Explanatory Notes for 6.3900

Shaunticlair Ruiz

Fall 2024

Contents

CHAPTER 13

Autoencoders

Through neural networks, we've **upgraded** the set of models we can use to do classification and regression tasks.

- Neural networks become more complex and **expressive** as we add more **layers**.

We'll spend the next few chapters exploring NNs: creating new variants (CNNs, RNNs), for example.

In this chapter, we'll investigate a different kind of application: **autoencoders.**

13.0.1 Unsupervised Learning

In chapter 6, we discussed an **unsupervised** learning problem: clustering.

- Classification tells us which classes to use. By contrast, clustering **doesn't** "know" the classes we're looking for.
  - Instead, we discover new classes ("clusters"), using the k-means algorithm.
- This lack of guidance is what makes clustering **unsupervised**.

Clustering was used as a form of **data analysis**: we were able to learn more about the **structure** of our data, by finding what sorts of "groups" existed.

We'll use autoencoders for a different task, that follows the same theme: learning more about the data, by attempting to look for an **unknown solution** to a simple problem.

13.0.2 Autoencoders: Compression

This time, our problem is not cluster-finding, but instead, **compression**. We want to take our input data, and find a more **space-efficient** way to represent it.

Typically, this means reducing the number of **dimensions**/variables.

- **Example:** turning a 10-dim data point into a 4-dim data point.

Why would we do this? Because of what we gain from this task:

- A good compression algorithm should be able to be **decompressed**, while keeping the result mostly similar.
  - That means that our compression must preserve **essential** information, so it's possible to retrieve later.
- By observing what's the algorithm decides to preserve and discard, can teach us what matters, and what doesn't.

> **Concept 1**
> Learning a **compression algorithm** for your dataset creates a **simplified** representation, that still keeps the most important, distinct information.
>
> Based on the information we find in that representation, we can figure out what components were "**important**".

Our compression/decompression system needs to do two things:

- **Reproduce** the original data well
- Do so in a way that lets us **distinguish** one data point from another

Based on this, a trained autoencoder can provide us some unexpected insights.

We can better see this with an example.

Example: Consider a database of human faces.

- It's more space-efficient if you can just re-use a "**template**" for a face, and just modify it.
  - So, your model might memorize what a face "generally" looks like.
  - That way, it doesn't have to waste space in the compression for data that appears frequently.

- Then, your compressed data only needs to store what's special, or different, about the face it represents.
  - So, you learn what separates different faces from each other, based on the info in the compressed model.

13.0.3 Training

This representation might even be easier for a new model to **train** with:

- We've omitted some unnecessary information that can **distract** our model.
- With a simpler input, it's faster to train, and compute answers.

The model can overfit to **noise** in these extra variables.

Concept 2

Just like how **clustering** is used to improve downstream tasks, **compression** can be, too.

Compressing your input can improve learning, so it takes less data to train.

Clarification 3

Not all compression is created equal!

Auto-encoding compresses in a way that contains **relevant information**.

However, in the Feature representation chapter, we discussed **binary code**: representing a number in **binary**, because it requires fewer features.

- This kind of compression doesn't emphasize what's "important" about the feature representation.

Binary compression is **worse**, not better! Instead of isolating useful information, it **obscures** it, forcing the model to learn binary.

13.1 Autoencoder Structure

13.1.1 Visualization

Our encoder is a **compression** algorithm, which takes an input $x^{(i)}$, and returns a new "transformed" input $a^{(i)}$, with a lower dimensionality.

Note: you don't have to take the vectors in equation **8.1** literally.

You're not **required** to have Dimension$(a) > 2$, as in **8.1**.

$$\overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix}}^{k<d} \longrightarrow \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix} \tag{13.1}$$

In other words, we're going from $x \in \mathbb{R}^d$ to $a \in \mathbb{R}^k$.

Example: Take the classic problem of the MNIST dataset: identifying the identity of a digit based on a 28 x 28 grid of pixels.

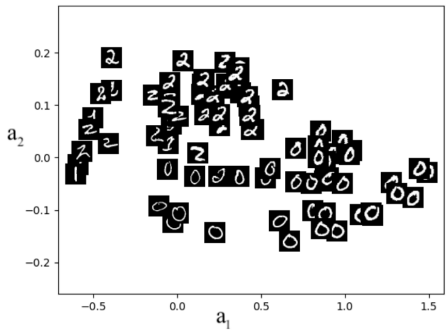
Here's an example of one data point: this represents the digit 9.

- That means our input data has 784 dimensions: $x \in \mathbb{R}^{784}$.
- Below, we've used an encoder to compress it down to 2 dimensions: $a \in \mathbb{R}^2$.

Each pixel is actually **restricted** to $[0, 255]$, but this statement is still technically true.

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix} \longrightarrow \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \tag{13.2}$$

- The x-axis and y-axis indicate the two dimensions of our output, a.



Notice that digits with the same identity are near each other: our compressed representation seems to be storing some useful data about digits!

Somehow, we've created a **simplified** representation that, despite having only 2 variables, has a lot of useful information!

- With only a couple labelled data points, we could guess the digits of most of these pictures, based on how close they are.

Clarification 4

This property, of "similar" data points, being **close** in the latent space, is **not** guaranteed, for any **compressed representation** you create.

However, it's a property we *hope* to find, in a useful one.

But whether or not our data "organizes" nicely like this, we've still demonstrated another benefit of compression:

- It allows us to transform our data into a lower-dimensional, more **digestible** format.
- So, we can create better visualizations.

Concept 5

One application of **compression** is using it for **visualization**.

A lower-dimensional version of a dataset is usually easier to diagram in a way humans can **interpret** directly.

- Because this representation tends to be more **dense** with information, it's often easier to draw useful conclusions.

It can also let us make simple predictions, or find patterns, since there are **fewer** variables to pay attention to.

This compressed version of data is called a "latent representation".

Definition 6

The **latent representation** of our data is the **compressed** version, containing as much useful information as possible, with fewer variables.

- We call it **latent** because original data x is in a "hidden" form, but we can retrieve it by **decompressing**.

The **latent space** represents all of the possible latent representations.

- This "space" follows the tradition of "input/feature spaces": sets with structure. In this case, the **distance** between our data gives us the **structure**.

Generally, a good latent space preserves **information**: the reconstructed input still **communicates** what we were interested in, from the original input.

13.1.2 Anatomy of an Autoencoder

Our autoencoder's purpose is **compression**, but we need to make sure that this compression preserves the information that we want.

Definition 7

An **autoencoder** comes in two parts:

- An **encoder**, which **compresses** our (d-dim) input into the (k-dim) latent representation.
- A **decoder**, which **de-compresses** our latent representation, to try to re-create the input.
  - This is used to make sure that our representation contains the information we need, to accurately re-construct the input.

The goals are:

- To create a **smaller** latent representation, with dimensions $k < d$
- To make sure that this latent representation can **accurately** re-construct our input

- The encoder and decoder can be any kind of function, but we will use **neural networks**.

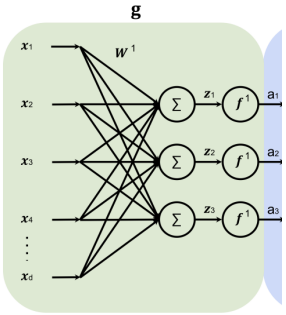
This is where neural networks shine: with more powerful model class, we can do more complex math to create our "encoding".

13.1.3 One layer encoder/one layer decoder

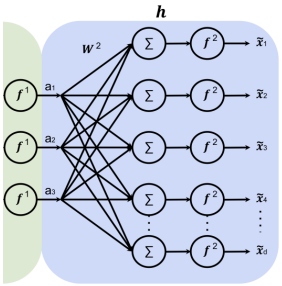
For a simple demonstration, we'll use a version with a one-layer encoder and a one-layer decoder.

We'll take our (d-dim) input, and compress it into a (3-dim) latent representation.

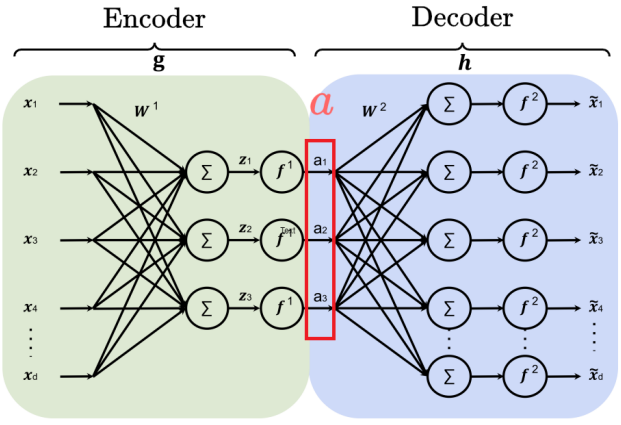
- Encoder: input $x \in \mathbb{R}^d$, output (compressed) $a \in \mathbb{R}^3$.



- Decoder: input (compressed) $a \in \mathbb{R}^3$, output (decompressed) $\tilde{x} \in \mathbb{R}^d$.



Taken together, we get our autoencoder:



Note that, in addition to W^1 and W^2, we have a set of offsets that are not shown: W_0^1 and W_0^2.

Let's run through our network:

- The **input** x goes through the encoder network. This compresses into the **latent representation** a.
  - W^1 has shape $(d \times k)$, or equivalently, $W^1 \in \mathbb{R}^{d \times k}$
  - W_0^1 has shape $(k \times 1)$, or equivalently, $W_0^1 \in \mathbb{R}^k$

$$z^1 = (W^1)^T x + W_0^1 \qquad a = f(z^1) \tag{13.3}$$

- The **latent representation** goes through the decoder network. This de-compresses it into the **re-constructed input**.
  - W^2 has shape $(k \times d)$, or equivalently, $W^2 \in \mathbb{R}^{k \times d}$
  - W_0^1 has shape $(d \times 1)$, or equivalently, $W_0^2 \in \mathbb{R}^d$

$$z^2 = (W^2)^T a + W_0^2 \qquad \tilde{x} = f(z^2) \tag{13.4}$$

The red layer in the center, is the "**latent representation**": the latent representation is **not** the output.

Our autoencoder can have more layers than we do here: this was just an example.

13.1.4 Autoencoders in general

Let's focus on that point about the "red layer" in the center:

Clarification 8

When we train an autoencoder, our goal is to create a **latent representation**.

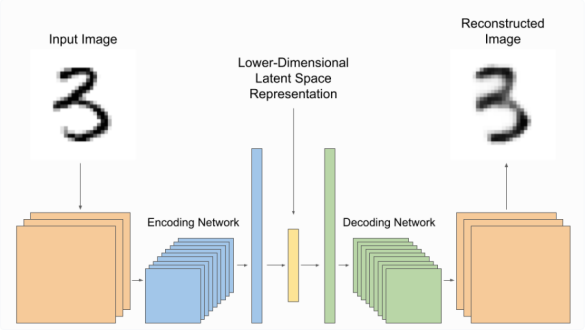
Deceptively, however, that representation is **not the output** of our autoencoder.

- Rather it's in the **middle** of the network: the output of the **encoder**, input to the decoder.

The actual autoencoder output is an **approximate** re-construction of our **input**.

Note that point: we're *approximately* re-constructing our input.

- The **quality** of the re-construction depends on our choice of encoder, and the size of our latent space.
- A bigger latent space (more dimensions) generally allows for a **better** re-construction, but takes up **more** space.



Network Architecture for a Convolutional Ordinary Autoencoder

An example of the original vs reconstructed image from MNIST. Credit to assemblyai.com

The reconstructed 3 above is still very recognizable, but it's clearly been "degraded" somewhat: it's not the same.

Clarification 9

The reconstructed input is usually **not exactly the same** as the input.

$$x \neq \tilde{x}$$

Our re-construction is an **approximation**.

We're essentially running our input through a "**bottleneck**", in a lower dimension.

- We do so, hoping that the right compression size "squeezes out" only unnecessary information.

Concept 10

An autoencoder is, technically, just a normal neural network, where:

- We want the **input** to equal the **output** (re-constructed input)
- We monitor an **intermediate layer** (latent representation), where the layer output size (k) **decreases** below the input size (d)
- If our input and output match well enough, then we use the **output** of that **intermediate layer**.

13.2 Autoencoder Learning

Now that we've defined our autoencoder, it's time to **train** it.

What's our objective? Well, we want to create a lower-dimensional representation that can be used to **re-construct** the original.

- We've handled the lower-dim aspect with the **structure** of the neural network: the last layer of the **encoder** will output k values, giving our **latent representation**.
- So now, we need to show that this representation contains the information we want: it's able to **re-construct** the original.
  - This is the job of the **decoder**.

Where $k < d$, given d is the original input dimension.

That latter point is what we need to address: we need to check the quality of our re-construction, $\tilde{x}$.

Concept 11

We measure the **quality** of our autoencoder by measuring the **similarity** between the original input x, and the re-constructed input $\tilde{x}$.

- If the re-construction is similar to the original input, that means our latent representation successfully **encodes** information about x.

So, we need some kind of **similarity** metric. We'll encode this into our loss function, $\mathcal{L}$. $\mathcal{L}$ tells us how **different** our re-construction is, from the original.

- For **continuous** variables, you might use **squared distance** between x and $\tilde{x}$: loss $\mathcal{L}_{SE}$.

"SE" stands for "squared error". It's different from MSE, "mean squared error", because we're not dividing by d.

$$\mathcal{L}_{SE} = \|x - \tilde{x}\|^2 = \sum_{i=1}^{d} (x_i - \tilde{x}_i)^2 \tag{13.5}$$

Note that often, an input does not only contain one data type: it may include different types of **discrete** data.

So, you may need to use different loss functions for different dimensions of the input.

Now that our problem is fully framed, we can simply **optimize** it, to minimize loss, using our parameters.

Concept 12

Once we've chosen our loss function, we can **optimize** our autoencoder as an ordinary neural network, in order to create our **encoder** for latent representations.

- W_{en} and W_{de} are our encoder and decoder weights, respectively.

Our goal is to optimize these weights, with respect to the **loss**, over our n data points:

$$W_{en}^*, W_{de}^* = \text{argmin}_{W_{en}, W_{de}} \left(\sum_{i=1}^{n} \mathcal{L}(\tilde{x}^{(i)}, x^{(i)}) \right)$$

Or, if we want to be more explicit,

$$W_{en}^*, W_{de}^* = \text{argmin}_{W_{en}, W_{de}} \left(\sum_{i=1}^{n} \mathcal{L}(\text{NN}(x^{(i)}; W_{en}, W_{de}),\ x^{(i)}) \right)$$

- As usual for neural networks, we often optimize using gradient descent via **back-propagation**.

Remember that W^* notation is used to indicate "optimal" parameters.

Example: For our one-layer encoder/decoder above, we would optimize over W^1, W_0^1, W^2, W_0^2.

13.3 Evaluating an Autoencoder

After **training** our autoencoder, we want to be able to **confirm** that it does what we want.

What do we **want**?

- A representation that contains **fewer** dimensions than the input: $k < d$.
  - Remember that k, in this case, is the dimension of our **latent** representation, and d is the dimension of our **input**.
- For this representation to contain useful **information** about our input.
  - This second aspect is (hopefully) addressed by our **loss** function.
  - If our **re-constructions** are really good, then we've managed to preserve our information.

Often, a latent representation can be **much** smaller than the input.

Remember our MNIST example at the start of the chapter, going from 784 dimensions, to 2.

13.3.1 Dimensionality of a

Our loss function handles the latter of these two problems, but the **dimensionality** is based on our **choice** of NN structure.

Well, if we're compressing our data, we want to reduce our number of dimensions, typically. But there's a tradeoff:

Concept 13

The **smaller** our latent representation is, the **less information** we can store in it.

- So, our re-constructions will be generally **worse**.

But a **larger** latent representation uses more **space**/computation, and doesn't **filter** out as much unnecessary, distracting information.

Clarification 14

Because we want to **compress** our data, we require $k < d$.

If we allow $k = d$, then our "latent representation" could be the **exact same** as our original representation.

- In fact, that would be the most **efficient** way to always be correct.
- If $k > d$, then we have extra dimensions, prone to overfitting.

If we **remove** the $k < d$ requirement, we're not forcing our network to do any real work, and our representation **doesn't** have to become more efficient.

In short: if we're not making the dimensions smaller, then we're not really compressing our data!

But what if we wanted to try making the dimensions **larger** on purpose? Does that have applications?

Concept 15

You might wonder if $k > d$ would let us "**unpack**" some of our input data into those extra dimensions.

An encoder with $k > d$ is called an **overcomplete autoencoder**.

Usually, this is **not** desirable:

- Our goal of "recreating the input" doesn't lend it itself well to "unpacking the data": it's usually easier to just **copy** the input.
- Moreover, **overfitting** makes these autoencoders hard to train.

That said, this sort of approach does have applications in de-noising, and learning sparse representations.

Just like in clustering, your exact choice of latent dimensionality k (usually $k < d$) is often subjective or task-based, and requires some trial and error.

We might have different considerations:

- How well does the plot seem to organize our data?
- Are we missing some crucial kind of information?
- Have we encoded information we don't care about on accident?

And more.

13.3.2 Data Analysis

One of the reasons we wanted to design autoencoders was to learn more about our data.

So, a useful encoder might be one that gives us some new or interesting insights.

Concept 16

We can learn more about our (already trained) encoding by **experimenting** with it.

How? By **modifying** the latent representation a, and seeing how that affects the reconstructed version $\tilde{x}$.

- We could start with a known data point, and modify one **dimension** of it, a_j.
- If we increase a_j and get a noticeable change, we can make guesses about what that dimension "**represents**".

Example: Suppose that we embedded the MNIST digits with k dimensions.

- We select one random data point, $x^{(i)}$. This data point happens to be a picture of the number 6.
- We scale up/down one dimension, a_j.
- We might see that the line thickness of the 6 increases. Maybe a_j represents line thickness.

There could be many possible features: how "angular" the number is, how it loops, etc.

Not to say that those are the particular features you *will* find. In fact, sometimes, it's totally unclear what your latent representation is "representing".

A few other examples of how to do our data analysis:

Concept 17

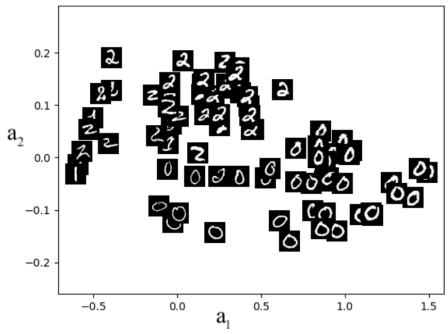
Rather than "experimenting" with individual axes, we could **plot** data points in the **latent space**.

There are two ways to do this:

- Take **real** data points, and plot where they appear in the latent space, compared to how the **input** looks.
- Directly **sample** points from the latent space, and see what their **re-construction** looks like.

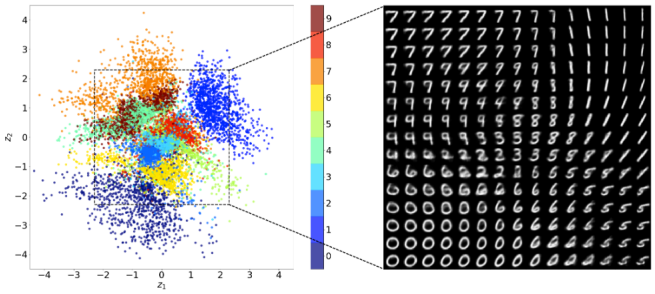
In both cases, we get an idea of how the autoencoder interprets the data it receives.

Example: We started the chapter with an example of the former technique:



Here, we take real data points, and plot them in the latent space.

Example: We could also use the latter: seeing how our re-construction appears, as we modify our latent variables.



The left shows the position of real digits in latent space, while the right shows our re-constructions, moving in a grid across the space. Credit to argmax.ai.

Concept 18

Lastly, we could see what the model thinks the data **generally** looks like, based on the biases/offsets in the network.

- The offset values (W_0^ℓ) are the **same**, regardless of the data point.
- We could set all of the values of a to 0: in a **linear** autoencoder, this would give the **average** of our data points.

If our data clusters around the average, it would be reasonable to expect the average to be **somewhat similar** to all of our data.

- And if it looks similar to each data point, it could somewhat **represent** what it looks like in general.

In a **non-linear** autoencoder, our above approach doesn't give us the **average**, but could be useful regardless.

After analyzing all our dimensions, we can try to figure out what **information** the encoding decided was "important", or at least, necessary for re-construction.

This kind of analysis has a wide array of applications, including natural language processing, disease subtyping, and image processing.

13.3.3 Downstream Tasks

If we're using the encoder for downstream tasks, we can evaluate the autoencoder based on the performance of those downstream tasks.

- We mentioned the same kind of metric when we discussed clustering.

Concept 19

Performance on **downstream tasks** is one way to evaluate the quality of an encoding.

One simple approach, is to use **semi-supervised learning**.

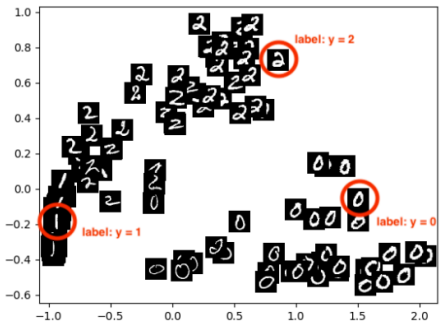
Definition 20

Semi-supervised learning provides labels for only **some** of the data we use to train. The rest of it is unlabelled.

So, the model has to **extrapolate** from that data, to figure out a pattern for the rest of the data.

If we have a **meaningful** latent representation, we could use it to help with this type of problem.

Example: We'll re-use our MNIST example. Suppose we were only given a **few** labelled digits:



Only the three labelled data points are "supervised".

- In this case, we could label many of these digits, just based on what we have.

3 data points is a really small amount of information! This makes it even more impressive.

The fact that this approach works so well, with only 2 dimensions, tells us that our encoding is impressively effective.

13.4 Linear Encoders and Decoders

Even simpler than our one-layer example above, is the **linear** autoencoder.

It turns out that, despite its simplicity, even a linear autoencoder can be useful! In some contexts, it's even **better**:

- A linear autoencoder often has a **closed-form** solution: we don't have to do gradient descent.
- The linear autoencoder tends to create a very simple kind of **interpretability**.

This closed form solution is equivalent to **principle component analysis** (PCA), which you might be familiar with from linear algebra.

- We obtain this solution using a technique called **singular value decomposition** (SVD).

We can draw some parallels to a paradigm we've seen before:

- **PCA** can be thought of as the simplified, linear version of the **autoencoder** problem.
- This is similar to how **linear regression** is the simple, linear version of a **neural network**.

13.4.1 Principle Component Analysis (Optional)

Remark (Optional)

The following section briefly covers the concepts of PCA.

These may provide some intuition for what autoencoders do, in a simple, linear environment.

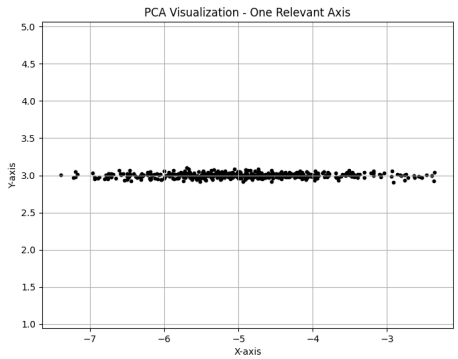
We mentioned that autoencoders need to make their data points clearly **distinct** from one another.

- In other words, it's best to focus on ways that they're **different** from each other.

The easiest way to do that is to focus on sources of **variance** in the data.

13.4.2 Low-variance: less important (Optional)

Let's give a motivating example:



Almost all data is encoded on the x-axis.

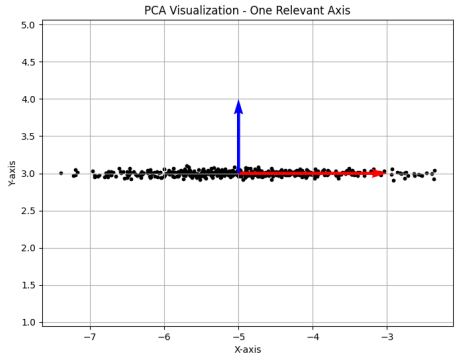
If we look at this dataset, there's a clear **difference** between our two axes: one is very high-variance (x), one is low (y).

If I told you "the y coordinate of this data point is roughly 3", that tells you essentially **nothing**: that's true of all of our data.

Meanwhile, the x coordinate is much more **informative**.

- It seems that **high-variance** dimensions tend to contain **more information** than low-variance dimensions.

So, if we break our data up based on coordinates:



We could remove the y-axis while preserving most of our information! This would be a good target for **omitting** from the latent space.

$$\begin{bmatrix} x \\ y \end{bmatrix} \longrightarrow \begin{bmatrix} x \end{bmatrix} \tag{13.6}$$

Concept 21

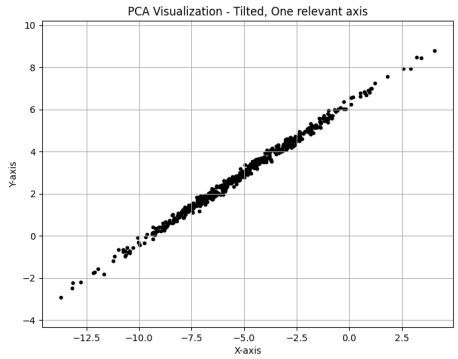
When doing PCA, we tend to focus on axes of **high variance**.

Axes with low variance carry **less information**.

So, if we want to **compress** our data, we can remove those low-variance dimensions.

13.4.3 Different axes (Optional)

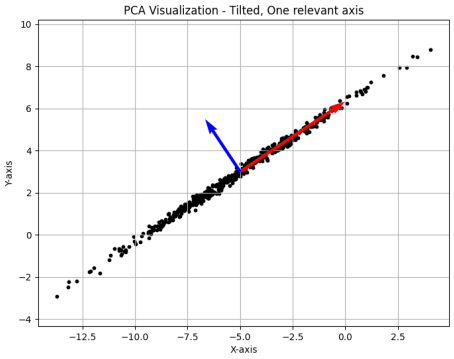
Our data doesn't always (or even usually) line up perfectly with our axes, though.



Almost all data is on a single line.

In this case, it looks very **similar** to our x-axis data. But the problem is, we can't just reduce it to one of our two axes.

The solution? **Change coordinate systems**.



Now, we have a high variance axis, and a low variance axis.

Now, we have the same situation as before! Almost all of our data is on **one axis**: we can omit the other.

Concept 22

Often, the most "**information**-heavy" directions in our data, aren't our default axes.

So, we look for a **new coordinate system**, where most of our variance is contained ("can be explained") by only a few dimensions.

- That way, we can **remove** the other remaining dimensions, which contribute very little to our understanding of the data.
- With fewer dimensions, our data is often more **interpretable**.

This idea, of finding high-variance components, and discarding low-variance components, is the core principle of **principle component analysis**.

- This is equivalent to how our linear autoencoders remove extra dimensions.

Sometimes, it can be a bit difficult to **interpret** these high-variance components: what does it mean to have high variance along a "different axis", conceptually?

But other times, we find that two "separate" variables, are both giving us the **same information**: they would correlate strongly, and thus, would give us the line we see above!

- This is why we can **remove** one dimension: we're using two axes to represent basically the same thing.

Example: If something is sold for a fixed price, then "number of units sold" and "profits from sales" are telling us **the same thing**.

So, we can compress into one dimension with low information loss.

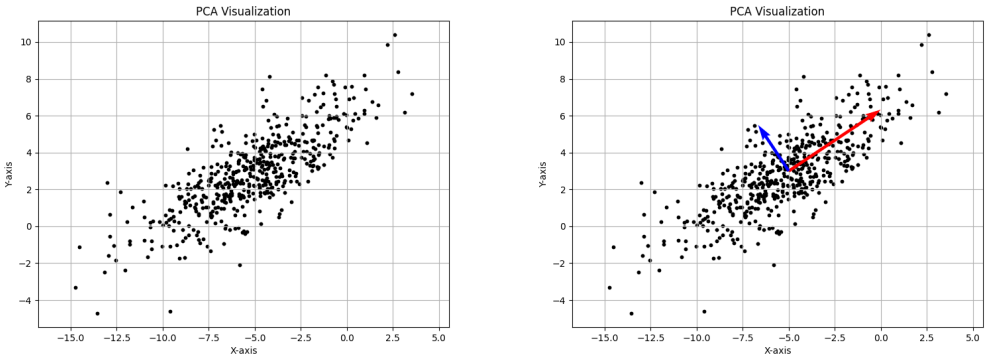
Many examples aren't perfectly correlated like this, but are still pretty similar: for example, sales and advertising spending at a company.

13.4.4 General example (Optional)

How many dimensions we need to capture most of our variance **depends** on the situation.

The goal is often, for visualization, to reduce it to **2 or 3** primary components.

But even if we can't remove axes, PCA can give us a useful way to focus in on the **relationship** between our variables:



Here, we have identified which axis matters "more" to our data, and how much.

13.4.5 Non-linear encoders (Optional)

Of course, things aren't always so simple: we often won't have this nice blob, which we can measure across a few perpendicular axes.

Our real data might take on a different form, that has a coherent "shape", or "structure", but not one that fits our **linear** model.

This is where our **non-linear** autoencoders come in. They follow the same kind of principles as PCA:

- We want to distinguish patterns and curves that contain **variance**, and allow us to **distinguish** data points.
- Those which contain more information, we **keep**. The rest, we discard.
- We can then **visualize** those remaining dimensions, or use them for data analysis.

One more detail of PCA we've ignored so far: all of variance comes from some "baseline": an **average** position, which all of our data points are **shifted** from.

- We could view this as a "**template**", which our PCA components point away from, for any given data point.

In PCA, we didn't focus on this as much, because it was just the average of all of our data points: useful, but simple.

This concept becomes interesting in the more complex case of non-linear autoencoders: it isn't so easy to **guess** what's the "average"/typical example in the latent space.

- **Example:** Suppose we plot all of the data points labeled "7", based on their latent representations.
- We could see what happens if we average those, and then convert it back into a picture: it would teach us about how our model sees the number 7 in general.

The result depends on your exact choice of encoder, but it's not always what you expect: that's why it's so **informative**!

13.5 Advanced Encoders and Decoders (Optional)

We can build on this framework to create models for different, practical tasks.

13.5.1 Generative Networks (Optional)

One useful application of our latent space is to **artificially** create more data which is **similar** to the data we already have.

How?

We saw that, with **some** very effective autoencoders, we find some useful **structure** in our latent space:

- Data points which were **nearby** in latent space, appeared **similar** in the input space.

If we successfully train an autoencoder with this property, then creating new data is possible:

- We take our real data, and slightly modify it in the latent space. This should be similar to our real data, but not exactly the same.

How do we train our autoencoder to do this? We'll discuss one popular approach below: the "variational autoencoder" (VAE).

This is the basis of a **generative network**:

Definition 23

Generative networks are networks which are used to **generate** artificial data, which is similar to **real** data used to train it.

These often come in the form of an autoencoder:

- We start by using real data to **train** the autoencoder, and create a **latent space**.
- We use those same data, and **create** new data that is "close" to the real data, within our latent space.
- Finally, we use the decoder to **reconstruct** our **new data**.

This technique relies on the assumption that data points which are **close** in the latent space, are **similar** in the input space.

These models have many applications:

- Augmenting (increasing the size of) **smaller** datasets
- Reducing **overfitting**, by perturbing data
- Art and Media

– **Example:** Superresolution: filling in "details" in images where they need to be believable, not necessarily real

This, of course, can cause problems if you don't know you're working with fictional data!

And more, which we'll discuss below.

13.5.1.1 Variational Autoencoders (Optional)

We've introduced the general outline of a **generative network**: we create artificial data, using our latent space.

But we have **two problems**:

- First: how do we **generate new** data points, "close" to the real data? What's our procedure?
- Second: this relies on the assumption that our **encoding** is **smooth**: nearby points in latent space represent similar information. How do we guarantee that?

We'll find that one technique addresses both of these problems: representing our data as a **probability distribution**.

One simple way to generate new data, is to **randomly** perturb a data point in the latent space, by a small amount.

The outcomes of random processes, like this, have a certain **probability** of occurring.

- So, the output of our encoder isn't a point in the latent space, but a **probability distribution** of possible points.

Definition 24

A **variational autoencoder**, rather than creating a **single** encoding for a data point, encodes a **probability distribution**.

- This distribution represents different possible encodings, for the **same** input.

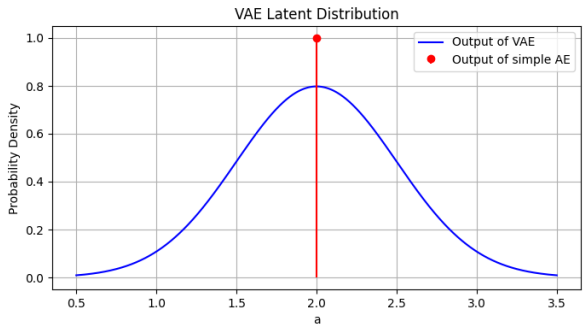
Then, we **sample** from this distribution, to randomly select an encoding.

Finally, we **decode** this to get our "modified" data point.

This process actually **regularizes** our model: it needs to be able to re-produce the input, even if it's slightly modified in the latent space.

- We're also restricting our probability distribution to have a certain shape/structure (like a normal distribution).

Example: Suppose that we created a 1-D encoding, and for this particular data point, we encode it as $a = 2$. If we compare the simple AE to the VAE:



We've gone from "always output 2" to "output a normal distribution centered on 2".

Technical comment, that isn't important to this class:

The simple AE distribution depicted is the **dirac delta function**, where all the probability is 'stored' at $a = 2$.

We can directly compare the process in both cases. First, the **simple autoencoder**:

$$x \xrightarrow[\text{encoding}]{} a = e(x) \xrightarrow[\text{decoding}]{} \tilde{x} = d(a) \tag{13.7}$$

Next, the **variational autoencoder**:

$$x \xrightarrow[\text{encoding}]{} \overbrace{p(a|x)}^{\text{Distribution}} \xrightarrow[\text{sampling}]{} \overbrace{a \sim p(a|x)}^{a \text{ sampled from } p(a|x)} \xrightarrow[\text{decoding}]{} \tilde{x} = d(a)$$

Now, we can handle the second problem: how do we ensure that nearby points in the latent space are **similar**?

Well, if we want our model to do something, we **train** it with that goal in mind.

- As we've designed it, our VAE **already** generates points "nearby" to our encoding, using its probability distribution.
- Because they're nearby, we want the **original** data and the **sampled** data to be relatively similar.

We usually use something like the normal distribution above: most data is close to the mean.

Concept 25

Unlike a simple AE, our VAE **doesn't** return exactly the same data that it started with:

- Instead, our VAE **samples** the latent space, **nearby** to the "pure" encoding.

Because the sampled and original data are nearby, we want them to be **similar**.

How do we compare the original and sampled data? The original data is given as the input. Meanwhile, we convert the sampled data by **decoding** (reconstructing) it.

We want our original (input) data x and sampled (+decoded) data $\tilde{x}$ to be **similar**.

- This is similar to our goal for the simple autoencoder: there, too, we wanted our original and re-constructed data to match.

Concept 26

Just like with a simple autoencoder, we want the **input** and **output** of our VAE to be close to **equal**.

$$x \approx \tilde{x}$$

This serves a few functions:

- Same goal as we had before: being able to **reconstruct** the input shows that our encoding preserves meaningful **information**.
- The simple encoding of x, and the encoding that produces $\tilde{x}$, are **not** the same, but they are **similar**.
  - This helps us train our model to "**smooth out**" its surface: we teach it that nearby points should be similar.

So, we **train** our model with this goal in mind.

To help "smooth out" the VAE representation further, we often add a **regularizer** term to the loss.

- We won't go into detail on this feature here.

13.5.2 Adversarial Optimization (Optional)

We'll make a brief detour, to discuss a different way we can generate **artificial** data.

In order to "test" your network, and see how **robust** it is, you might want to deliberately find examples it **struggles** with, but *shouldn't*.

- We could do this by just running a large amount of data, and manually looking for examples that fail, despite seeming "obvious" to humans.
- But this is labor-intensive, and **expensive**.

Instead, we introduce a different way to create so-called "**adversaries**": examples specifically designed to "trick" our model.

We want a data point that:

- **Should** be classified correctly, and would be classified correctly by humans
- But the machine fails, despite looking **similar** to valid data points.

We'll start with a valid, **correctly** labelled data point: this handles the first condition: "should have been labelled correctly".

But we want to **modify** it so that it's labelled incorrectly.

- If our model isn't **robust** enough, we might be able to **confuse** it, without changing the data very much at all.

How do we modify it most efficiently? Using the **gradient**.

Previously, we used the gradient to compute, "what's the most efficient way to change my **model**, to **decrease** my loss?"

$$W - \eta \frac{\partial L}{\partial W} \tag{13.8}$$

This time, we want to ask, "what's the most efficient way to change my **data point**, to **increase** my loss?"

So, we want the gradient between the loss and the data point.

$$x + \eta \frac{\partial L}{\partial x} \tag{13.9}$$

We take **steps** in this process, repeatedly changing our gradient to match the model.

We continue this process until our data point is successfully classified **incorrectly**.

- Often, we can accomplish this without significantly modifying our data point.

This time, instead of traveling across the parameter space, we're traveling across the input space!

Definition 27

Adversarial training is a way to "trick" a model into making inaccurate predictions:

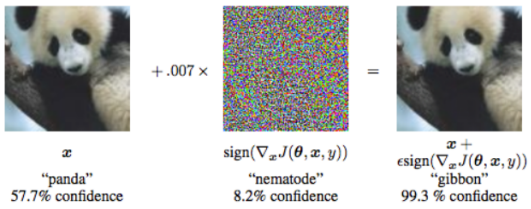
This is done by **training** data points that appear very similar to real data, but exploit weaknesses of the model.

To accomplish this, we take a real, correctly labelled data point, and slowly apply gradient descent **to the data point**, gradually increasing the loss:

$$x_{new} = x_{old} + \eta \frac{\partial L}{\partial x}$$

With some training, we can design a new data point that our model evaluates incorrectly, despite being very similar to the original data point.

Example: Here's a *real example* of applying this to image classification:



This panda, despite no visible change in appearance, is now a gibbon. (Source: Ian Goodfellow et al., 2014

This kind of data is actually incredibly valuable:

- If we use it to train our model further (i.e., teaching it that it's wrong about these data points), it tends to become **more robust** against this kind of attack.

But, we need to be careful in this process:

Clarification 28

When creating adversarial data, we have to set a **termination** condition:

If we continue gradient descent too long, our data point will change **classification**, but it can also actually change enough that it **should** be identified differently.

Example: If you do gradient descent to turn a picture of a "9" into a "1", and it looks like a "1" to a human, it should be reasonable that the model makes the same decision.

We want to find the point where the data is different enough to be deceptive, but not different enough that it's obviously, genuinely a new data point.

Adversarial data is widely useful:

- Improving model robustness
- Defending against security threats
- Learning more about the nature of your model

13.5.3 Generative Adversarial Networks (Optional)

We've developed two useful ideas:

- Generative networks: used to generate artificial, plausible data
- Adversarial data: data designed to exploit weaknesses in a model

We can combine these ideas to create a powerful tool, called, appropriately, a **Generative Adversarial Network**.

Here's the general idea:

- **Generators**: we want to generate artificial data, that closely reflects the **original** distribution.
- We found that an **adversary** could teach us (and in turn, our models), their weaknesses, by actively seeking them out.
  - So, we'll create an **adversary** for the generator: the **discriminator**. This punishes our model for creating data that is "detectably different" from real data.

The generator will create "adversarial data": this data is designed to **trick** the discriminator into thinking it's real.

The discriminator will try to learn how to tell the two apart, and provide feedback to the generator, telling us how it made a mistake the previous time.

This feedback loop gradually improves how "realistic" our data generated is, through a form of **unsupervised** learning.

The two models are "supervising" each other!

Definition 29

A **Generative Adversarial Network** is a model that comes into two opposing, or "adversarial", parts:

- The **generator** is trained to create "fake" data, that looks as plausible and realistic as possible.
- The **discriminator** is trained to detect fake data

When one of these models fails, they teach the other model what they did wrong, to improve.

- This process repeats until the discriminator accuracy reaches 50%: if it's equally likely to mix up real/fake data, our generator has become indistinguishable from real data.

This is a sort of "arms race" between the two halves of our model.

Clarification 30

Why is best performance for the generator at **50%**?

If it was 100%, then the model always assumes the generator is real, and the real data is fake.

- But that's not true: the real data is *also* real. The discriminator isn't doing its job correctly anymore.

Another perspective: if you knew your discriminator was always **wrong**, then you could easily create a discriminator that was always right: just do the opposite of what you got before.

In short: 50% is the best you can do, because your discriminator is completely **unsure** of its answer: which is what it really means to be "**indistinguishable**".

GANs have been high successful: this procedure not only improves robustness, but often creates a generally improved model.

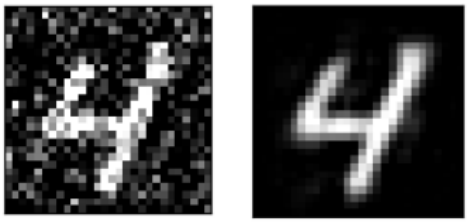
They're useful for creating very effective generators of new data, and have been particular useful in the past for image generation.

GANs are still relevant, but if you want a more **modern** approach, you could look into **Diffusion** models, like Stable Diffusion.

13.5.4 De-noising (Optional)

One useful application of autoencoders is **de-noising**.

- We want to turn a noisy input into a less-noisy input.



The left is our input, right our desired, cleaner output.

The process for creating this kind of autoencoder is straightforward enough: we give noisy data and encourage the model to match the original, noiseless version.

Concept 31

Autoencoders can be used for **de-noising**:

By training with

- Noisy data as input
- Noiseless data as desired output

You can teach the model to create a latent representation that's resistant to this kind of noise.

13.5.5 Attention (Optional)

Transformer networks are a very modern, very powerful approach to many problems, most famously **language processing**.

As of writing, GPT-4 is the most famous example.

In order to create detailed context within a sentence, these models use a technique called "**self-attention**", which has proven to be incredibly powerful for working with language.

- Attention is an in-depth topic that deserves its own section; we'll **skip** the math.
- The main idea: attention helps determine how words create **context** for each other.

Transformers aren't **limited** to language, but we'll focus on that use case, for ease of explanation.

Concept 32

Attention, at a very high level, is based on the idea that:

- Different **words** in a sentence have different levels of **importance** to each other.

The words that are more **important** to a particular word X, are a bigger part of the **context** you use to understand that word.

With that in mind, you can apply the **context** from each other word Y, to better understand the meaning of word X.

Example: "The **blue dog** bites the **red ball**": the word '**red**' is describing the '**ball**', so it is **less** important for understanding '**dog**' in this sentence.

Based on that...

Definition 33

Attention is a mechanism for determining how, for a **pair of words**, one word X might might be **important** to understanding the other word Y, contextually.

In other words, X might require your **attention**, if you want to understand Y.

- Attention is applied to **every** pair of words in a sentence: we measure every word's impact on every other word.
- Finally, for each word, we **integrate** information from the other words, based on how "important" they were.

Attention has the benefit of allowing us to analyze our entire prompt simultaneously, speeding up the process.

Self-attention is used by a single sequence of text, by itself: it "learns about" itself.

Clarification 34

A few lingering comments:

- Word X and Y are typically asymmetric: X may not affect the meaning of Y the same way as Y affects X.
- This meaning is **contextual**, not the "importance" of each word in isolation.
- We "integrate information" by taking a linear combination of each word: a larger weight is given to words which are more "important" to word Y.

"Multi-headed attention" simply refers to having several of these attention mechanisms, applied to the same input in parallel.

So, each "attention head" receives the same data, like neurons in the same layer of a network.

13.5.6 Transformer Networks (Optional)

Now that we loosely understand attention, we can think about the bigger picture.

The goal of a transformer is to **predict text**.

At a very high level, transformers have a structure *similar* to **autoencoders**. They're broken into the same sort of two parts, though their internal structure is a bit **different**:

- **Encoder**: this is actually a **stack** of several encoders, one after another. This (presumably) creates an **internal representation** of the "meaning" of the input text, similar to a latent representation.
  - Each encoder contains a fully connected network, as we've shown above, but also a "self-attention mechanism".
- **Decoder**: a stack of decoders, one after another. Based on the **internal representation**, it creates predicted text.
  - Each decoder also contains an FC network+attention mechanisms.
  - Once our decoder predicts some text, that's part of the "**past text**": this gets included alongside the internal representation, for future predictions.

You could say that this converts the input "prompt" into something similar to the "latent representation", which retains our key information.

This would create the "response" to your "prompt".

Concept 35

A transformer network follows a structure with some similarities to auto-encoders: using an **internal representation**.

- Converting its **input** (prior text) into an **internal representation**, similar to a latent representation.
- Converting that **representation** into an **output** (predicted text)

However, there are key differences:

- As the output creates new text, that is **included** with the internal representation.
- Our "predicted text" will **not** be the **same** as our past text.

Why multiple encoders?

Concept 36

The **multiple** encoders are used to gradually find "more **complex**" patterns (or concepts), as we move through more layers.

- The idea is that, the first encoder combines words to finds **basic**, simple language structures.
- The second encoder has access to these simple structures, and can **combine** them together to create a more complex structure.
- Each encoder is combining the results of the **previous** layer, to create something more complex.

This is similar to the structure and motivation behind **convolutional neural networks**, which we will cover later.

Definition 37

A **transformer network** is a model made of encoders and decoders that use **attention** to determine the **structure** and **context** of the input "prompt", and create an output "response".

The structure of a transformer network is:

- Several "layers" of **encoders**, that take the input and interpret it, creating the internal representation
- An **internal representation** of the input, and the previous output of the decoder
- Several "layers" of **decoders** that turn this representation+output, into more **predicted text**

Predicted text is given as a **probability distribution**, using softmax.

- We select an element from this probability distribution before moving on to the next word.

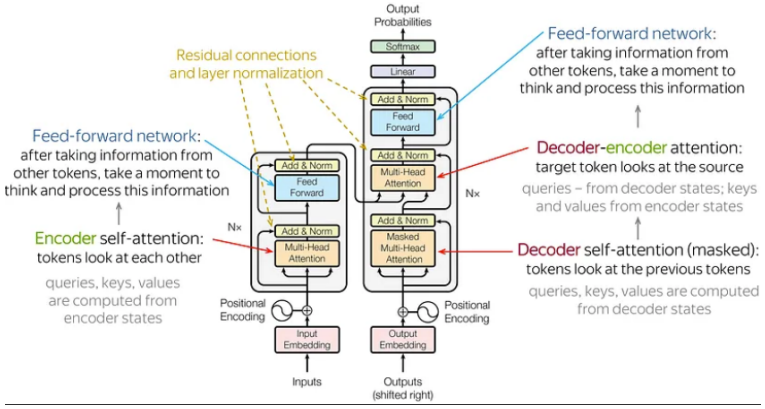
We've skipped over many important details of transformers, but this gives us the gist.

In particular, we've skipped some components, like normalization layers.

Clarification 38

A few key differences between an autoencoder and transformer networks:

- The input and output are the same in an autoencoder.
- Transformers use attention mechanisms.
- The internal representation in an autoencoder ("latent representation") is **smaller** than the input or output.
- They serve different purposes.



If you want a more in-depth explanation, go to this very helpful resource, and the source of this lovely diagram!

13.6 Terms

- Unsupervised Learning (Review)
- Clustering (Review)
- Compression
- Decompression/Re-construction
- Encoder
- Decoder
- Autoencoder
- Latent Representation
- Latent Space
- Bottleneck
- Dimensionality (Review)
- Overcomplete Autoencoder
- Downstream Task (Review)
- Semi-supervised Learning
- Principle Component Analysis
- Singular Value Decomposition
- Generative Networks
- Variational Autoencoders
- Transformer Networks

Optional

- Adversarial Data
- Generative Adversarial Networks
- De-noising
- Attention
- Transformer Networks
- Internal Representation