Learn Latent Representations

Autoencoders and Self-supervised Learning

Jonathon Hare
Vision, Learning and Control
University of Southampton
One of the common features of many of the deep learning models we have looked at to this point is that they often try to reduce the dimensionality of the input data in order to capture some kind of underlying information.

A few lectures ago this was particularly evident when when we looked at embedding models like word2vec which explicitly try to capture relationships in the data in a low dimensional ‘latent’ space.

Self-supervised Learning

I now call it "self-supervised learning", because "unsupervised" is both a loaded and confusing term.

In self-supervised learning, the system learns to predict part of its input from other parts of it input. In other words a portion of the input is used as a supervisory signal to a predictor fed with the remaining portion of the input.
Self-supervised Learning

- The word2vec models are examples of *self-supervised learning*
  - CBOW learns to predict the focus word from the context words
  - Skip-gram learns to predict the context words from the focus word
- Let’s now consider a different type self-supervised task where we want to learn a model that learns to **copy** its input to its output.

Autoencoders

- An **autoencoder** is a network that is trained to copy its input to its output
  - Internally there is some hidden vector $z$ that describes a **code** that represents the input.
  - Conceptually the autoencoder consists of two parts:
    - The encoder $z = f(x)$
    - The decoder $r = g(z)$
  - and has loss that tries to minimise the reconstruction error (typically SSE/MSE: $\|x - r\|_2^2$)

![Structure of an Autoencoder](image)

14.1 Undercomplete Autoencoders

Copying the input to the output may sound useless, but we are typically not interested in the output of the decoder. Instead, we hope that training the autoencoder to perform the input copying task will result in $h$ taking on useful properties.

One way to obtain useful features from the autoencoder is to constrain $h$ to have smaller dimension than $x$. An autoencoder whose code dimension is less than the input dimension is called **undercomplete**. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function

$$L(x, g(f(x)))$$

where $L$ is a loss function penalizing $g(f(x))$ for being dissimilar from $x$, such as the mean squared error. When the decoder is linear and $L$ is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA. In this case, an autoencoder trained to perform the copying task has learned the principal subspace of the training data as a side-effect.

Autoencoders with nonlinear encoder functions $f$ and nonlinear decoder functions $g$ can thus learn a more powerful nonlinear generalization of PCA. Unfortunately,
Autoencoder constraints

- Clearly a linear autoencoder with a sufficient number of weights (e.g. if the dimension of $z$ was greater than or equal to that of $x$) could learn set $g(f(x)) = x$ everywhere, but this obviously wouldn’t be useful!
- In practice we apply restrictions\(^1\) to stop this happening.
- The objective is to use these restrictions to force the autoencoder to learn useful properties of the data.

\(^1\)these are ‘inductive biases’ and the ‘innate priors’ of the model and learning algorithm

Undercomplete Autoencoders

- Undercomplete autoencoders have $\text{dim}(z) << \text{dim}(x)$.
- This forces the encoder to learn a *compressed representation* of the input.
- The representation will capture the most *salient* features of the input data.
Consider the single-hidden layer linear autoencoder network given by:

\[ h = W_e x + b_e \]
\[ r = W_d z + b_d \]

where \( x \in \mathbb{R}^n \), \( z \in \mathbb{R}^m \) and \( m < n \).

With the MSE loss, this autoencoder will learn to span the same subspace as PCA for a given set of training data.

Note that the autoencoder weights are not however constrained to be orthogonal (like they would be in PCA).

A linear autoencoder with a single hidden layer learns to map into the same subspace as PCA.

Clearly, a deeper, linear autoencoder would also do the same thing.

What happens if you introduce non-linearity?

- Interestingly, a single hidden layer network with non-linear activations on the encoder (keeping the decoder linear) and MSE loss also just learns to span the PCA subspace\(^2\!\)
- But, if you add more hidden layers with non-linear activations (to either the encoder, decoder or both) you can effectively perform a powerful non-linear generalisation of PCA

Deep Autoencoders - caveat

- There is a slight catch: if you give the deep autoencoder network too much capacity (too many weights) it will learn to perform the copying task without extracting anything useful about the data.
- Of course this means that will likely not generalise to unseen data.
- Extreme example:
  - Consider a powerful encoder that maps \(x\) to \(z \in \mathbb{R}^1\)
  - Each training example \(x^{(i)}\) could e.g. be mapped to \(i\).
  - The decoder just needs to memorise the training examples so that it can map back from \(i\).
Thus far, we only considered autoencoders with vector inputs/outputs and fully-connected layers.

There is nothing stopping us using any other kinds of layers though...

If we’re working with image data, where we know that much of the structure is ‘local’, then using convolutions in both the decoder makes sense.
Regularised Autoencoders

- Rather than (necessarily) forcing the hidden vector to have a lower dimensionality than the input, we could instead utilise some form of regularisation to force the network to learn interesting representations...
- Many ways to do this; let’s look at two of them:
  - Denoising Autoencoders
  - Sparse Autoencoders

Denoising Autoencoders

- Denoising autoencoders take a partially corrupted input and train to recover the original undistorted input.
- To train an autoencoder to denoise data, it is necessary to perform a preliminary stochastic mapping to corrupt the data \( x \rightarrow \tilde{x} \).
  - E.g. by adding Gaussian noise.
- The loss is computed between the reconstruction (computed from the noisy input) against the original noise-free data.
In a sparse autoencoder, there can be more hidden units than inputs, but only a small number of the hidden units are allowed to be active at the same time.

This is simply achieved with a regularised loss function:
\[ \ell = \ell_{\text{mse}} + \Omega(z) \]

A popular choice that you’ve seen before would be to use an l1 penalty \( \Omega(z) = \lambda \sum |h_i| \)

this of course does have a slight problem... what is the derivative of \( y = |x| \) with respect to \( x \) at \( x = 0 \)?

Autoencoder Applications

Any basic AE (or its variant) can be used to learn a compact representation of data.

- You can learn useful features from data without the need for labelled data.
- Denoising can help generalise over the test set since the data is distorted by adding noise.

Pretraining networks
- Anomaly Detection
- Machine translation
- Semantic segmentation
When we trained supervised classification networks we usually assume that the network produces an output distribution $p(y|x)$ and try to minimise the negative log-likelihood $-\log(p(y|x))$.

In a decoder of an autoencoder we could do the same thing and have the decoder learn $p_{\text{decoder}}(x|z)$ by minimising $-\log(p(x|z))$.

- A linear output layer could parameterise the mean of a Gaussian distribution for real-valued $x$; in this case the negative log likelihood yields the MSE criterion.
- Binary $x$ would correspond to a Bernoulli distribution parameterised by sigmoid outputs.
- Discrete (or categorical) $x$ would correspond to a softmax distribution.

What about the encoder - could we make that output $p(z|x)$?