COMS 4030A

Adaptive Computation and Machine Learning

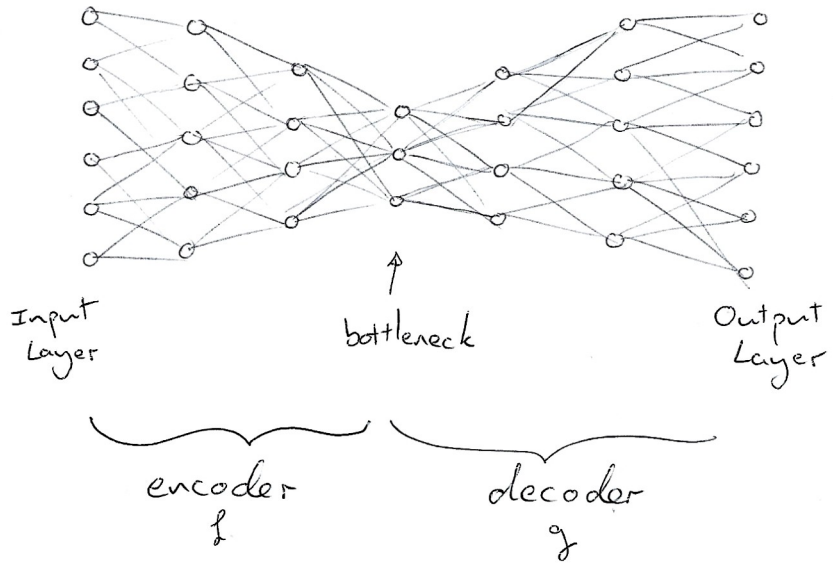
9. Autoencoders

Autoencoders provide a machine learning approach to data compression and dimensionality reduction, and can also be used to reduce noise in data and to generate new data samples.

The main goal for autoencoders is to represent a complex datapoint with a new datapoint in lower dimensional space, called the **code**. The code is considered a compressed version of the original datapoint; it should contain sufficient information about the original datapoint but using fewer dimensions. The space in which the code resides is called the **latent space**. It should be possible to reconstruct the original datapoint from the code with little to no reconstruction or compression loss.

An autoencoder has to construct a function that can transform a particular instance of data into a meaningful code. We can think of this as a remapping of the original data using fewer dimensions.

The following diagram shows the structure of a neural network autoencoder:



The above autoencoder has two parts, the encoder and the decoder.

The Encoder: The encoder's role is to compress the data into a code. In neural networks, this can be implemented by connecting a series of layers, each one reducing the number of dimensions that are present in the data.

The layer of the neural network that has the fewest dimensions, generally in the middle of all the layers, is called the **bottleneck**. Given an input to the network, the vector of activation values at the bottleneck layer is the **encoded** version of the input, or just the **code**.

The number of nodes in the bottleneck is called the **code size**, which is a hyperparameter. In general, the objective of the autoencoder is to produce codes that are smaller than the input but retain enough information or enough features of the original so that it can be reconstructed from the code. An autoencoder such as the above, in which the code size is smaller than the input size, is said to be **undercomplete**. (One can also consider **overcomplete** autoencoders in which the code size is greater than the input size.)

The Decoder: The decoder component of the network acts as an interpreter for the code. We can think of this component as a reconstruction, value extraction, interpretation, or decompression tool. The output layer is of the same size as the input layer to the network since the objective is to be able to reconstruct the original input from the code.

9.1. **Training an Autoencoder.**

Let f denote the function that transforms an input into its code, and let g denote the function that transforms a code into an output value. Then f is the **encoder** and g is the **decoder**.

An autoencoder neural network is trained using a dataset consisting of datapoints without labels. The training of the network is done using supervised training algorithms in which every datapoint in the dataset is used as its own target.
So, if some $\boldsymbol{x}$ is input into the network, then the output of the network should be a copy of $\boldsymbol{x}$. That is, we want $g(f(\boldsymbol{x})) = \boldsymbol{x}$.

The network is trained using backpropagation with a loss function $L(\boldsymbol{x}, g(f(\boldsymbol{x})))$, where L is usually taken to be the sum-of-squares loss function.
The loss between the $\boldsymbol{x}$ and $g(f(\boldsymbol{x}))$ is called the **reconstruction loss**.

The usual method of training using train/validation/test sets applies.

9.2. **Using the Code.**

Once an autoencoder has been trained on a dataset, every datapoint in the dataset can be fed into the trained autoencoder and the code for the datapoint obtained. That is, just the encoder part of the autoencoder is used, and a new dataset is obtained containing all the codes. This new dataset of codes is intended to be a lower dimension representation of the original dataset; i.e., dimensionality reduction, or data compression, has been applied to the original dataset. The space in which the code dataset resides is the **latent space**.

The dataset of codes can be used in place of the original dataset to train models using either supervised learning if the original datapoints had targets, or unsupervised learning, or any other type of learning. New datapoints that were not part of the original dataset can also be used since the encoder can first be applied to the datapoint, and the code then input into whatever model was trained on the codes.

As an example of how an autoencoder is used a compression tool, consider an autoencoder trained on a dataset containing high resolution images. The codes produced by the encoder part of the autoencoder can be used as a compressed form of the original data. The decoder is then required whenever a particular code is selected for reconstruction. It is expected that the quality of the reconstructed image is a bit lower than the original. The reconstruction loss in this case is loss in quality between the output and input images.

Note that the autoencoder networks can use convolutional layers if required, although some reverse convolution is then required to reconstruct an image of the original size.

There are many other ways to compress data, or reduce dimensionality. **Principal component analysis** (PCA) is a commonly used method; it is a purely statistical method that can help identify the key features that account for the variability in the data and use these to represent the information using fewer bits. However, PCA can only offer an encoding with linearly uncorrelated features.

The number of nodes in the bottleneck layer is chosen initially by the user. The type of codes produced can be influenced by adding a penalty to the loss function, such as

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h}),$$

where $\boldsymbol{h}$ is the output of the bottleneck layer. Thus, output values in the bottleneck layer directly affect the loss function at the output layer.

Apart from data compression, or dimensionality reduction, there are a number of other uses of autoencoders, as well as various types of autoencoders. We discuss a few of them next.

9.3. **Sparse Autoencoders.**

Sparse autoencoders are often used when the intention is to use the code dataset as a set of features for another task. In some cases, it is desirable that the codes be 'sparse', that is, they should have only a few non-zero values.

Sparseness in the codes can be obtained by adding the following sparsity penalty to the loss function:

$$\Omega(\boldsymbol{h}) = \lambda \sum_i |h_i|$$

which forces the values in $\boldsymbol{h}$ to stay small.

9.4. **Denoising Autoencoders.**

Denoising autoencoders, or DAEs, are types of autoencoders used to encode noisy data efficiently. For inputs that have been corrupted by some random noise, the autoencoder should be able to reconstruct a denoised version of the input. This means that the output of the autoencoder is expected to be different to the input.

Since an autoencoder seeks to reconstruct its input, in order to train a denoising autoencoder, it is necessary to introduce noise in the input. To do this, some random noise is added to each input during training. The resulting input is said to be **corrupted**. Different random noise is added every time an input is used for training.

If $\boldsymbol{x}$ is an input and we denote by $\tilde{\boldsymbol{x}}$ the corrupted version of $\boldsymbol{x}$, then the goal of the denoising autoencoder is to reconstruct the original $\boldsymbol{x}$, i.e., to 'denoise' $\tilde{\boldsymbol{x}}$.
Thus, the loss function used in backpropagation is $L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}})))$, where L is usually sum-of-squares loss.

9.5. **Variational autoencoders.**

Variational autoencoders, or VAEs, encode data as distributions as opposed to single points in space. This is a way of having a more regular latent space that can be used to generate new data with. Hence, VAEs are referred to as generative models.

The process to achieve this latent space distribution makes the training process a little different. First, instead of mapping an instance as a point in space, it is mapped as the center of a normal distribution. Next, a point from that distribution is selected to decode and compute the reconstruction error that is backpropagated through the network.

9.6. **Learning Manifolds with Autoencoders.**

In many cases, a dataset of points residing in $\mathbb{R}^n$ can be described by a function in lower dimensional space. For example, if S is a set of points in $\mathbb{R}^2$ that all lie on a straight line, then S can be considered as a subset of $\mathbb{R}^1$ if a suitable coordinate system is used. In this case, S is a 1-dimensional manifold lying in 2-dimensional space. A curved line can also be considered a 1-dimensional manifold. In addition, such lines can also reside in $\mathbb{R}^3$ or higher dimensional space. The points in S do not need to lie precisely on the line, there can be an allowance for some error.

For another example, consider a set of points in $\mathbb{R}^3$ that lie on a plane. This is a 2-dimensional manifold in 3-dimensional space. With a suitable coordinate system the set can be described in 2 dimensions. Even if the plane is folded in 3-dimensional space, it is still 2-dimensional.

Datasets that lie on a manifold of dimension lower than the space in which it resides, are obtained when the dataset is created by a process that has a lower degree of freedom than the space it resides in.

The method of principal component analysis, or PCA, is very good at finding lower dimensional descriptions of data that is linear (such as straight lines, planes and hyperplanes). PCA uses eigenvectors to form a suitable coordinate system. It does not need any training to find the eigenvalues – linear algebra gives the solutions.

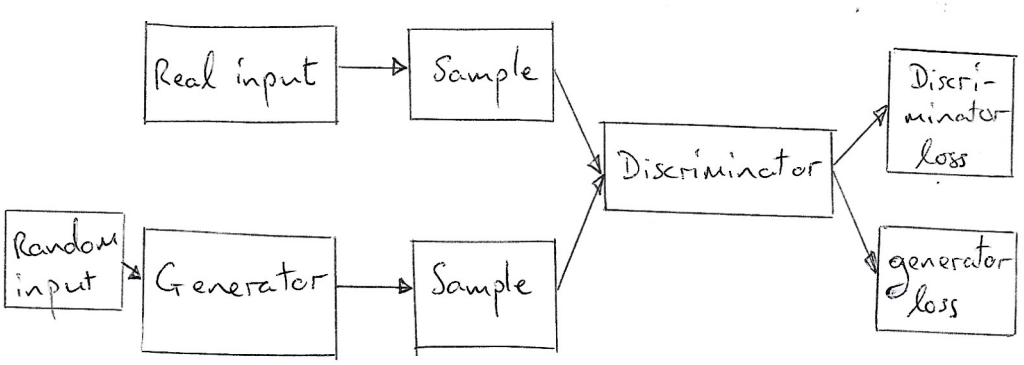
For manifolds that are not linear, such as curved lines or folded planes, or folded hyperplanes, neural network autoencoders can be used to find suitable desrciptions of the manifolds. Essentially, the codes obtained from an autoencoder describe the dataset in a lower dimension.

10. Generative Adversarial Networks

Generative adversarial networks, or **GANs**, are generative models: they create new data instances that resemble the training data. For example, GANs can create images that look like photographs of human faces, even though the faces don't belong to any real person.

GANs achieve this level of realism by pairing a **generator**, which learns to produce the target output, with a **discriminator**, which learns to distinguish true data from the output of the generator. The generator tries to fool the discriminator, and the discriminator tries to keep from being fooled.

The following diagram shows the structure of a GAN:



A GAN has two parts:

The Discriminator: The discriminator is a neural network that take inputs of a certain type and outputs a decision of 'real ' or 'fake'. Typically, the neural network has a single output node that uses a sigmoid activation function so that the output is value between 0 and 1. An output close to 1 indicates that the input is a real sample, whereas an output close 0 indicates that the input is a fake sample. A number between 0 and 1 is the estimated probability that the sample input is real.

The Generator: The generator is also a neural network that learns to generate plausible data, i.e., samples that would be predicted as 'real' by the discriminator.
To generate new samples, some sort of randomly generated **noise** is used as input.
The output from the generator is a sample of the same type as the real samples and this output feeds directly into the discriminator.

By introducing noise, the generator can be used to produce a wide variety of data, sampling from different places in the target distribution. The noise can be sampled from some kind of uniform distribution. The space from which the noise is sampled is usually of smaller dimension than the dimensionality of the output space.

The discriminator has two sources of input samples – those from the dataset which are 'real', and those from the generator which are 'fake'. The goal of the discriminator is to learn to distinguish between real and fake samples.
Thus, during training the real samples are given a target of 1 and the generated (fake) samples are given a target of 0.

10.1. **Training a GAN.**

Training of both the discriminator neural network and the generator neural network is done through backpropagation, as discussed below. When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake. As training progresses, the generator gets closer to producing output that can fool the discriminator. Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

The discriminator and the generator are trained separately.

The first loss function that was used to train GANs was **minimax loss**, which is derived from cross-entropy between the real and generated distributions:

$$\mathbb{E}_{\boldsymbol{x}}[\log(D(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z}}[\log(1 - D(G(\boldsymbol{z})))],$$

where
$D(\boldsymbol{x})$ is the discriminator's output for input $\boldsymbol{x}$, i.e., its estimate of the probability that a real data instance $\boldsymbol{x}$ is real.
$\mathbb{E}_{\boldsymbol{x}}$ is the expected value over all real data instances.
$G(\boldsymbol{z})$ is the generator's output when given random noise input $\boldsymbol{z}$.
$D(G(\boldsymbol{z}))$ is the discriminator's output for input $G(\boldsymbol{z})$, i.e., its estimate of the probability that a fake instance is real.
$\mathbb{E}_{\boldsymbol{z}}$ is the expected value over all random noise inputs to the generator (in effect, the expected value over all generated fake instances $G(\boldsymbol{z})$).

Training the Discriminator

The discriminator trains on inputs from the real dataset and inputs from the generator. The discriminator is penalised for misclassifying a real instance as fake or a fake instance as real.

For a single datapoint, the discriminator loss function is:

$$L_D = \begin{cases} \log(D(\boldsymbol{x})) & \text{for real input } \boldsymbol{x}, \\ \log(1 - D(G(\boldsymbol{z}))) & \text{for fake input generated from } \boldsymbol{z}. \end{cases}$$

For a real input $\boldsymbol{x}$, the output from the discriminator is $D(\boldsymbol{x})$, where $0 \leq D(\boldsymbol{x}) \leq 1$.
The effect of the log is that $-\infty < \log(D(\boldsymbol{x})) \leq 0$, so the loss function is alway negative.
For real inputs $\boldsymbol{x}$ the target is 1 and $L_D = \log(D(\boldsymbol{x}))$ has maximum value of 0 when $D(\boldsymbol{x}) = 1$; thus, we seek to **maximize** L_D for real inputs.

For fake inputs $G(\boldsymbol{z})$ from the generator, the ouput from the disciminator is $D(G(\boldsymbol{z}))$, where $0 \leq D(G(\boldsymbol{z})) \leq 1$. Then, $0 \leq 1 - D(G(\boldsymbol{z})) \leq 1$ hence $-\infty < \log(1 - D(G(\boldsymbol{z}))) \leq 0$.
For fake inputs $G(\boldsymbol{z})$ the target is 0 and $L_D = \log(1 - D(G(\boldsymbol{z})))$ has maximum value when $D(G(\boldsymbol{z})) = 0$; thus, again, we seek to **maximize** L_D for fake inputs.

Thus, the goal for the discriminator is to maximize the function L_D.
Therefore, when the discriminator updates its weights through backpropagation of the discriminator loss, **gradient ascent** is used in place of gradient descent, i.e., the update rule for weights is $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta(\frac{\partial L_D}{\partial \boldsymbol{w}}|_{\boldsymbol{xtW}})$.

The generator is kept constant during the discriminator training.

Training the Generator

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

For a single datapoint, the generator loss function is:

$$L_G = \log(1 - D(G(\boldsymbol{z}))).$$

For an input $\boldsymbol{z}$ into the generator, which outputs $G(\boldsymbol{z})$, which is then fed into the discriminator.
In this case, the target used is 1 since the generator wants to fool the discriminator.
Thus, the generator seeks to **minimize** the above loss. The usual gradient descent is used, so the update rule for weights is $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta(\frac{\partial L_G}{\partial \boldsymbol{w}}|_{\boldsymbol{xtW}})$.

The generator training process is as follows:

Sample random noise $\boldsymbol{z}$.
Produce generator output $G(\boldsymbol{z})$ using the sampled random noise as input.
Feed the generator output into the discriminator to get 'real' or 'fake' classification $D(G(\boldsymbol{z}))$.
Backpropagate the loss L_G through both the discriminator and generator to obtain gradients.
Use the gradients to update the generator weights, but **do not** update the weights in the discriminator.

GAN Training

Before GAN training begins, the discriminator is trained to recognise samples from its target dataset. This requires a larger dataset that also contains negative samples, but these negative samples are not used in training the GAN.

Because a GAN contains two separately trained networks, its training algorithm must address the complication of juggling two different kinds of training (generator and discriminator), as well as the issue of convergence, which is hard to identify.

GAN training proceeds in alternating periods:

The discriminator trains for one or more epochs.
The generator trains for one or more epochs.
Repeat above steps to continue to train the generator and discriminator networks.

As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. If the generator succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator flips a coin to make its prediction.

This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse. For a GAN, convergence is often a fleeting, rather than stable, state.

11. Semi-supervised Learning

Recall that supervised learning is a method of training a machine learning model using a labelled dataset, and that unsupervised learning is used for unlabelled data, where a model tries to mine hidden patterns in the data by grouping datapoints into clusters based on similarities.
Semi-supervised learning bridges supervised learning and unsupervised learning techniques and can be applied for a variety of problems from classification and regression to clustering.

Semi-supervised learning, or **SSL**, is a machine learning technique that is applicable to datasets that have a mixture of datapoints with targets, called **labelled data**, and datapoints without targets, called **unlabelled data**. A predictive model is trained that makes use of both labelled and unlabelled datapoints.

Semi-supervised learning is usually applied in cases where the amount of labelled data is far less than the amount of unlabelled data. In many applications, unlabelled data is often abundant and easy to get, while obtaining suitable labels for data is expensive, so one has to make do with a small set of labelled data. For this reason, semi-supervised learning finds many applications.

The key idea in SSL is that the datapoints that have targets can be used to create targets for the unlabelled datapoints. Targets created in this way are called **pseudo-labels**. In order for such an approach to be successful, the pseudo-labels must be reasonably close to what could be expected from the data. Thus, the following assumptions about the dataset are usually made when applying SSL:

Continuity assumption: It is assumed that datapoints near each other tend to share the same target label. This assumption is also used in supervised learning where datasets are separated by decision boundaries.

Cluster assumptions: It is assumed that data can be naturally clustered and that datapoints in the same cluster share the target label.

Manifold assumptions: It is assumed that the data lies on a manifold of fewer dimensions than input space.

SSL fits well for clustering and anomaly detection purposes, assuming the data is suitable based on the requirements of the SSL approach used. But, SSL does not apply well to all tasks. If

the portion of labelled data isn't representative of the entire distribution, the approach may fall short.

11.1. **Pseudo-labelling.**

The method of **pseudo-labelling** of the unlabelled data, based on the labelled data, is as follows.

First, use the dataset with the labelled data to train a model using usual supervised algorithms.
Next, use the trained model to predict target labels for some of the unlabelled data.
(These are the pseudo-labels.)
Add these datapoints, together with their pseudo-labels, to the labelled dataset.
Repeat the above process with the now larger set of labelled data until all data have labels.

The above approach can make use of pretty much any supervised algorithm with some modifications if needed.

11.2. **Self-Training.**

Self-training is a variant of pseudo-labelling that only accepts predictions that have a high confidence. The process is as follows.

First, use the dataset with the labelled data to train a model using usual supervised algorithms.
Next, use the trained model to predict target labels for some of the unlabelled data.
Retain only the most confident predictions made with the model.
For example, retain only predictions made with confidence of over 80%.
For the pseudo-labels that exceed this confidence level, add the datapoints, together with their pseudo-labels, to the labelled dataset.
Repeat the above process with the now larger set of labelled data until all (or a sufficient number of) data have labels.

The process can go through several iterations (usually about 10) with more pseudo-labels being added every time. Provided the data is suitable for the process, the performance of the model should keep increasing at each iteration.