4 + 1 \cdot -2 - 1 \cdot 1 + 2 \cdot 2 + 0 \cdot -1 = 7$. No padding is used, and the stride is 1. Adapted from Churchman (2015) with permission.

Recall that the vec operator is invertible for given dimensions, so these representations can be converted back and forth where needed.

Note also that this formalism is compatible with our earlier definition of a neural network, where we had $\mathbf{X} \in \mathbb{R}^{\#U \times D}$ and similarly $\mathbf{Y} \in \mathbb{R}^{\#U' \times K}$: we can simply take the number of units to be $\#U = HW$ and $\#U' = H'W'$, and then the convolutional inputs and outputs can be invertibly converted to this representation, making the two representations of convolutional neural networks equivalent.

B.6 Specific layers and their derivatives

A neural network's behaviour is implemented by, and therefore defined by, its specific layers. Therefore, the layers used in convolutional neural networks deserve proper description. As noted in Section B.3, to use back-propagation we also need to be able to compute $\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_i}$ for all types of layers in our network, as well as $\frac{\partial \mathbf{y}_i}{\partial \mathbf{w}_i}$ for those types of layers we want to learn weights for. Thus, we also need to derive these expressions for layers of interest to convolutional neural networks in particular.

B.6.1 Convolutional layers

The distinguishing feature of a convolutional neural network is that it has one or more convolutional layers. A convolutional layer uses as weights a filter bank $\mathbf{F} \in \mathbb{R}^{H_F \times W_F \times D \times K}$ (with the vectorisation $\text{vec}(\mathbf{F}) =: \mathbf{f} \in \mathbb{R}^{H_F W_F D K}$) containing K filters as matrices from $\mathbb{R}^{H_F \times W_F \times D}$, that can be learnt using back-propagation. Furthermore, they have an array of biases $\mathbf{b} \in \mathbb{R}^K$, containing one bias per output dimension, which can also be trained.

Convolutional layers compute the output $\mathbf{Y} \in \mathbb{R}^{H' \times W' \times K}$ as the convolution of the input $\mathbf{X} \in \mathbb{R}^{H \times W \times D}$ with the filters from its filter bank $\mathbf{F}$, and using its biases $\mathbf{b}$ specified above.

Normally, $H' = H - H_F + 1$ and $W' = W - W_F + 1$, but it is also possible to zero-pad the data $\mathbf{X}$ at the edges. If we pad the data with zeros P_t at the top, P_b at the bottom, P_l to the left and P_r to the right, we will instead have the relations $H' = H - H_F + P_t + P_b + 1$ and $W' = W - W_F + P_l + P_r + 1$ on the output size of our layer. As a rule, P_t, P_b, P_l and P_r are all non-negative and $P_t + P_b < H_F, P_l + P_r < W_F$ must hold. Thus, we always have $H' \leq H$ and $W' \leq W$. We will ignore padding in our further derivations, as it can be interpreted as simply extending the input matrix $\mathbf{X}$ with zeros, and running this through the convolutional layer instead. This removes the need to take account of whether padding is used, as far as the relevant derivatives are concerned.

We shift a given filter k from $\mathbf{F}$ over all possible base positions in $\mathbf{X}$, and record elements of $\mathbf{Y}$ as the sum of the element-wise multiplications in all dimensions of the input and the filter. Formally, we have that

$$Y_{i',j',k} = b_k + \sum_{i_F=1}^{H_F} \sum_{j_F=1}^{W_F} \sum_{d=1}^{D} F_{i_F,j_F,d,k} \cdot X_{i'+i_F-1,j'+j_F-1,d},$$

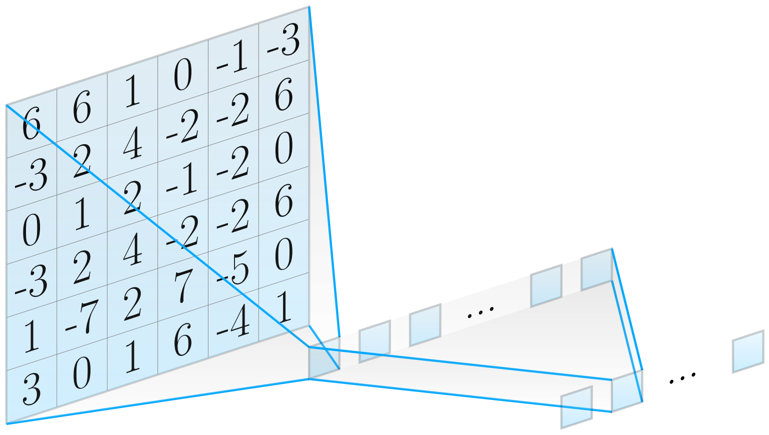


Figure 9: An example of a fully-connected (convolutional) layer, where each output element depends on every single element of the previous layer. No padding is used. Used with permission from Churchman (2015).

where $1 \leq i' \leq H'$, $1 \leq j' \leq W'$ and $1 \leq k \leq K$ code for the possible base positions and filters (not accounting for padding). See Figure 8.

Convolution also supports using stride, or subsampling of the input. It takes two stride parameters, δ_h and δ_w, which are integers greater than or equal to one. The stride parameters have two major effects. Firstly, certain elements of $\mathbf{X}$ are generally skipped as base positions for the convolutional filter when a stride parameter is larger than 1. The base positions that *are* recorded are spaced equally from each other, with $\delta_h \times \delta_w$ rectangles in-between adjacent base positions. Instead of the above formula, we then get

$$Y_{i',j',k} = b_k + \sum_{i_F=1}^{H_F} \sum_{j_F=1}^{W_F} \sum_{d=1}^{D} F_{i_F,j_F,d,k} \cdot X_{(i'-1)\delta_h+i_F,(j'-1)\delta_w+j_F,d}.$$

Thus, an output element $Y_{i',j',k}$ corresponds with the base position $X_{(i'-1)\delta_h+1,(j'-1)\delta_w+1,d}$ (the input dimension d is summed over). Note this reduces nicely to the initial formula of convolution in the case that $\delta_h = \delta_w = 1$. The second effect of the stride parameters is that they directly affect the output size $H' \times W'$, to account for the spread of convolution base positions. The output size is

$$H' = \left\lfloor \frac{H - H_F + P_t + P_b}{\delta_h} \right\rfloor + 1, W' = \left\lfloor \frac{W - W_F + P_l + P_r}{\delta_w} \right\rfloor + 1.$$

Though it is important to know how stride works, we do not actually use it for convolutional layers in our networks, for reasons explained in Section 4.2. Therefore, in this section we assume (without loss of generality, within our context) that $\delta_h = \delta_w = 1$. However, stride will be important in max-pooling layers, see Section B.6.2.

Finally, note that fully connected layers are a special case of convolutional layers, with $H = H_F, W = W_F$ (and therefore $H' = W' = 1$). The effect of this is that the 1×1 'images' of the output depend on all elements in $\mathbf{X}$. See Figure 9 for an example.

Now, given our element-wise definition of convolution, it is not easy to directly take the derivative of this expression, since it would only give us an (inefficiently computable) element-wise derivative. Therefore, it is easier to use another, equivalent definition of the convolutional layer, which is essentially a single matrix multiplication (Vedaldi and Lenc, 2014). Define ϕ to be the `im2row` operator, which takes all $H_F \times W_F$ patches from its input and stores them as rows of its output matrix. Formally, $\phi : \mathbb{R}^{H \times W \times D} \to \mathbb{R}^{H'W' \times H_F W_F D}$ is defined by

$$[\phi(\mathbf{X})]_{p,q} \underset{(i,j,d)=t(p,q)}{=} X_{i,j,d},$$

where t is defined such that $p = i' + H'(j'-1)$, $q = i_F + H_F(j_F - 1) + H_F W_F(d-1)$, $i = i' + i_F - 1$ and $j = j' + j_F - 1$. Respective bounds apply ($1 \leq i \leq H$, $1 \leq j \leq W$, etc., like before). Note that this is well-defined, i.e. i, j and d can be determined uniquely from p and q.

It is easy to see that ϕ is a linear transformation, given that it only copies fixed elements of its input, and does not alter them. Linear algebra tells us that, therefore, there must be a matrix $\mathbf{H} \in \mathbb{R}^{H'W'H_F W_F D \times HWD}$ such that

$$\text{vec}(\phi(\mathbf{X})) = \mathbf{H} \cdot \text{vec}(\mathbf{X}).$$

Given that there is such a matrix, we can also define the `row2im` operator $\phi^* : \mathbb{R}^{H'W' \times H_F W_F D} \to \mathbb{R}^{H \times W \times D}$ to be given by

$$\mathrm{vec}(\phi^*(\mathbf{M})) = \mathbf{H}^T \cdot \mathrm{vec}(\mathbf{M}).$$

With the above vectorised form, we can also write ϕ^* as follows:

$$[\phi^*(\mathbf{M})]_{i,j,d} = \sum_{\substack{p,q \\ t(p,q)=(i,j,d)}} M_{p,q}.$$

Now that we have this definition of ϕ, we can describe convolution as a single matrix multiplication. However, since matrix multiplication only works on 2D matrices, we need to reshape the relevant matrices. We reshape $\mathbf{F}$ into $\mathbf{F'} \in \mathbb{R}^{H_F W_F D \times K}$, and reshape $\mathbf{Y}$ into $\mathbf{Y'} \in \mathbb{R}^{H'W' \times K}$. Note that this is an invertible transformation, and that we can therefore formulate convolution in terms of $\mathbf{F'}$ and $\mathbf{Y'}$ as follows (where $\mathbf{1}^{H'W'} \in \mathbb{R}^{H'W'}$ is the all-ones vector of length $H'W'$):

$$\mathbf{Y'} = \mathbf{1}^{H'W'} \cdot \mathbf{b}^T + \phi(\mathbf{X}) \cdot \mathbf{F'}.$$

Now we want to derive the partial derivatives of $\mathbf{Y}$ with respect to $\mathbf{X}$ and $\mathbf{F}$, as well as $\mathbf{b}$. However, matrix-by-matrix derivatives in general are not defined, so for the former two, we will instead work with the (vectorised) vector-by-vector derivatives $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{y}}{\partial \mathbf{f}}$. We can base ourselves on the vectorised version of the equation above:

$$\mathbf{y} = \mathrm{vec}(\mathbf{Y'}) = \mathrm{vec}(\mathbf{1}^{H'W'} \cdot \mathbf{b}^T + \phi(\mathbf{X}) \cdot \mathbf{F'}) = \mathrm{vec}(\mathbf{1}^{H'W'} \cdot \mathbf{b}^T) + \mathrm{vec}(\phi(\mathbf{X}) \cdot \mathbf{F'}).$$

Let us first derive $\frac{\partial \mathbf{y}}{\partial \mathbf{b}}$. The second half of $\mathbf{y}$, $\phi(\mathbf{X}) \cdot \mathbf{F'}$, is independent from $\mathbf{b}$, so we can leave it out of the equation:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \frac{\partial}{\partial \mathbf{b}} \Big(\mathrm{vec}(\mathbf{1}^{H'W'} \cdot \mathbf{b}^T) + \mathrm{vec}(\phi(\mathbf{X}) \cdot \mathbf{F'}) \Big) = \frac{\partial}{\partial \mathbf{b}} \mathrm{vec}(\mathbf{1}^{H'W'} \cdot \mathbf{b}^T).$$

If we look at a single element of this derivative, we get

$$\frac{\partial Y'_{ij}}{\partial b_k} = \delta_{jk}.$$

This is already a sufficient expression for our derivative $\frac{\partial \mathbf{y}}{\partial \mathbf{b}}$, since the mapping from $\mathbf{Y'}$ to $\mathbf{y}$ is invertible. However, later in this section we will adopt a more convenient form for backprop.

Let us now derive $\frac{\partial \mathbf{y}}{\partial \mathbf{f}}$ and $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$. The first half of $\mathbf{y}$, $\mathrm{vec}(\mathbf{1}^{H'W'} \cdot \mathbf{b}^T)$, is independent from both $\mathbf{x}$ and $\mathbf{f}$, so it is irrelevant for these two derivatives. Hence, we can assume without loss of generality that all biases are zero ($\mathbf{b} = \mathbf{0}$), given that the relevant derivatives do not change.

For the following derivations we use two identities relating to the vec operator, taken from Graham (1981). Let $\mathbf{A} \in \mathbb{R}^{k \times l}$, $\mathbf{B} \in \mathbb{R}^{l \times m}$. Then we have

$$\mathrm{vec}(\mathbf{A} \cdot \mathbf{B}) = (\mathbf{I}_m \otimes \mathbf{A}) \cdot \mathrm{vec}(\mathbf{B}) = (\mathbf{B}^T \otimes \mathbf{I}_k) \cdot \mathrm{vec}(\mathbf{A}),$$

where $\otimes$ denotes the Kronecker product (for a formal definition, see Graham (1981)) and $\mathbf{I}_n$ represents the $n \times n$ identity matrix. Applying this to the equation for our convolution layer gives us that

$$\mathbf{y} = (\mathbf{I}_K \otimes \phi(\mathbf{X})) \cdot \mathrm{vec}(\mathbf{F'}) = \big((\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}\big) \cdot \mathrm{vec}(\phi(\mathbf{X})).$$

We can now derive $\frac{\partial \mathbf{y}}{\partial \mathbf{f}} = \frac{\partial \mathbf{y}}{\partial \mathrm{vec}(\mathbf{F'})}$. Since $\mathbf{I}_K \otimes \phi(\mathbf{X})$ does not depend on $\mathbf{F'}$, we get that

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}} = \frac{\partial}{\partial \mathrm{vec}(\mathbf{F'})} \left((\mathbf{I}_K \otimes \phi(\mathbf{X})) \cdot \mathrm{vec}(\mathbf{F'}) \right) = (\mathbf{I}_K \otimes \phi(\mathbf{X})) \cdot \frac{\partial \mathrm{vec}(\mathbf{F'})}{\partial \mathrm{vec}(\mathbf{F'})} = \mathbf{I}_K \otimes \phi(\mathbf{X}),$$

since $\frac{\partial \mathrm{vec}(\mathbf{F'})}{\partial \mathrm{vec}(\mathbf{F'})} = \mathbf{I}$, the identity matrix (of suitable size).

Now we will derive $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathrm{vec}(\mathbf{X'})}$. For this we use the second vectorisation identity mentioned above, together with the fact that $(\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}$ does not depend on $\mathbf{X}$:

$$\begin{aligned}
\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathrm{vec}(\mathbf{X})} &= \frac{\partial}{\partial \mathrm{vec}(\mathbf{X})} \big((\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}\big) \cdot \mathrm{vec}(\phi(\mathbf{X})) \\
&= \big((\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}\big) \cdot \frac{\partial \mathrm{vec}(\phi(\mathbf{X}))}{\partial \mathrm{vec}(\mathbf{X})} \\
&= \big((\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}\big) \cdot \mathbf{H},
\end{aligned}$$

where $\mathbf{H}$ is the matrix that defines ϕ by $\text{vec}(\phi(\mathbf{X})) = \mathbf{H} \cdot \text{vec}(\mathbf{X})$.

In principle these expressions are enough to do gradient descent, but as noted in Section B.3, we still need to multiply by $\frac{\partial E^n}{\partial \mathbf{y}^n}$ to get the actual gradient. We could do this by repeated matrix multiplication, but there is a more compact way to write this down without needing the relevant variables' vectorisations. By doing so, we can write the above partial derivatives directly in terms of $\frac{\partial E^n}{\partial \mathbf{Y'}^n}$. The $\cdot^n$ superscript is left out for convenience.

First, let us examine $\frac{\partial E}{\partial \mathbf{b}}$:

$$\frac{\partial E}{\partial b_k} = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial b_k} = \sum_{i=1}^{H'W'} \sum_{j=1}^{K} \frac{\partial E}{\partial Y_{ij}} \delta_{jk} = \sum_{i=1}^{H'W'} \frac{\partial E}{\partial Y_{ik}}.$$

We can easily write this as a matrix-vector multiplication, where we multiply over the $H'W'$ base filter positions:

$$\frac{\partial E}{\partial \mathbf{b}} = \left(\frac{\partial E}{\partial \mathbf{Y'}}\right)^T \cdot \mathbf{1}^{H'W'}.$$

Now, let us turn to $\frac{\partial E}{\partial \mathbf{f}}$. So far we have found that

$$\frac{\partial E}{\partial \mathbf{f}} = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{f}} = \frac{\partial E}{\partial \mathbf{y}} \cdot (\mathbf{I}_K \otimes \phi(\mathbf{X})).$$

Let us examine the transpose of the right-hand expression, using several properties of the Kronecker product from Graham (1981). Note that, while they contain the same partial derivatives in the same order, $\frac{\partial E}{\partial \mathbf{y}} = \frac{\partial E}{\partial \text{vec}(\mathbf{Y'})}$ is a row vector, whereas $\text{vec}\left(\frac{\partial E}{\partial \mathbf{Y'}}\right)$ is a column vector. Hence, we know that $\frac{\partial E}{\partial \mathbf{y}} = \left(\text{vec}\left(\frac{\partial E}{\partial \mathbf{Y'}}\right)\right)^T$.

$$\begin{aligned}
\left(\frac{\partial E}{\partial \mathbf{f}}\right)^T &= \left(\frac{\partial E}{\partial \mathbf{y}} \cdot (\mathbf{I}_K \otimes \phi(\mathbf{X}))\right)^T \\
&= (\mathbf{I}_K \otimes \phi(\mathbf{X}))^T \cdot \left(\frac{\partial E}{\partial \mathbf{y}}\right)^T \\
&= (\mathbf{I}_K{}^T \otimes \phi(\mathbf{X})^T) \cdot \text{vec}\left(\frac{\partial E}{\partial \mathbf{Y'}}\right) \\
&= (\mathbf{I}_K \otimes \phi(\mathbf{X})^T) \cdot \text{vec}\left(\frac{\partial E}{\partial \mathbf{Y'}}\right).
\end{aligned}$$

However, applying our vectorisation identity in reverse and taking the transpose tells us that this is equivalent to

$$\frac{\partial E}{\partial \mathbf{f}} = \left(\text{vec}\left(\phi(\mathbf{X})^T \cdot \frac{\partial E}{\partial \mathbf{Y'}}\right)\right)^T.$$

Given that $\frac{\partial E}{\partial \mathbf{f}} = \left(\text{vec}\left(\frac{\partial E}{\partial \mathbf{F'}}\right)\right)^T$ by the same argument as for $\mathbf{y}$ and $\mathbf{Y'}$, we get that

$$\frac{\partial E}{\partial \mathbf{f}} = \left(\text{vec}\left(\frac{\partial E}{\partial \mathbf{F'}}\right)\right)^T = \left(\text{vec}\left(\phi(\mathbf{X})^T \cdot \frac{\partial E}{\partial \mathbf{Y'}}\right)\right)^T.$$

Therefore, by taking the transpose and using invertibility of vec (for fixed dimensions), we get

$$\frac{\partial E}{\partial \mathbf{F'}} = \phi(\mathbf{X})^T \cdot \frac{\partial E}{\partial \mathbf{Y'}}.$$

We can carry out a similar derivation for $\frac{\partial E}{\partial \mathbf{x}}$. So far we have

$$\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{E}}{\partial \mathbf{y}} \cdot \left((\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}\right) \cdot \mathbf{H},$$

where $\mathbf{H}$ was the matrix for which $\text{vec}(\phi(\mathbf{X})) = \mathbf{H} \cdot \text{vec}(\mathbf{X})$. We again examine the transpose of this

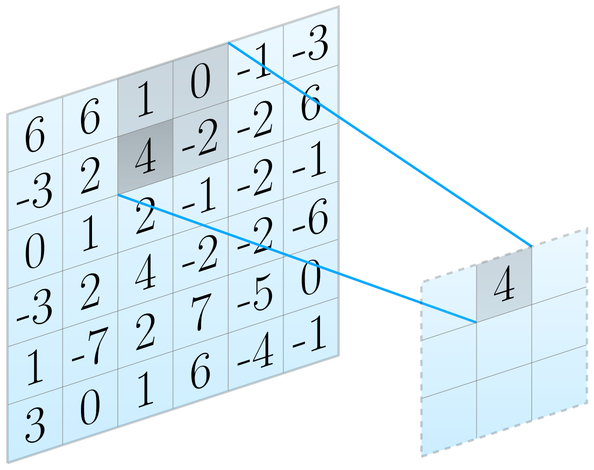


Figure 10: An example of a max-pooling layer with a 2×2 pooling region and stride 2. No padding is used. The output is computed as the maximum of the elements in the pooling region. Adapted from Churchman (2015) with permission.

expression:

$$\begin{aligned}\left(\frac{\partial E}{\partial \mathbf{x}}\right)^T &= \left(\frac{\partial \mathbf{E}}{\partial \mathbf{y}} \cdot \left((\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}\right) \cdot \mathbf{H}\right)^T \\ &= \mathbf{H}^T \cdot \left((\mathbf{F'})^T \otimes \mathbf{I}_{H'W'}\right)^T \cdot \left(\frac{\partial \mathbf{E}}{\partial \mathbf{y}}\right)^T \\ &= \mathbf{H}^T \cdot \left(\left((\mathbf{F'})^T\right)^T \otimes \mathbf{I}_{H'W'}{}^T\right) \cdot \left(\frac{\partial \mathbf{E}}{\partial \mathbf{y}}\right)^T \\ &= \mathbf{H}^T \cdot \left(\left((\mathbf{F'})^T\right)^T \otimes \mathbf{I}_{H'W'}\right) \cdot \operatorname{vec}\left(\frac{\partial E}{\partial \mathbf{Y'}}\right).\end{aligned}$$

For this, we can use the second identity we had for vectorisation, giving us

$$\left(\frac{\partial E}{\partial \mathbf{x}}\right)^T = \mathbf{H}^T \cdot \operatorname{vec}\left(\frac{\partial E}{\partial \mathbf{Y'}} \cdot (\mathbf{F'})^T\right).$$

As above, we have that $\frac{\partial E}{\partial \mathbf{x}} = \operatorname{vec}\left(\frac{\partial E}{\partial \mathbf{X}}\right)^T$. However, this is where the `row2im` operator ϕ^*, which we defined earlier, comes into play. We defined ϕ^* such that $\operatorname{vec}(\phi^*(\mathbf{M})) = \mathbf{H}^T \operatorname{vec}(\mathbf{M})$, and we can apply this here:

$$\left(\frac{\partial E}{\partial \mathbf{x}}\right)^T = \operatorname{vec}\left(\frac{\partial E}{\partial \mathbf{X}}\right) = \operatorname{vec}\left(\phi^*\left(\frac{\partial E}{\partial \mathbf{Y'}} \cdot (\mathbf{F'})^T\right)\right).$$

Thus, we arrive at the equation

$$\frac{\partial E}{\partial \mathbf{X}} = \phi^*\left(\frac{\partial E}{\partial \mathbf{Y'}} \cdot (\mathbf{F'})^T\right).$$

B.6.2 Max-pooling layers

Max-pooling layers are used in convolutional neural networks to decrease the input size in higher-level layers. They use almost the same rules for input and output size as convolutional layers: the input $\mathbf{X} \in \mathbb{R}^{H \times W \times D}$ is transformed into output $\mathbf{Y} \in \mathbb{R}^{H' \times W' \times D}$, using a pooling region of size $H_F \times W_F$, where H, H_F, H' are related as described in Section B.6.1. See also Figure 10.

As in that section, we ignore padding without loss of generality, as we incorporate padding into the input $\mathbf{X}$. Given that we use max-pooling, however, padding simply with zeros can cause issues when all elements are negative. Therefore, we instead pad with $-\infty$, so no actual element of $\mathbf{X}$ is smaller our padding values. In most cases, pooling layers will have stride larger than 1, so we do keep stride in the equations here, as opposed to our assumption that they were always 1 for convolutional layers ,in Section B.6.1.

Then, we arrive at the following expression for max-pooling:

$$Y_{i',j',d} = \max_{\substack{1 \leq i_F \leq H_F \\ 1 \leq j_F \leq W_F}} X_{(i'-1)\delta_h + i_F, (j'-1)\delta_w + j_F, d}.$$

Now, it must be mentioned that the max operator is not differentiable for all inputs $\mathbf{x}$. Specifically, when (at least) two elements i and j of $\mathbf{x}$ inside the same pooling region are exactly equal to each other and to the maximum of their pooling region, $\frac{\partial \mathbf{y}}{\partial x_i}$ and $\frac{\partial \mathbf{y}}{\partial x_j}$ are undefined. This is because, if we increase x_i by any amount, then $\max(x_i, x_j) = x_i$, and thus the derivative with respect to x_i is equal to one. However, if we instead decrease x_i by any amount, we get $\max(x_i, x_j) = x_j$, and thus the derivative with respect to x_i is equal to zero. Since the derivative would be both equal to one and zero if it existed, it cannot exist and max is not differentiable for ties. The upside, however, is that this rarely ever happens, and these cases can therefore be safely ignored (a small error in one gradient update does not cause large problems, as it will be straightened out by many more updates to follow).

As before, we want to express this in matrix form, so that we can compute the total derivative $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$. For this, it useful to remember that all the max operator does is select one of the matrix elements it evaluates. Therefore, there must exist a 'selector matrix' $\mathbf{S}(\mathbf{x}) \in \{0,1\}^{H'W'D \times HWD}$(Vedaldi and Lenc, 2014), for which it holds that

$$\mathbf{y} = \mathbf{S}(\mathbf{x}) \cdot \mathbf{x}.$$

$[\mathbf{S}(\mathbf{x})]_{ij}$ contains a 1 if x_j is the maximum element of the pooling region of y_i, and a 0 otherwise, and is unique, since ties are ignored. If ties are allowed (i.e., in practical applications), we require that every row of $\mathbf{S}(\mathbf{x})$ contains exactly one 1 for *a* maximum element of the corresponding pooling region, and that all other elements are 0. This does make $\mathbf{S}(\mathbf{x})$ non-unique, but this is not a problem as long as these ties are broken in a consistent way (which is reflected in $\mathbf{S}(\mathbf{x})$).

Now, let x_j be an element of $\mathbf{x}$, inside the pooling region whose maximum is recorded in y_i. x_j is either the maximum element of this pooling region, M, or it is smaller than M. Thus, the following expression holds at least in some neighbourhood of x_j (since ties are ignored):

$$y_i = \begin{cases} x_j & \text{if } x_j = M \\ M & \text{if } x_j < M \end{cases}.$$

This expression is easy to differentiate:

$$\frac{\partial y_i}{\partial x_j} = \begin{cases} 1 & \text{if } x_j = M \\ 0 & \text{if } x_j < M \end{cases}.$$

For x_j outside the pooling region of y_i, $\frac{\partial y_i}{\partial x_j}$ is always zero. Thus, all $\frac{\partial y_i}{\partial x_j}$ all zero, except where x_j is the maximum of the pooling region of y_i. However, this is exactly our definition of $\mathbf{S}(\mathbf{x})$. This gives us very simply:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{S}(\mathbf{x}).$$

Given this simple expression of $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$, we easily get a simple expression for $\frac{\partial E}{\partial \mathbf{x}}$:

$$\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \mathbf{S}(\mathbf{x}).$$

B.6.3 Rectified linear unit (relu) layers

A rectified linear unit, or relu, layer applies the following non-linear activation function to every input element:

$$Y_{ijd} = \text{relu}(X_{ijd}) = \max(X_{ijd}, 0).$$

The dimensions of $\mathbf{Y}$ and $\mathbf{X}$ are the same.

Analogously to the max-pooling layer of Section B.6.2, $\frac{\partial y_i}{\partial x_i}$ is undefined (i.e., y_i is not differentiable with respect to x_i) when $x_i = 0$, but these cases can be ignored. Similarly to max-pooling, there exists a diagonal matrix $\mathbf{S}(\mathbf{x}) \in \{0,1\}^{HWD \times HWD}$ with $\mathbf{y} = \mathbf{S}(\mathbf{x}) \cdot \mathbf{x}$, where $\mathbf{S}(\mathbf{x})$ is defined as:

$$[\mathbf{S}(\mathbf{x})]_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } x_j > 0 \\ 0 & \text{otherwise} \end{cases},$$

and as was the case for max-pooling layers, we have

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{S}(\mathbf{x}), \frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \mathbf{S}(\mathbf{x}).$$

B.6.4 Dropout layers

Dropout layers are used to fight overfitting in our networks. For an input $\mathbf{X}$ with vectorisation $\mathbf{x} \in \mathbb{R}^{HWD}$, a dropout layer uses a 'mask' $\mathbf{s} \in \{0,1\}^{HWD}$ whose elements indicate whether the corresponding element in $\mathbf{x}$ is allowed to continue onwards, or whether it is blocked (dropped out), as follows:

$$y_i = x_i \cdot s_i.$$

This again gives us a very simple matrix form, as with the rectified linear units of Section B.6.3. Define the diagonal matrix $\mathbf{S}(\mathbf{s}) \in \{0,1\}^{HWD \times HWD}$ as

$$[\mathbf{S}(\mathbf{s})]_{ij} = \begin{cases} s_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

Then, again, we have

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{S}(\mathbf{s}), \frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \mathbf{S}(\mathbf{s}).$$

B.6.5 Softmax-loss layers

In Section B.4, we have already defined the softmax-loss function ℓ of the network output for a given input, and the correct class label c for that input. In that section, the derivative was also already evaluated, so most of the work here has already been done before. For completeness, the results of Section B.4 are repeated here, rephrased to use the same notation as the previous sections, about other layers.

$$y = \ell(\mathbf{X}, c) = -\sum_{i=1}^{H} \sum_{j=1}^{W} \left(X_{ijc} - \log \sum_{k=1}^{D} e^{X_{ijk}} \right),$$

$$\frac{\partial \ell}{\partial \mathbf{X}} = \sigma(\mathbf{X}) - \mathbf{1}^{HW} \cdot (\mathbf{e}_c^D)^T,$$

where $\mathbf{1}^{HW}$ is the all-ones vector of length HW and $\mathbf{e}_c^D \in \{0,1\}^D$ the c-th indicator vector ($[\mathbf{e}_c^D]_d = \delta_{cd}$, i.e. the c-th element is a 1 and the rest are zero), and σ was the softmax function where

$$[\sigma(\mathbf{X})]_{ijd} = \frac{e^{X_{ijd}}}{\sum_{k=1}^{D} e^{X_{ijk}}}.$$

The only thing of relevance that has not been done in Section B.4 is to derive a direct expression for $\frac{\partial E}{\partial \mathbf{x}}$:

$$\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial \mathbf{x}} = \frac{\partial E}{\partial y} \left(\sigma(\mathbf{X}) - \mathbf{1}^{HW} \cdot (\mathbf{e}_c^D)^T \right).$$

B.7 Autoencoders

The previous few sections have been about convolutional neural networks, specifically used for classification. However, these are trained in a supervised manner, and we want to learn a representation of our data in an unsupervised way, using autoencoders. As explained in Section 4, autoencoders are trained such that their output is as close as possible to their input, as defined by the Euclidean error. In other words, instead of using the softmax-loss classification error of Section B.4, we use the following error function:

$$E^k(\mathbf{w}) = ||\mathbf{y}^k - \mathbf{x}^k||^2 = \left(\sqrt{\sum_{j=1}^{HWD} (y_j^k - x_j^k)^2} \right)^2 = \sum_{j=1}^{HWD} (y_j^k - x_j^k)^2.$$

Again, the dependence on $\mathbf{w}$ is caused by the fact that $\mathbf{y}^k = \mathbf{f}(\mathbf{x}^k, \mathbf{w})$ depends on $\mathbf{w}$.

To be able to use back-propagation, we need to evaluate the derivative $\frac{\partial E}{\partial \mathbf{y}}$ (superscripts $\cdot^k$ are left out for convenience, as they are not relevant). Let us first examine a single element of this vector:

$$\frac{\partial E}{\partial y_i} = \frac{\partial}{\partial y_i} \left(\sum_{j=1}^{HWD} (y_j^k - x_j^k)^2 \right) = \frac{\partial}{\partial y_i} \left((y_i^k - x_i^k)^2 \right) = 2(y_i^k - x_i^k).$$

This gives us the following expression for the error gradient:

$$\frac{\partial E}{\partial \mathbf{y}} = 2(\mathbf{y} - \mathbf{x}).$$

Note that the constant factor of 2 can be absorbed into the learning rate, and is therefore irrelevant. It is wholly equivalent to use $\mathbf{y} - \mathbf{x}$ as error gradient, while doubling the learning rate.

B.8 Inverting network layers

As noted in Section 4.2, we need a few additional definitions to be able to invert convolutional and pooling layers.

B.8.1 Convolutional transpose

For convolution, we only need to define the convolutional transpose, as either this or simply learning new filters is enough to invert convolution (see Section 4.2.1). The convolutional transpose can be formulated as a linear transformation $T : \mathbb{R}^{H\times W\times D\times K} \to \mathbb{R}^{H\times W\times K\times D}$, where

$$[T(\mathbf{F})]_{i,j,k,d} = F_{H-i+1,W-j+1,d,k},$$

where $1 \leq i \leq H$ and $1 \leq j \leq W$ as before. In other words, the input and output channels are permuted, and for each input-output channel pair the filters are flipped vertically and horizontally.

For the convolution-transpose pseudo-layer, simply copy the corresponding convolution layer earlier in the network, but instead of using its filters $\mathbf{F}$ for the pseudo-layer, use $T(\mathbf{F})$. Biases need to be created anew, as there is a (possibly) different output dimension. The derivatives of this pseudo-layer can be computed with the formulas derived for regular convolutional layers in Section B.6.1.

B.8.2 Switch unpooling

To invert pooling layers, however, we need to define an all-new unpooling layer. As noted in Section 4.2.2, there are two techniques for this, and one of these is switch unpooling. Given is a corresponding max-pooling region earlier in the network, which used a pooling region of size $H_F \times W_F$ and stride parameters δ_h, δ_w. For an input $\mathbf{X} \in \mathbb{R}^{H\times W\times D}$, the output is $\mathbf{Y} \in \mathbb{R}^{H'\times W'\times D}$ such that

$$\left\lfloor \frac{H' - H_F + P_t + P_b}{\delta_h} \right\rfloor + 1 = H, \left\lfloor \frac{W' - W_F + P_l + P_r}{\delta_w} \right\rfloor + 1 = W$$

holds. Note this is not unique when $\delta_h > 1$ or $\delta_w > 1$. However, we use as additional constraint that the output size of this unpooling layer equals the input size of the corresponding layer (this includes the above requirement, which is listed for completeness). Padding is incorporated in the input as in Section B.6.1.

We first need to slightly adapt our definition of max-pooling, so that it saves the 'switches' when it runs. This does not change its derivative, but gives it extra outputs. See Section 4.2.2.

Our definition of max-pooling was that it gave an output $\mathbf{Y} \in \mathbb{R}^{H\times W\times D}$ (note the output of the corresponding max-pooling layer must the same size as the input of switch unpooling), such that

$$Y_{i,j,d} = \max_{\substack{1\leq i_F\leq H_F \\ 1\leq j_F\leq W_F}} X_{(i-1)\delta_h+i_F,(j-1)\delta_w+j_F,d}.$$

Now define two 'switch' matrices $\mathbf{P}, \mathbf{Q} \in \mathbb{R}^{H\times W\times D}$, such that

$$P_{i,j,d} = \operatorname*{arg\,max}_{1\leq i_F\leq H_F} \max_{1\leq j_F\leq W_F} X_{(i-1)\delta_h+i_F,(j-1)\delta_w+j_F,d},$$

$$Q_{i,j,d} = \operatorname*{arg\,max}_{1\leq j_F\leq W_F} \max_{1\leq i_F\leq H_F} X_{(i-1)\delta_h+i_F,(j-1)\delta_w+j_F,d}.$$

In other words, $\mathbf{P}_{i,j,d}$ and $\mathbf{Q}_{i,j,d}$ simply store the indices i_F and j_F (respectively) for which the corresponding element in the pooling region is the maximum.

Then switch unpooling works as follows. For all $1 \leq i \leq H, 1 \leq j \leq W, 1 \leq d \leq D$, it holds that

$$Y_{(i-1)\delta_h+P_{i,j,d},(j-1)\delta_w+Q_{i,j,d},d} = X_{i,j,d},$$

and for all $1 \leq i' \leq H', 1 \leq j' \leq W', 1 \leq d \leq D$ that did not fall under the above definition,

$$Y_{i',j',d} = 0.$$

In other words, all maximum elements are placed on their original positions in the output of switch unpooling, whereas all non-maximum elements are set to zero. See Figure 11.

There are two minor remarks that have to be made. One is that, there may be ties when taking the maximum. However, as we noted in Section B.6.2, we can ignore these, as they will be rare; we can assume that they are consistently broken, and use the same tie-breaking mechanism in the formulas for $\mathbf{Y}, \mathbf{P}$ and $\mathbf{Q}$. Secondly, when $H_F > \delta_h$ or $W_F > \delta_w$, an element of $\mathbf{Y}$ might have been the maximum of

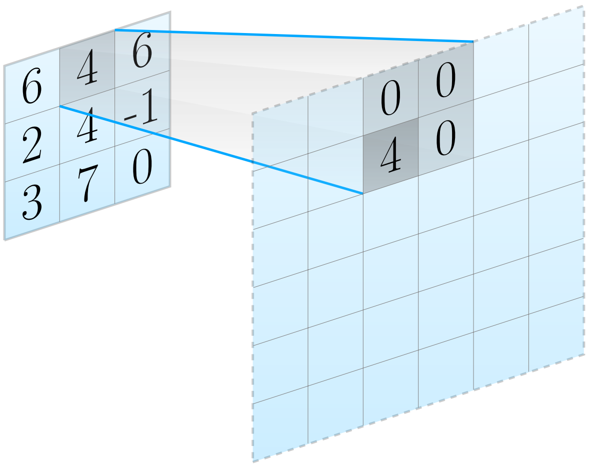


Figure 11: An example of switch unpooling with a 2×2 pooling region and stride 2. No padding is used. The recorded maximum is set as the value on the location of the maximum in the (encoder's) corresponding pooling region, other elements are set to zero. Adapted from Churchman (2015) with permission.

multiple pooling regions during max pooling, causing it to be written to more than once by the above equations. However, this is not a problem, because the same value is written to that element, so the output is still uniquely-defined.

For the derivative, we again remark that, since we only *select* certain elements from $\mathbf{X}$ and place these in $\mathbf{Y}$, and the rest is set to 0, switch unpooling is a linear transformation. Hence, there is a selector matrix $\mathbf{S}(\mathbf{x}) \in \{0,1\}^{H'W'D \times HWD}$, as in Section B.6.2, such that

$$\mathbf{y} = \mathbf{S}(\mathbf{x}) \cdot \mathbf{x}.$$

Then, as we have seen before, we have

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{S}(\mathbf{x}), \frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \mathbf{S}(\mathbf{x}).$$

B.8.3 Blind upsampling

For our input and output dimensions, we use the same set-up as in Section B.8.2.

However, instead of defining the switch matrices, we define the sets

$$P_{i'} = \{i : 1 \leq i \leq H, (i-1)\delta_h + 1 \leq i' \leq (i-1)\delta_h + H_F\},$$

$$Q_{j'} = \{j : 1 \leq j \leq W, (j-1)\delta_w + 1 \leq j' \leq (j-1)\delta_w + W_F\},$$

for all $1 \leq i' \leq H', 1 \leq j' \leq W'$. In other words, $P_{i'} \times Q_{i'}$ is the set of indices (i, j) such that $Y_{i',j',d}$ belongs to the pooling region of $X_{i,j,d}$. Note that $P_{i'}, Q_{j'} \neq \varnothing$ for all allowed i', j', as each input element of max-pooling must have belonged to at least one pooling region.

For blind upsampling, as described in Section 4.2.2, we simply put the recorded maximum of each pooling region back in all of its member elements, averaging when an element belongs to multiple regions. This gives us the formula:

$$Y_{i',j',d} = \frac{1}{\#P_{i'} \cdot \#Q_{j'}} \sum_{i \in P_{i'}} \sum_{j \in Q_{j'}} X_{i,j,d}.$$

This is defined, as $P_{i'}, Q_{j'} \neq \varnothing$, and therefore also $\#P_{i'} \cdot \#Q_{j'} \neq 0$. See Figure 12.

Note that, again, blind upsampling is a linear transformation, as it only selects elements from $\mathbf{X}$ and places them in $\mathbf{Y}$ (multiplied by a constant, but this is no problem for linearity). Thus, there must again exist a selector matrix, $\mathbf{S}(\mathbf{x}) \in [0,1]^{H'W'D \times HWD}$ (since fractions between 0 and 1 are also possible), such that

$$\mathbf{y} = \mathbf{S}(\mathbf{x}) \cdot \mathbf{x}, \text{ and}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{S}(\mathbf{x}), \frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{y}} \mathbf{S}(\mathbf{x}).$$

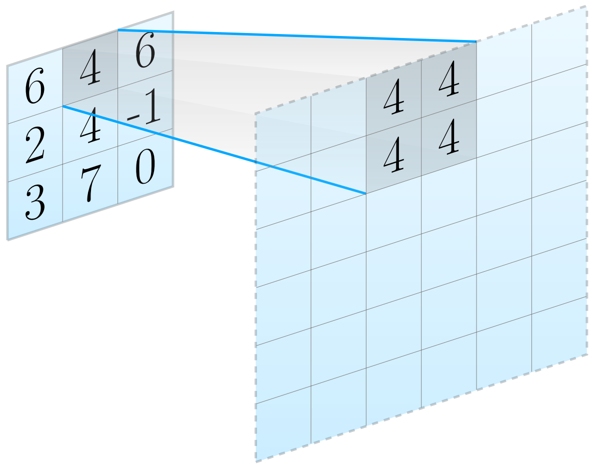


Figure 12: An example of blind upsampling with a 2×2 pooling region and stride 2. No padding is used. The recorded maximum is set as the value for all unpooled elements in the corresponding pooling region. Adapted from Churchman (2015) with permission.

B.9 Adaptive Gradient algorithm (AdaGrad)

AdaGrad (Duchi et al., 2011) is a mathematically-sound learning rate annealing algorithm, using the sum of squares of earlier gradient updates per filter (or bias) element. Updates to often-changed filter elements are decreased in strength, whereas updates to seldom-changed filter elements are (relatively) strengthened. We ignore the biases in this description, but as the filter elements are only used as a vector, they can easily be replaced by the biases for the bias version of the algorithm.

Let $\mathbf{f} \in \mathbb{R}^{H_F W_F D K}$ be the vectorisation of the filter bank to be trained, and define $\mathbf{G}_t \in \mathbb{R}^{H_F W_F D K \times H_F W_F D K}$ a diagonal matrix for each $t \geq 0$, where $\mathbf{G}_0$ is the all-zero matrix.

Normally in gradient descent (see Section B.2), we update these filters as follows:

$$\mathbf{f} \leftarrow \mathbf{f} - \eta \frac{\partial E}{\partial \mathbf{f}}, \text{ or}$$

$$\mathbf{f}^{(t+1)} = \mathbf{f}^{(t)} - \eta \frac{\partial E}{\partial \mathbf{f}^{(t)}}.$$

where E is the error function used (it is irrelevant to this discussion whether E is the element-wise or total error function) and where t indicates the time step. Now, when updating $\mathbf{f}$ in this way, we also update $\mathbf{G}$:

$$\mathbf{G}_{ij}^{(t+1)} = \mathbf{G}_{ij}^{(t)} + \delta_{ij} \left(\frac{\partial E}{\partial \mathbf{f}_i^{(t)}} \right)^2,$$

where δ_{ij} is the Kronecker delta (ensuring that only diagonal elements of $\mathbf{G}$ are updated). In other words, $\mathbf{G}_t$ contains the sum of squares of previous gradients.

Then, with AdaGrad, the filters are updated as (the expression is slightly unwieldy due to the square root of a matrix not being defined properly):

$$\mathbf{f}_i^{(t+1)} = \mathbf{f}_i^{(t)} - \frac{\eta}{\sqrt{\mathbf{G}_{ii}^{(t)}}} \cdot \frac{\partial E}{\partial \mathbf{f}_i^{(t)}}.$$