Lecture 11 · Part II · Extended notes

Autoencoders and representation learning

Dataset: Fashion-MNIST, 12,000 training images

Why these notes exist

This lecture sits outside Géron, the course's primary text. For it *these notes are the primary source*, and it is examined on exactly the same terms as the textbook lectures. Everything in the slide deck is here, with the arguments written out at length rather than compressed onto a slide; the numbers are the ones the accompanying notebook prints, and every one of them is a measurement on the dataset named above rather than a figure quoted from a paper.

Contents

1 Where we are **2**

2 The problem **3**

2.1 The data, and the one rule about labels 3

3 What an autoencoder is **3**

3.1 The bottleneck is a constraint on what the network can reach 3

4 The undercomplete autoencoder **4**

4.1 The model 4

4.2 Run it at $d = 32$ 4

4.3 The code size, swept 5

4.4 What the loss sees, and what it does not 5

5 The code as a representation **6**

5.1 The code as a picture 6

5.2 What a representation is for 6

5.3 Similar items: nearest neighbours 7
5.4 The bill 7
5.5 Groups: k-means on the code 8

6 The denoising autoencoder **8**

7 What can go wrong, in one place **9**

8 The variational autoencoder **9**

9 Where this leads **10**

10 The notebook **10**

11 Design exercise: "see similar" on a fashion marketplace **10**

12 Exercises **12**

1 Where we are

Every network of Part II so far needed a label for every image. Lecture 7 built a first network and said what a layer computes; Lecture 8 wrote the training loop out in five lines. Each of them was scored by a label it was given, and so were the systems of Lecture 9 and Lecture 10: a harmful-content detector designed end to end, and a model audited for what it relies on and for whom it fails.

This lecture removes the label. A network that is asked to *reproduce its input* through a code too small to hold it has to learn what the inputs have in common, and the code it learns is a description of the input that nobody wrote down. The lecture is about that code: how it is trained, what it gains over the best linear compression, and what it is good for.

What you should be able to do afterwards.

- Say what an autoencoder is trained to do, and what stops it learning the identity.
- Say what the non-linear model buys over the best linear compression, and when a trained approximation loses to an exact one.
- Read a learned code as a representation: measure it on retrieval and clustering against the baseline it was built from, and say what it bought.
- State what the denoising corruption does, and when the denoising autoencoder is the better choice.

2 The problem

A marketplace holds millions of product photographs uploaded by its sellers. The category field is free text, often empty, sometimes wrong. The product team wants three things from the photographs alone: given one garment, the ten most like it, in milliseconds, across the whole catalogue; a first structure for a catalogue that has none; and the uploads that are not what the catalogue holds, flagged before a shopper sees them. None of these comes with a label.

2.1 The data, and the one rule about labels

Fashion-MNIST stands in for the catalogue: seventy thousand greyscale images of garments, 28 by 28 pixels, ten classes. The split is the one Lecture 7 and Lecture 8 used, so the numbers compare: a validation set of 5,000 carved out of the training images from a fixed seed, the first 12,000 of the rest as the fit set, and the official 10,000 test images as the catalogue.

Quantity	Count
Images to fit on	12,000
Validation images	5,000
Test images, the catalogue	10,000
Pixels per image	784
Classes	10

The labels are loaded, and used for exactly one thing

Scoring. Nothing in this lecture is fitted on a label. Precision at ten and the clusters' purity both read the labels *after* the fact, to say whether a representation learned without them means anything.

3 What an autoencoder is

Definition 3.1 (Autoencoder)**.** *An **encoder**, which turns the 784 pixels of an image into d numbers, and a **decoder**, which turns the d numbers back into 784 pixels, trained together so that the output reproduces the input. The d numbers are the **code**; d is its size. The loss is the mean squared difference between the input's pixels and the reconstruction's.*

The loss is the squared loss of least squares with the input standing in for the label, and the training loop is the five-line loop of Lecture 8 with its third step, the loss, comparing the output with the input instead of the labels. Nothing in the loop knows there is no label.

3.1 The bottleneck is a constraint on what the network can reach

A regulariser — dropout, a penalty on the size of the weights, randomly altered copies of the training inputs — works by shrinking the set of functions the network can reach, so that fitting the training data no longer means memorising it. An autoencoder needs the same thing, for a

sharper reason. With $d = n$ the identity is reachable, the loss is zero, and nothing was learned. Three constraints are in common use.

Constraint	What it forbids	Name
$d \ll n$	copying: 784 numbers cannot pass through 32	undercomplete
corrupt the input, keep the target clean	copying: what comes in is not what must go out	denoising
penalise the code	using every coordinate for every image	sparse

This lecture uses the first two.

4 The undercomplete autoencoder

4.1 The model

Two ReLU layers around the bottleneck, and a sigmoid on the output so that a reconstruction is an image in $[0, 1]$:

```
class AutoEncoder(nn.Module):
    def __init__(self, d, hidden=256):
        super().__init__()
        self.enc = nn.Sequential(nn.Linear(784, hidden), nn.ReLU(),
                                 nn.Linear(hidden, d))
        self.dec = nn.Sequential(nn.Linear(d, hidden), nn.ReLU(),
                                 nn.Linear(hidden, 784), nn.Sigmoid())

    def forward(self, x):
        return self.dec(self.enc(x))
```

`enc` and `dec` are attributes on purpose: the code is read by calling the encoder alone. At $d = 32$ the model has 419,120 parameters, almost all of them in the two 784-wide layers. The loop is the one of Lecture 8 with `F.mse_loss(out, xb)` in place of the cross-entropy: twenty epochs of Adam at 10^{-3}, batches of 128, 12,000 images, which a CPU handles. Everything else in the loop — the optimiser, the batches, the evaluation — carries over unchanged, because nothing in it ever depended on the target being a label.

4.2 Run it at $d = 32$

Model, test error per pixel	Error
PCA with 32 components — the best any linear compression can do	0.01519
Linear autoencoder: one linear encoder, one linear decoder	0.01523
Non-linear autoencoder, 419,120 parameters	**0.01275**

About a sixth below the best linear compression. Without ReLUs a network lands where PCA is, to the third digit: a linear network can only find the best flat compression. The sixth is what a curved map buys over a flat one. It is the entire justification for the two ReLU layers. Training error 0.01188 against validation 0.01276: a small gap, so little overfitting, and small because the bottleneck is doing the regulariser's job.

4.3 The code size, swept

The same three models at $d = 2, 8, 32, 128$, each scored on the test set, under one recipe — the same epochs, learning rate and width — so that a difference between columns is the code size and not the budget.

d	PCA (exact)	Linear autoencoder	Non-linear autoencoder
2	0.04610	0.04611	**0.03041**
8	0.02662	0.02665	**0.01617**
32	0.01519	0.01523	**0.01275**
128	**0.00641**	0.00649	0.01179

The linear network tracks PCA at every d. The non-linear one wins where the bottleneck binds and, at $d = 128$, *loses*.

Failure condition — a trained approximation against an exact optimum

PCA is computed exactly, with no training. The network is a *trained* approximation to the best of a larger class, and at $d = 128$ its budget — one hidden layer of 256, twenty epochs — leaves it above what PCA gets with no training at all. More capacity is not more accuracy until the optimiser has found it. The notebook's `try` on this cell widens the hidden layer, and the $d = 128$ row is the one that moves.

4.4 What the loss sees, and what it does not

Figure 1 shows eight test garments reconstructed four ways. Rows two and three — PCA and the linear network — are indistinguishable: the same reconstruction. Row four is sharper. Row five, at $d = 2$, is the shape of a kind of garment and nothing of this one.

Squared error on pixels is a distance between two arrays, and it has no idea which pixels matter. It rewards blur: the average of a class is close, in pixel distance, to every member of the class, so a model short of capacity reconstructs the average and is scored well for it. It is blind to a shift: move a sharp edge one pixel and the error is large; smear it over three and the error is smaller, so the sharper reconstruction is the worse one. It measures pixels, not garments: a missing button costs nothing, a slightly different grey over a whole sleeve costs a lot.

Failure condition — a loss that rewards the average

Reconstruction error falling is not the same as reconstructions improving. Look at them. A code chosen by the loss alone is a code tuned to reproduce the common, and the common is what you already knew.

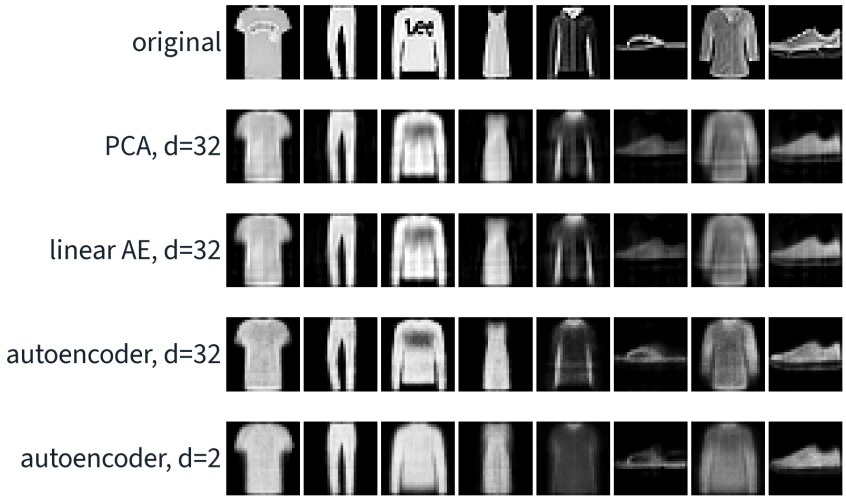


Figure 1: Eight test garments: the originals, PCA with 32 components, the linear autoencoder at 32, the non-linear autoencoder at 32, and the non-linear autoencoder at 2.

5 The code as a representation

5.1 The code as a picture

With $d = 2$ the code can be drawn. Every test image becomes a point, and the colour is its label, which the network never saw (Figure 2). Footwear sits apart from clothing, trousers apart from both, and the five torso garments on top of one another: two numbers cannot tell a shirt from a pullover, and the reconstruction error at $d = 2$ already said so.

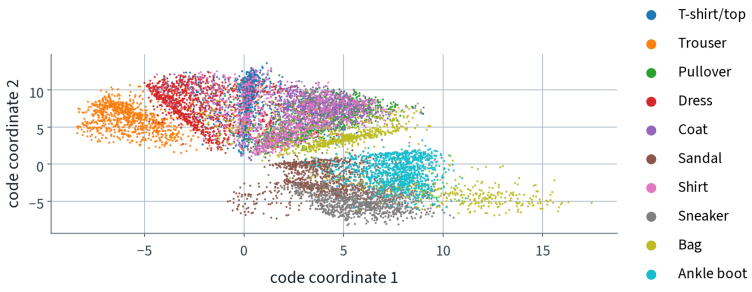


Figure 2: The two-dimensional code of every test image, learned without labels and with the colour added afterwards.

5.2 What a representation is for

Thirty-two numbers per garment, computed from the pixels, learned from the pixels. The product questions become geometry in that space.

Question	Operation on the codes	Scored by
similar items	nearest neighbours by Euclidean distance	precision at ten, against the labels
groups	k-means with $k = 10$	mean purity of the clusters, against the labels

Every one of them has a baseline that needs no learning: the same operation on the 784 raw pixels, and on the 32 PCA scores. A learned code that cannot beat its own input has learned nothing the question needed.

5.3 Similar items: nearest neighbours

The catalogue is the 10,000 test images; a thousand of them, drawn once from a fixed seed, are the queries, each searching the other 9,999 by Euclidean distance. With $\|a-b\|^2 = \|a\|^2 - 2a\cdot b + \|b\|^2$ the thousand searches are one matrix product; a query's own entry is set to infinity so that no image is its own neighbour.

Definition 5.1 (Precision at ten)**.** *Of the ten images returned for a query, the fraction whose class is the query's, averaged over the queries. The label is read here and nowhere else.*

Space	Numbers per item	Precision at 10
Raw pixels	784	76.71%
PCA scores	32	76.47%
Linear autoencoder code	32	77.07%
Autoencoder code	32	**77.32%**

Four spaces within a point of one another. The code did not beat the pixels, and it could not have: nothing in the reconstruction loss is about which garments are *similar*. What it bought is the second column. Swept over the code size, the non-linear code scores 58.18% at $d = 2$, 74.30% at $d = 8$, 77.32% at $d = 32$ and 78.27% at $d = 128$: two numbers lose a fifth of the precision, eight recover most of it, thirty-two match the pixels, and 128 add a point. From eight upwards the representation question is a cost question.

5.4 The bill

A nearest-neighbour search compares the query with every item in the catalogue, so the length of the code is the cost of the search and of the index that holds it.

Per item	Raw pixels	Code	Factor
Numbers stored	784	32	24.5
Multiplications per comparison	784	32	24.5
Bytes, as float32	3,136	128	24.5

At equal precision the last column is the factor on the index's memory and on the query's cost, and for a photograph of real size the 28 by 28 of this dataset becomes thousands of pixels and the factor grows with it. The case for a learned representation in a product is, first of all, this table. Quality is the problem of the lectures that train a representation on pairs.

5.5 Groups: k-means on the code

Cluster the test set into ten groups with k-means, with ten restarts from a fixed seed so that a comparison between spaces is not a comparison between starts, and score each clustering against the labels by its purity: for each cluster, the share of its members that belong to its commonest class, averaged over the ten clusters.

Space, $k = 10$	**Numbers per item**	**Mean purity**
Raw pixels	784	0.6473
PCA scores	32	0.6471
Autoencoder code	32	0.5891

Pixels and PCA scores tie, and the learned code loses six points: nothing in the reconstruction loss is about which garments belong together, and here it shows. The structure k-means finds is the structure the two-dimensional picture drew — footwear, trousers, bags, torso — and it is in the pixels already. The code carries it in 32 numbers, at a price the audit shows.

6 The denoising autoencoder

Corrupt the input, keep the target clean:

```
for xb in batches(X_fit):
    xin  = (xb + 0.3 * torch.randn_like(xb)).clamp(0, 1)   # corrupt the input
    opt.zero_grad()
    loss = F.mse_loss(net(xin), xb)                        # the target is clean
    loss.backward(); opt.step()
```

The second constraint from the table: what comes in is not what must go out, so copying is impossible even with a wide code. The noise is drawn fresh on every batch — the same corruption twenty times over is a fixed dataset, and can be memorised — and it is data augmentation, training on randomly altered copies of each input, with the clean image as the target: a regulariser applied to the training input only.

Error against the *clean* test image, per pixel	**Error**
The noisy image itself	0.05264
The plain autoencoder, fed the noisy image	0.02408
The denoising autoencoder, fed the noisy image	**0.01551**
The denoising autoencoder, fed the clean image	0.01781

Three things in four rows. The bottleneck alone removes half the noise — noise does not fit through 32 numbers either. Training on corrupted input removes most of the rest. And the price: on clean input the denoiser is a worse autoencoder, 0.01781 against 0.01275, and its code retrieves slightly worse, 75.87% against 77.32%. A regulariser is a trade: the constraint that stops copying also stops some fitting.

Which one to use. The denoising autoencoder, when the images it will process are noisy — phone photographs, scans, low light: a plain autoencoder reproduces part of the noise, and one trained noisy-in, clean-out has learned to remove it. The plain one, when the images are clean: there it is slightly better. The denoiser needs only clean images to train, because the noise is added during training.

7 What can go wrong, in one place

When	What happens	Measured
The code is as wide as the image	the network can copy its input: zero loss, nothing learned	at $d = 784$ the loss can reach 0
The code is wide, but not that wide	the trained network does worse than PCA	$d = 128$: 0.01179 against 0.00641

And for every autoencoder: squared error rewards a blurry average, so a lower loss does not mean better-looking images. Look at the reconstructions.

8 The variational autoencoder

Beyond the syllabus, for context.

The autoencoders so far give one code per image and nothing in between: decode a point between two codes and there is no reason the output is a garment. The **variational autoencoder** makes the encoder output a distribution — a mean and a spread for each number of the code — samples a code from it, and adds a second term to the loss that keeps every such distribution close to a standard normal. That term is a regulariser on the code — the third row of the table of constraints — and it fills the code space: a code drawn from a standard normal decodes to something plausible, which makes the decoder a generator.

A sampled code breaks the chain rule of Lecture 8: there is no derivative of a random draw with respect to the parameters that chose its distribution. The **reparametrisation trick** moves the randomness outside the graph: standard normal noise is drawn first and treated as an input, and the code is the mean plus the spread times the noise, an ordinary computation on the encoder's outputs. Reverse mode differentiates through the mean and the spread as through any layer; the draw is never differentiated. Not examined; it is here because every generative model you will meet after this course contains this move.

9 Where this leads

Representation	Trained by	What distances in it mean	Lecture
autoencoder code	reconstructing the input	pixel distance, through a curved map, in a basis nobody chose	11
a classifier's penultimate layer	labels	whatever separates the classes	13
a dense-retrieval embedding	query–document pairs	relevance	20
a contrastive embedding	pairs that belong together	similarity, as the pairs defined it	23

Today's code matched the pixels on retrieval and could not beat them, because the loss never saw a pair. The rest of the course is, in part, the story of what happens when it does.

10 The notebook

The notebook runs on the CPU, by choice: every network in it has under half a million parameters, and a CPU run is the one that reproduces on the machines students have. It fits PCA and trains the linear autoencoder as the baseline; trains the non-linear autoencoder, sweeps d, draws the reconstructions and the two-dimensional code; measures precision at ten and k-means in four spaces; and trains the denoiser. The numbers in these notes are its run.

What to change

Set `NOISE = 0` in the denoising section and run again. The denoising rows become the plain autoencoder's, to the digit — they were always the same model with one line changed.

11 Design exercise: "see similar" on a fashion marketplace

The lecture closes with a design exercise in the manner of an ML system-design interview, on the product this lecture's method is built for.

The task. An online fashion marketplace sells garments listed by thousands of independent brands and shops. The product is its shopping app; the feature is a "see similar" strip under every item page, and a camera search that finds catalogue items like a photo the shopper took. The users are shoppers; the customers are the sellers, who pay a commission on every sale. The business measures the strip's click-through, the conversion it adds, and the return rate of what it sells. Given: millions of seller-uploaded photographs with an unreliable category field, the shoppers' click and purchase logs, and the sellers' stock and price feeds. Where a regulator exists: data protection, for the shoppers' photographs; consumer protection and brand owners, for what "similar" is allowed to show.

1. **Frame it honestly.** The output is a ranked list of catalogue items for one query image; the metric is precision at the strip's length on held-out items, read beside its anchor, the same

strip filled from the query's category at random. The label for scoring comes from what shoppers did — items viewed or bought in the same session, cleaned of the strip's own clicks — with the seller's category field as a weak second source. The baseline before any model: the same category, in stock, most popular first. *Pitfall:* reporting the reconstruction error. It is the training signal and the shopper never sees it; a code can reconstruct every photograph beautifully and return the wrong garments.

2. **The photographs are not as alike as the dataset's.** Sellers shoot on different backgrounds, at different sizes, on or off a model, and a code learned from the pixels learns the studio: the strip returns the same seller's other items. Crop to the garment, resize, normalise the background before anything is fitted, and check the strip for seller clumping. A camera-search image is personal data: encode on the device or discard after encoding, and never keep the photo. The same garment is listed by four sellers and in six colours: link variants to one item before splitting. Write down what one item is before the first fit.

3. **Split by item, and by seller.** Hold out whole items — every photograph and variant — and whole sellers for a second check, because the feature has to work for a shop that joined this morning. Choose the code size, the distance and the ranking rules on the validation items; read the test items once. *Pitfall:* a random split over photographs puts the red and the blue of one dress on both sides, the nearest neighbour of a held-out photo is its own item in another colour, and the precision says nothing about a garment the catalogue has never seen.

4. **Learn the code from the catalogue, and size it by the ranking.** Train the autoencoder on the whole catalogue of standardised photographs — no labels needed, so every listing counts — with a convolutional encoder once the photographs are real-sized; read the code from the encoder alone; sweep its size and pick the smallest that holds the validation precision. Denoising and augmentation as the regulariser: the same garment, shifted, in another colour, on another background, must land on the same code. *Pitfall:* choosing the code size by reconstruction error, which falls at every increase forever while the ranking metric flattens early and the index cost does not.

5. **The index.** Encode every catalogue item once, when it is listed, and store the code beside it; encode the query at request time and compare against stored codes, never against photographs. Exact search is a matrix product at a hundred thousand items; at millions an approximate-neighbour index trades a little recall for a hundred-fold speed, and that recall is measured. Normalise the codes to unit length and compare by cosine. *Pitfall:* a distance nobody chose — the code's coordinates are in a basis the optimiser landed on, and one coordinate with a large range dominates the Euclidean distance.

6. **Similarity is one feature of the ranking.** The strip ranks the nearest hundred by a second model that sees similarity, stock, size availability, price band, the seller's return rate and the shopper's history — a table, and a boosted ensemble on it — with business rules on top: in stock only, the shopper's size, nothing from a seller under review. *Pitfall:* the look-alike under a brand's page. Visual similarity is blind to who made the garment; under a luxury brand's listing the strip shows the imitations that look most like it, the brand — a paying customer — sees its page selling look-alikes, and the counterfeit complaints land with the regulator. Brand and seller constraints belong in the ranking layer, and the similarity model is never the last word.

7. **Release, monitor, retrain.** Release behind an A/B test on live traffic, read on the strip's

click-through, the conversion it adds and the return rate of what it sells, with the sellers' exposure held fair across both arms. Monitor the distribution of codes as the catalogue turns over with the seasons; retrain on a schedule and re-encode the whole catalogue in one atomic swap of the index. Keep a randomised slot in the strip, so that the clicks the next model trains on are not only the ones this model chose. *Pitfall:* two encoders in one index. A retrained encoder puts the same garment at a different code, so codes from two versions compare numbers that no longer mean the same thing and the strip fills with garments that are near nothing. Version the encoder with the index.

8. **Ship it.** Listings, then cropped and standardised photographs with variants linked into items, then a split by item and by seller, then an autoencoder on the catalogue with the code sized on validation precision, then normalised codes in a versioned index, then a ranking model with business rules on top, then an A/B test, then retraining, re-encoding and monitoring — each step named for the pitfall it avoids.

12 Exercises

The five set at the end of this lecture's deck, in the style of the written paper. The solutions are on Lecture 12's deck and in the exercise book.

1. At a code of 32 numbers, the test error per pixel is 0.01519 for PCA with 32 components, 0.01523 for a linear autoencoder and 0.01275 for the same autoencoder with ReLUs. Explain why the first two agree, and what the third bought. [4]

2. At a code of 128 numbers the non-linear autoencoder's test error is 0.01179, above PCA's 0.00641. A colleague concludes that non-linear autoencoders are worse than PCA at wide codes. State what is wrong with the conclusion. [4]

3. The code size is set equal to the number of pixels and nothing else constrains the network. State what the trained network computes, and what its code is worth as a representation. [3]

4. On noisy test images, the error against the clean image is 0.05264 for the noisy image itself, 0.02408 for the plain autoencoder and 0.01551 for the denoising one. On clean test images it is 0.01275 for the plain autoencoder and 0.01781 for the denoising one. A team will serve phone photographs. Say which autoencoder it should use, and why each result comes out the way it does. [5]

5. Nearest-neighbour retrieval in a 32-number autoencoder code scores the same precision at ten as in the 784 raw pixels. State what the code bought, and why it could not have bought more. [4]

Summary

An autoencoder is a network trained to reproduce its input through a code too small to hold it; the bottleneck is a constraint on what the network can reach, and without one the identity is reachable. A linear autoencoder lands where PCA is, the best flat compression: 0.01523 against 0.01519 at

$d = 32$. The non-linear model sits a sixth below it at $d = 32$ and above it at $d = 128$, because a trained approximation is not an exact optimum. Read as a representation, 32 numbers retrieve and cluster as well as 784 pixels, 77.32% against 76.71% on precision at ten, and no better, at a twenty-fourfold saving: the loss never saw a pair. Corrupting the input is a regulariser that stops copying and costs some fitting: use the denoising autoencoder when the images in use are noisy, the plain one when they are clean. The closing design exercise built a marketplace's "see similar" feature in eight steps — a ranking metric against its anchor, standardised photographs, a split by item, a code sized by precision, a normalised and versioned index, business rules above similarity, an A/B test — each named for the pitfall it avoids.