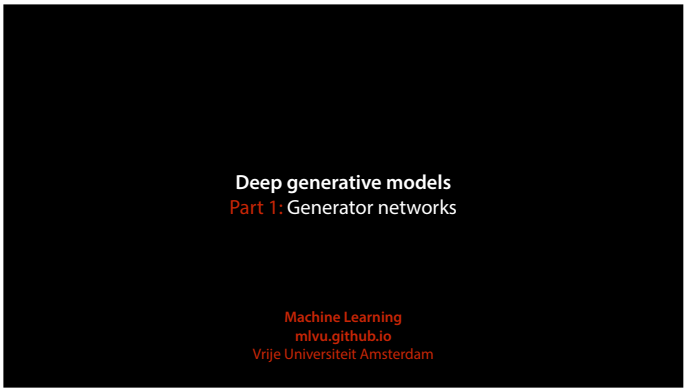
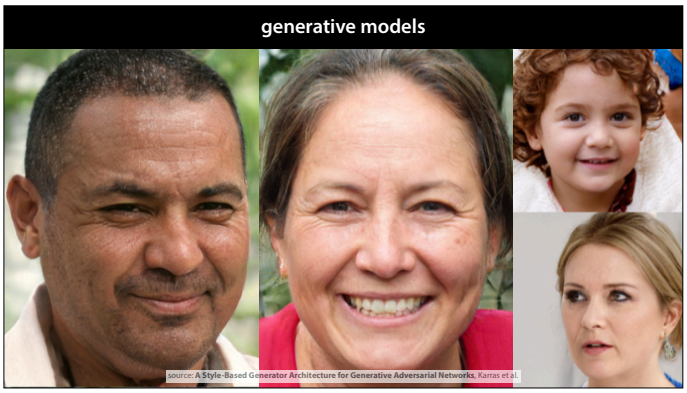


In this lecture, we'll look at generative modeling, The business of training probability models that are too complex to give us an explicit density function over our feature space, but that do allow us to sample points. If we train them well, we get points that look like those in our dataset.

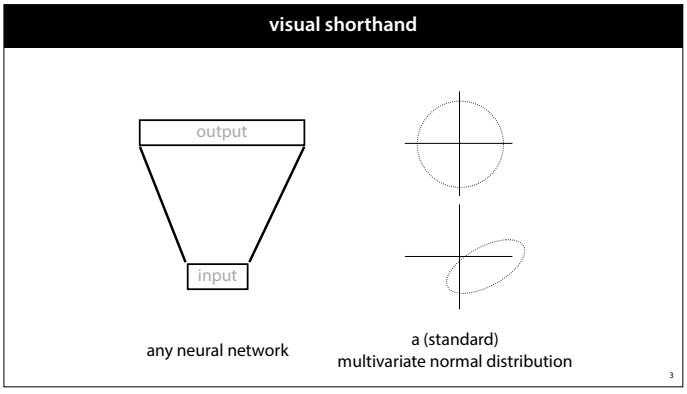
These kinds of methods are often combined with neural nets to produce very complex, high-dimensional objects, for instance images.

|section|Generator networks|
|video|https://www.youtube.com/embed/jAxUolSXGtg|



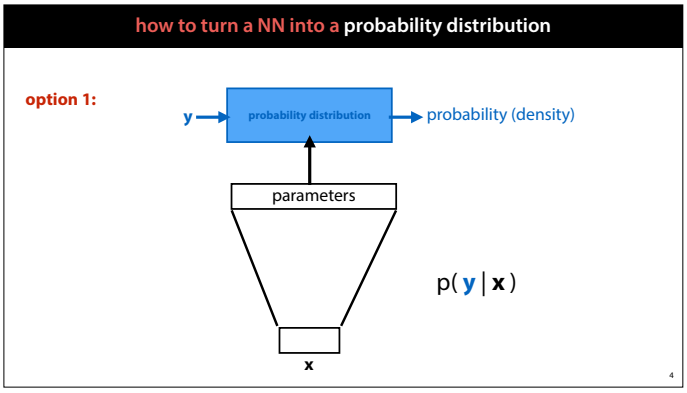
Here is the example, we gave in the first lecture. A deep neural network from which we can sample highly realistic images of human faces.

source: **A Style-Based Generator Architecture for Generative Adversarial Networks**, Karras et al.



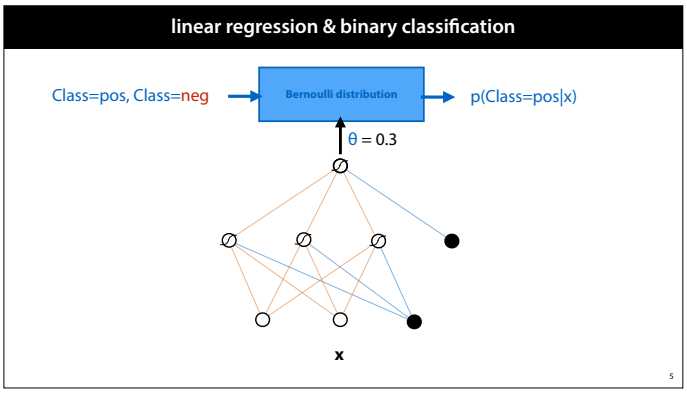
In the rest of the lecture, we will use the following visual shorthand. The diagram on the left represents any kind of neural network. We don't care about the precise architecture, whether it has one or a hundred hidden layers and whether it uses fully connected layers or convolutions, we just care about the shape of the input and the output.

The image on the right represents a multivariate normal distribution. Think of this as a contour line for the density function. We've drawn it in a 2D space, but we'll use it as a representation for MVNs in higher dimensional spaces as well. If the MVN is nonstandard, we'll draw it as an ellipse somewhere else in space.



A plain neural network is purely deterministic. It translates an input to an output and does the same thing every time with no randomness. How do we turn this into a probability distribution?

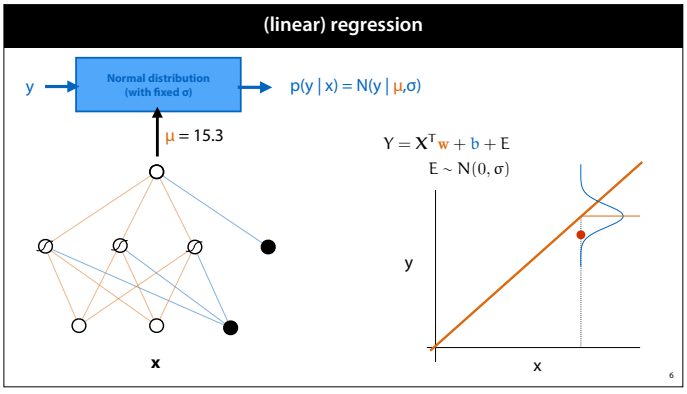
The first option is to take the output and to interpret it as the parameters of a probability distribution. We simply run the neural network for some input, let it produce some numbers as an output, and then interpret those as the parameters to a probability distribution. This distribution then defines a probability on a separate space. The network plus the probability distribution define a probability distribution conditional on the network input.



If this sounds abstract, note that it is something we've been doing already since the early lectures. For instance: to do binary classification, we defined a neural network with one sigmoid activated output node. We took that output value as the probability that the class was the positive one, but we could also say we're parametrizing a Bernoulli distribution with this value, and the Bernoulli distribution defines probabilities over the space containing the two outcomes "Class=pos" and "Class=neg".

If we do this with a multiclass problem and a softmax output, we are parametrizing a multinomial distribution.

Note that linear regression is just a special case of this with a very simple (one layer) neural net.



Another example is regression, either linear or with a neural network.

Here we simply produce a target prediction for x. However, what we saw in the previous lecture is that if we interpret this as the mean of a normal distribution on y, then maximizing the likelihood of this distribution is equivalent to the least squares loss function.

loss functions

output distribution	maximum log likelihood loss
Bernoulli	Binary cross-entropy
Categorical	Categorical cross-entropy
Normal (mean only)	Mean squared error loss
Normal (mean and variance)	$-\sum_i \ln \sigma_i + \frac{1}{2\sigma_i^2}(\mathbf{x}_i - \mu_i)^2$
Laplace (median only)	Mean absolute error loss

7

If we build a probability distribution parametrized by a neural network in this way, training it is pretty straightforward. We can easily compute the log-likelihood over our whole data, which then becomes a loss function. With backpropagation and gradient descent, we can train the parameters of the neural network to maximize the likelihood of the data under the model

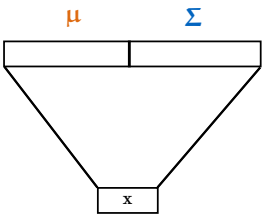
For many distributions, this leads to loss functions that we've seen already.

The loss function for a normal output distribution with a mean and variance, is a modification of the squared error. We can set the variance larger to reduce the impact of the squared errors, but we pay a penalty of the logarithm of sigma. If we know we are going to get the output value for instance i exactly right, then we will get almost no squared error and we can set the variance very small, paying little penalty. If we we're less sure, then we expect a sizable squared error and we should increase the variance to reduce the impact a little. This way, we get a neural network that tells us not just what its best guess is, but also how sure it is about that guess.

Neural networks are very poor at estimating how sure they should be, so take this with a grain of salt, but in principle, the machinery is there for the network to provide an indication of certainty.

---

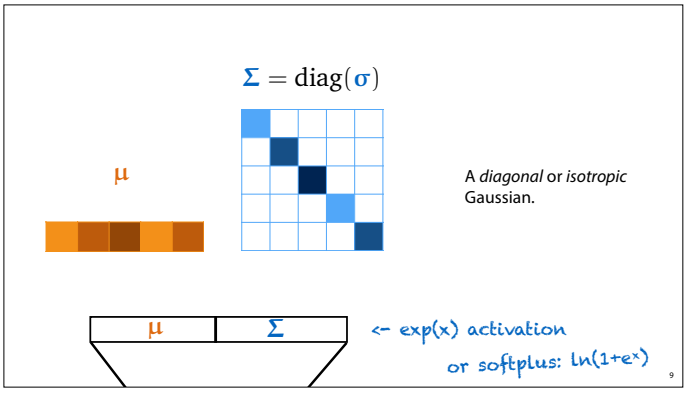
how to turn a NN into a probability distribution



$$p(\mathbf{y} \mid \mathbf{x}) = N(\mathbf{y} \mid \mu, \Sigma) \quad \text{with } (\mu, \Sigma) = f(\mathbf{x})$$

8

For a solution that applies to high-dimensional outputs like images, we can use the outputs to parametrize a multivariate normal distribution. Here we'll parametrize both the mean and the covariance matrix.



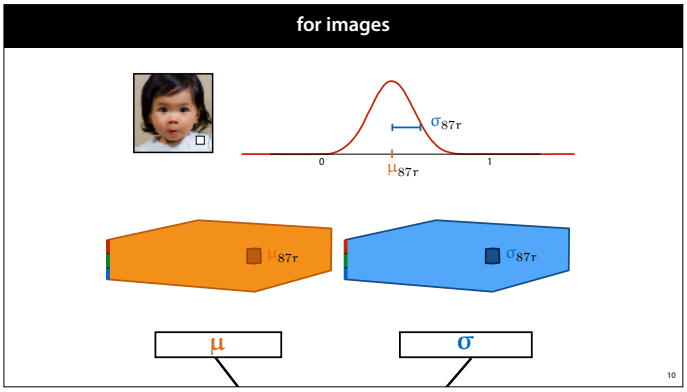
If we provide both the the mean and the variance of an output distribution, it looks like this (for an n-dimensional output space). We simply split the output layer in two parts, and interpret one part as the mean and the other as the covariance matrix.

Since representing a full covariance matrix would grow very big for high-dimensional outputs, we usually assume that the covariance matrix is diagonal (all off-diagonal values are zero). That way the representation of the covariance requires as many arguments as the representations of the mean, and we can simply split the output vector into two halves.

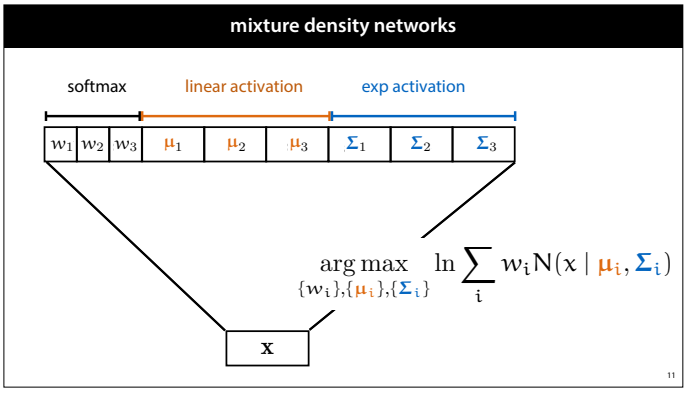
We will call the resulting normal distribution a diagonal Gaussian (the word isotropic is also used).

Equivalently, we can think of the output distribution as putting an independent 1D Gaussian on each dimension, with a mean and variance provided for each.

For the mean, we can use a linear activation, since it can have any value, including negative values. However, the values of the covariance matrix need to be positive. To achieve this, we often exponentiate them. We'll call this an exponential activation. An alternative option is the softplus function $\ln(1 + e^x)$, which grows less explosively.



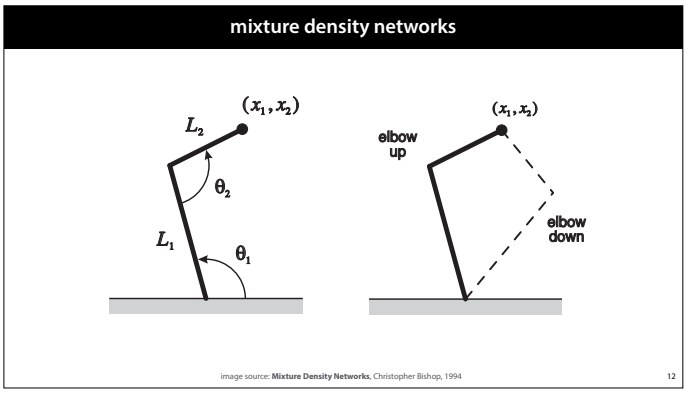
Here's what that looks like when we generate an image. The output distribution gives us a mean value for every channel of every pixel (a 3-tensor) and a corresponding variance for every mean. If we look at what that tells us about the red value of the pixel at coordinate (8, 7) we see that we get a univariate Gaussian with a particular mean and a variance. The mean tells us the network's best guess for that value, and the variance tells us how certain the network is about this output.



If we want to go all out, we can even make our neural network output the parameters of a Gaussian mixture model. This is called a mixture density network.

All we need to do, is make sure that we have one output node for every parameter required, and apply different activations to the different kinds of parameters. The means get a linear activation and the covariances get an exponential activation as before. The component weights need to sum to one, so we need to give them a softmax activation (over just these three output values).

If we want to train with maximum likelihood, we encounter this sum-inside-a-logarithm function again, which is difficult to deal with. But this time, it's not such a headache. As we noted in the last lecture we can work out the gradient for this loss, it's just a very ugly function. Since we are using backpropagation anyway, that needn't worry us here. All we need to work out are the local derivatives or backward functions for functions like the logarithm and the sum, and those are usually provided by systems like Pytorch and



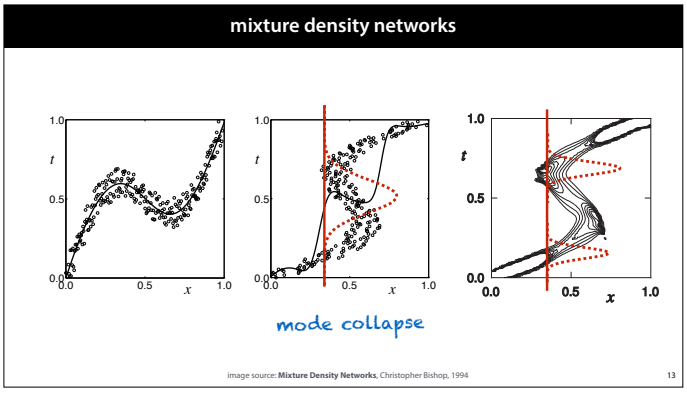
The mixture density network may seem like overkill, but it's actually very useful in regression problems where multiple answers may be valid.

Consider the problem of inverse kinematics in robotics. We have a robot arm with two joints, and we know where in space we want the hand of the arm to be. What angles should we set the two joints to? This is a great application for machine learning: it's a relatively simple, smooth function. It's easy to generate examples, and explicit solutions are a pain to write, and not robust to noise in the control of the robot. So we can solve it with a neural net.

The inputs are the two coordinates where we want the hand to be (x_1, x_2), and the outputs are the two angles we should set the joints to (θ_1, θ_2). The problem we run into, is that for every input, there are two solutions. One with the elbow up, and one with the elbow down. A normal neural network trained with an MSE loss would not pick one or the other, but it would average between the two.

A mixture density network with two components can fix this problem. For each input, it can simply put its components on the two valid solutions.

image source: Mixture Density Networks, Christopher Bishop, 1994



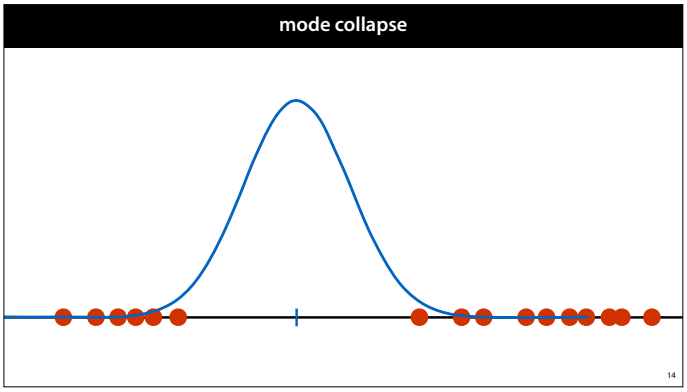
The problem with the robot arm is that the task is uniquely determined in one direction—every configuration of the robot arms leads to a unique point in space for the hand—but not when we try to reason backward from the hand to the configuration of the joints.

Here is a 2D version with that problem. Given x, we can predict t unambiguously. But, if we flip the problem and try to predict x given t, then at values like x=0.5, there are multiple predictions with high density. A network with a single Gaussian head (i.e. what we are implicitly doing when we are using least squares loss), will try to fit its Gaussian over both clusters of the data. This puts the mean, which is our ultimate prediction, between these two clusters, in a region where there is no data at all.

We can give the neural network control over the variance of this distribution as well, but all that achieves is that the variance grows to cover both groups of points. The mean stays in the same place.

By contrast, the mixture density network can output a distribution with two peaks. This allows it to cover the two groups of points in the output, and so solve the problem in a much more useful way.

The general problem in the middle picture is called mode collapse: we have a problem with multiple acceptable answers, and instead of picking one of the answers at random, the neural network averages over all of them and produces a terrible answer.

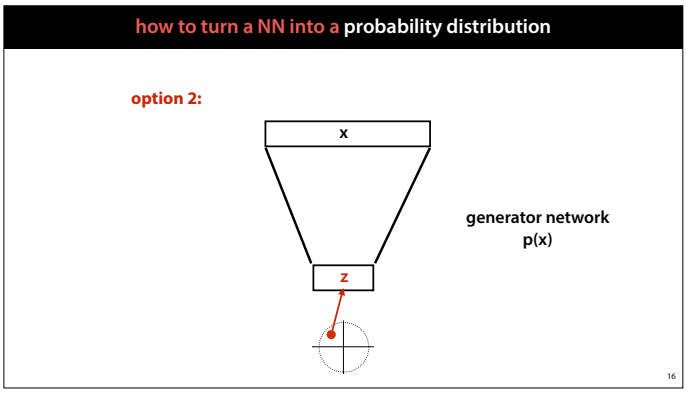


If our data is spread out in space in a complex, clustered pattern, and we fit a simple unimodal distribution to it (that is, a distribution with one peak) then the result is a distribution that puts all the probability mass on the average of our points, but very little probability mass where the points actually are.

Mixture density networks go some way towards letting us capture more complex distributions in our neural networks, but when we want to capture something as complex and rich as the distribution on images representing human faces, they're still insufficient.

A mixture model with k components gives us k modes. So in the space of images, we can pick k images to give high probability and the rest is just a simple Gaussian shape around those k points. The distribution on human faces has infinitely many modes (all possible human faces) that should all be about equally likely. To achieve a distribution this complex, we need to use the power of the neural net, not just to choose a finite set of modes, but to control the whole shape of the probability function.

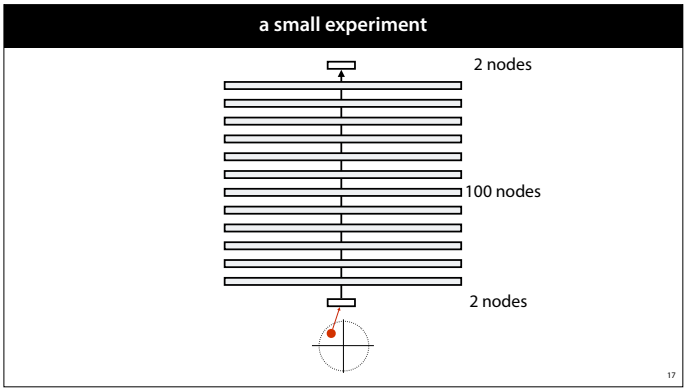
Letting the neural network pick the parameters of a distribution with a simple shape is only ever going to produce a distribution with a simple shape. We need to change our approach.



Here's a more powerful idea: we put the probability distribution at the start of the network instead of at the end of it. We sample a point from some straightforward distribution, usually a standard normal distribution, and we feed that point to a neural net. The result of these two steps is a random point, so we've defined another probability distribution. We call this construction a generator network.

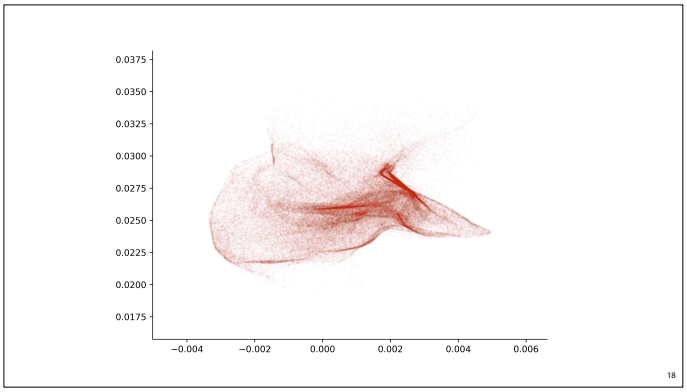
Compare this to how we defined parametrized multivariate normals in the previous lecture: we started with a standard normal distribution, and we applied a linear transformation. This is the same thing, but we've replaced the linear transformation by a nonlinear one.

If we ignore the value of the input, we are now sampling from an unconditional distribution on x.



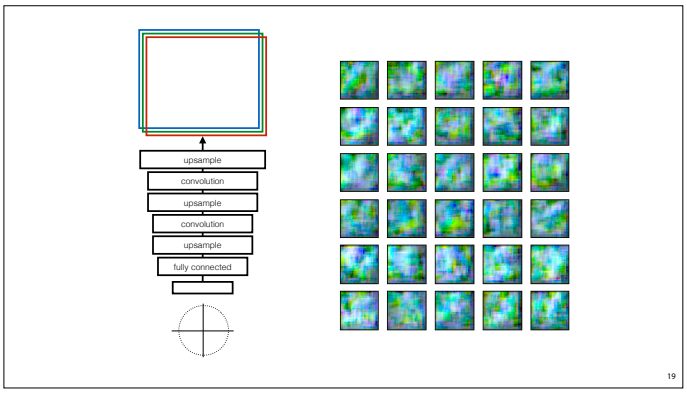
To see what kind of distributions we might get when we do this, let's try a little experiment.

We wire up a random network as shown: a two-node input layer, followed by 12, 100-node fully-connected hidden layers with ReLU activations., and a final transformation back to two points. We don't train the network. We just use Glorot initialisation to pick the parameters, and then sample some points. Since the output is 2D, we can easily scatter-plot it.



Here's a plot of 100k points sampled in this way. Clearly, we've defined a highly complex distribution. Instead of having a finite set of single points as modes, we get strands of high probability in space, and sheets of lower, but nonzero probability. Remember, this network hasn't been trained, so it's not representing anything meaningful, but it shows that the distributions we can represent in this way is a highly complex family.

Note that the variance has shrunk, and the mean has drifted away from (0, 0); apparently our weight initialisation is not quite perfect.

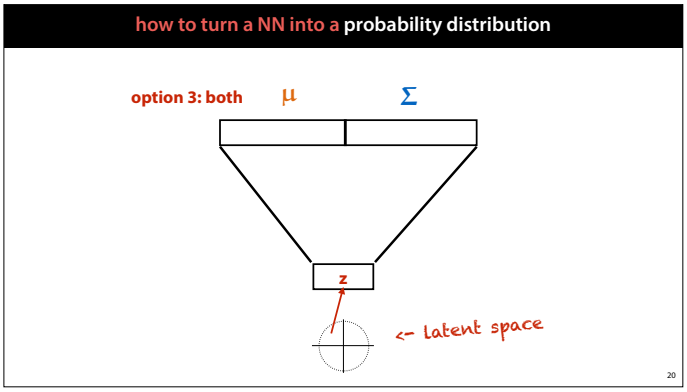


We can also use this trick to generate images. A normal convolutional net starts with a low-channel, high resolution image, and slowly decreases the resolution by maxpooling, while increasing the number of channels. Here, we reverse the process. We shape our input into a low resolution image with a large number of channels. We slowly increase the resolution by upsampling layers, and decrease the number of channels.

We can use regular convolution layers, or deconvolutions, which are a kind of upside-down convolution. Both approaches give us effective generator networks for images.

We see, that even without training, we have produced a distribution on images that is very complex and non-uniform.

I've enhanced the contrast and saturation in these images to make the colors stand out a little more.



Of course, we can also use both options: we sample the input from a standard MVN, interpret the output as another MVN, and then sample from that.

In these kinds of generator networks, the input is often called z, and the space of inputs is often called the latent space. As we will see later, this maps perfectly onto the hidden variable models of the previous lecture.

training a generator: the naive approach

loop:

Sample a random instance **x** from the data

Generate a random output **y**

Compute loss(**x**, **y**) and backpropagate

loss: mean-squared error, binary cross-entropy, L1, etc.
Anything that computes a distance between **x** and **y**.

21

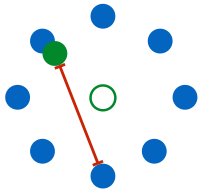
So the big question is, how do we train a generator network? Given some data, how do we set the weights of the network so that the sampled outputs start to look like the examples we have in our data?

We'll start with something that doesn't work, to help us understand why the problem is difficult. Here is a naive approach: we simply sample a random point x (e.g. a picture) from the data, and sample a point y from the model and train on how close they are.

The loss could be any distance function between two tensors. The mean-squared error over the elements is a simple approach. If the elements are values between 0 and 1 (like in images), the binary cross-entropy makes sense too. For images the absolute value of the error (also called L1 loss) is also popular.

---

mode collapse



22

If we implement this naive approach, we do not get a good probability distribution. Instead, we get mode collapse.

Here is a schematic example of what's happening: the blue points represent the modes (likely points) of the data. The green point is generated by the model. It's close to one of the blue points, so the model should be rewarded, but it's also far away from almost all of the other points. During training, there's no guarantee that we will pair it up with the correct point, and we are likely to compute the loss to a completely different point.

On average the model is punished for generating such points much more often than it is rewarded. The model that generates only the open point in the middle gets a smaller loss (and less variance). Under backpropagation, neural networks tend to converge to a distribution that generates only the open point over and over again.

In other words, the many different modes (areas of high probability) of the data distribution end up being averaged ("collapsing") into a single point.

---

mode collapse

23

Even though we have a probability distribution that is able to represent highly complex, multi-modal outputs, if we train it like this, we still end up producing a unimodal output centered on the mean of our data. If the dataset contains human faces, we get a fuzzy average of all faces, not a sample with individual details.

How do we get the the network to imagine details, instead of averaging over all possibilities?

training generator networks

Generative Adversarial Networks
Train an adversary to tell fake data from real data.

(Variational) Autoencoders
Train an encoder to tell us which data point a generated point should be compared to.

24

There are two main approaches: GANs and variational autoencoders. We'll give a quick overview of the basic principle of GANs in the next part, and then a more detailed treatment of autoencoders and variational autoencoders in the last two parts.

Deep generative models
Part 2: Generative Adversarial Networks

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

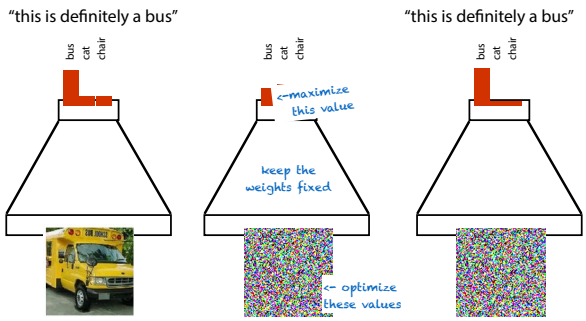
In the last video, we defined generator networks, and we saw that they represent a very rich family of probability distributions. We also saw, that training them can be a tricky business.

In this video we'll look one way of training such networks: the method of generative adversarial networks (GANs).

|section|Generative adversarial networks|
|video|https://www.youtube.com/embed/eaWxDebDDo8|

---

adversarial examples (2014)



26

GANs originated just after Convolutional networks were breaking new ground, showing spectacular, sometimes super-human, performance in image labeling. The suggestion arose that perhaps convolutional networks were doing more or less the same as what humans do when they look at something.

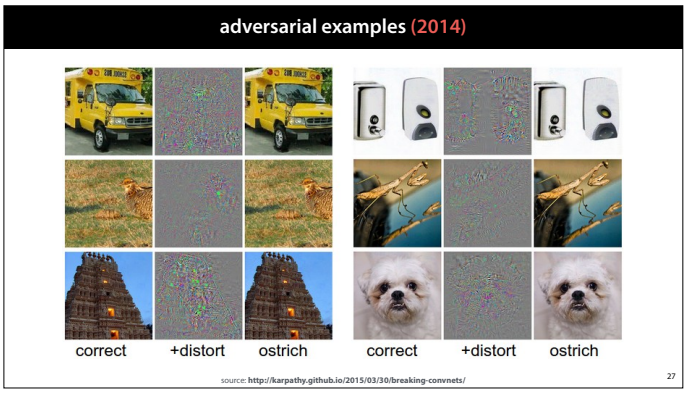
To verify this, researchers decided to start investigating what kind of inputs would make a trained convolutional network give a certain output. This is easy to do, you just compute the gradient with respect to the input of the network, and train the input to maximise the activation of a particular label, while keeping the parameters of the network fixed.

This is similar to the feature visualizing approach we saw before, but you do it on the output nodes instead of the hidden nodes.

You would expect that if you start with a random image, and follow the gradient to maximize the activation of the output node corresponding to the label "bus", you'd get a picture of

a bus. Or at least something that looks a little bit like a bus. What you actually get is something that is indistinguishable from the noise you started with. Only the very tiniest of changes is require to make the network see a bus.

These are called adversarial examples. Instances that are specifically crafted to trip up a given model.



The researchers also found that if they started the search not at a random image, but at an image of another class, all that was needed to turn it into another class (according to the network) was a very small distortion. So small, that to us the image looks unchanged. In short, a tiny change to the image is enough to make a convolutional neural net think that a picture of a bus is a picture of an ostrich.

Adversarial examples are an active area of research (both how to generate them and make models more robust against them).

Even manipulating objects in the physical world can have this effect. A stop sign can be made to look like a different traffic sign by the simple addition of some stickers. Clearly, this has some worrying implications for the development of self-driving cars.

generative adversarial networks

loop:

train a classifier to tell X_{Pos} from X_{Neg}

Generate adversarial examples
Clearly not Pos, but the classifier thinks so anyway

Add the adversarial examples to X_{Neg}

The classifier (discriminator) gets more *robust*, the generator gets more *realistic*.

29

Pretty soon, this bad news was turned into good news by realising that if you can generate adversarial examples automatically, you can also add them to the dataset as negatives and retrain your network to make it more robust. You can simply tell your network that these things are not stop signs. Then, once your network is robust to the original adversarial examples, you can generate some new adversarial examples, and start the whole thing over again.

We can think of this as a kind of iterated 2 player game (or an arms race). The discriminator (our classifier) tries to get good enough to tell fake data from real data and the generator (the algorithm that generates the adversarial examples) tries to get good enough to fool the discriminator.

This is the basic idea of the generative adversarial network.

GANs

- Vanilla GANs
- Conditional GANs
- CycleGAN
- StyleGAN

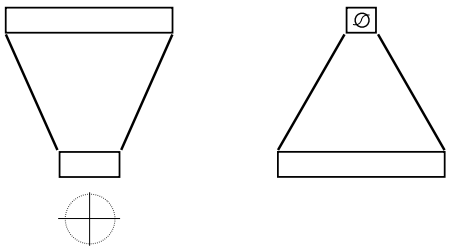

30

We'll look at four different examples of GANs. We'll call the basic approach the "vanilla GAN"

---

end-to-end GANs

generator: G **discriminator: D**



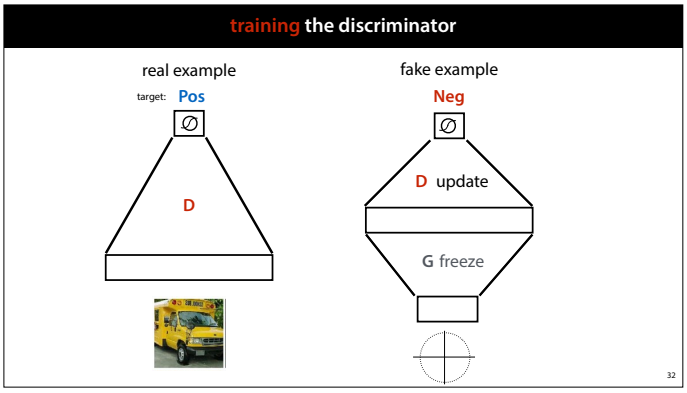
31

Generating adversarial examples by gradient descent is possible, but it's much nicer if both our generator and our discriminator are separate neural networks. This will lead to a much cleaner approach for training GANs.

We will draw the two components like this. The generator takes an input sampled from a standard MVN and produces an image. This is a generator network as described in the previous video. We don't give it an output distribution (i.e. we're using option 2 from the previous part).

The discriminator takes an image and classifies it as Positive (a real image of the target class) or Negative (a fake image sampled from the generator).

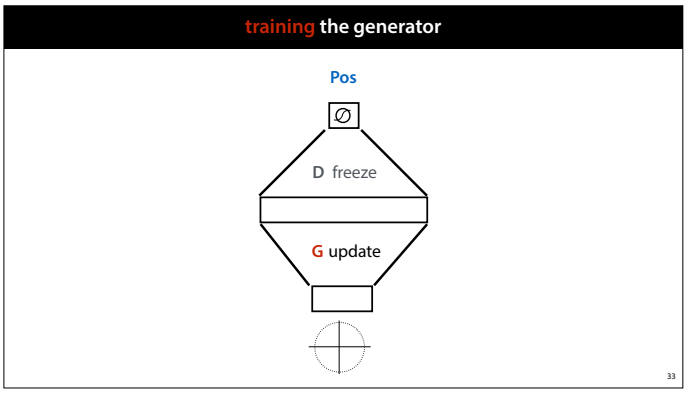
If we have other images that are not of the target class, we can add those to the negative examples as well, but often, the positive class is just our dataset (like a collection of human faces), and the negative class is just the fake images created by the generator.



To train the discriminator, we feed it examples from the positive class, and train it to classify these as Pos.

We also sample images from the generator (whose weights we keep fixed) and train the discriminator to recognize these as negative. At first, these will just be random noise, but there's little harm in telling our network that such images are not busses (or whatever our positive class is).

Note that since the generator is a neural network, we don't need to collect a dataset of fake images which we then feed to the discriminator. We can just stick the discriminator on top of the generator, making a single computation graph, and train it by gradient descent to classify the result as negative. We just need to make sure to freeze the weights of the generator, so that gradient descent only updates the discriminator.

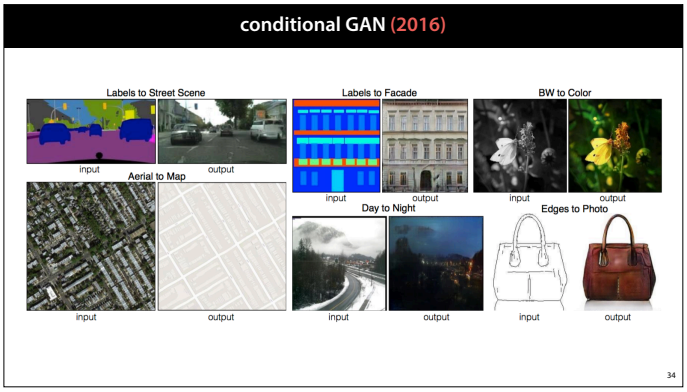


Then, to train the generator, we freeze the discriminator and train the weights of the generator to produce images that cause the discriminator to label them as Positive.

This step may take a little time to wrap your head around. If it helps, think of the whole disciminator as a very complicated loss function. Whatever the generator produces, the more likely the discriminator is to call it positive, the lower the loss.

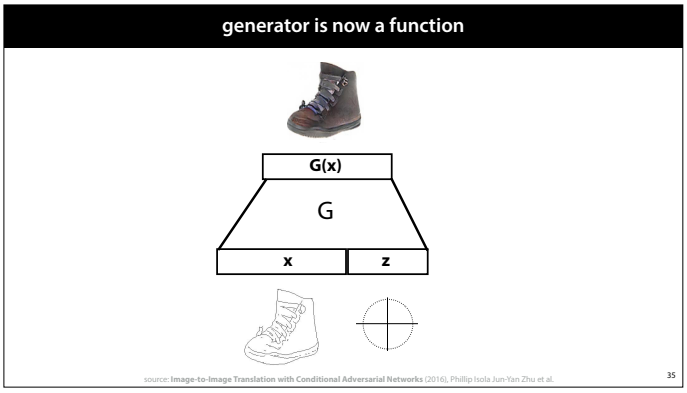
We don't need to wait for either step to converge. We can just train a the discriminator for one batch (i.e. one step of gradient descent) and then train the generator for one batch, and so on.

And this is what we'll call the vanilla GAN.



Sometimes we want to train the network to map an input to an output, but to generate the output probabilistically. For instance, when we train a network to color in a black-and-white photograph of a flower, it could choose many colors for the flower. We want to avoid mode collapse here: instead of averaging over all possible colors, giving us a brown or gray flower, we want it to pick one color, from all the possibilities.

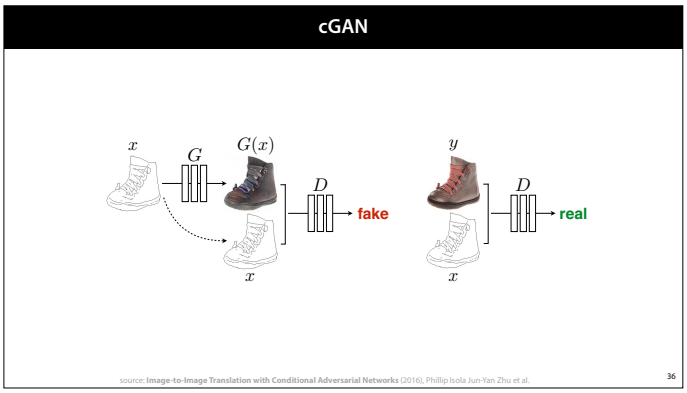
A conditional GAN lets us train generator networks that can do this.



In a conditional GAN, the generator is a function with an image input, which it maps it to an image output. However, it uses randomness to imagine specific details in the output.

In this example, it imagines the photograph corresponding to a line drawing of a shoe. Running this generator twice would result in different shoes that are both "correct" instantiations of the input line drawing.

source: Image-to-Image Translation with Conditional Adversarial Networks (2016), Phillip Isola Jun-Yan Zhu et al.

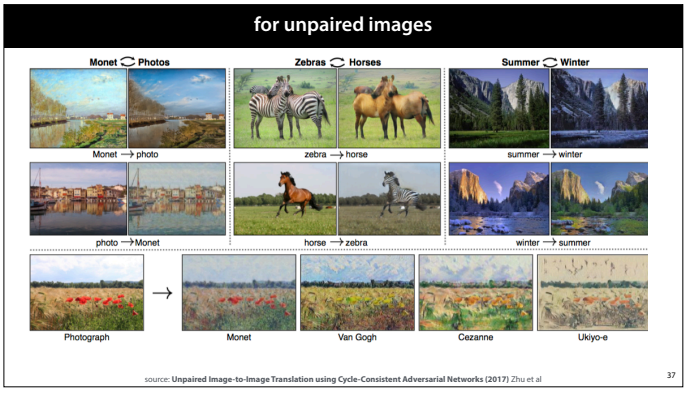


To train a conditional GAN, we give the discriminator pairs of inputs and outputs. If these come from the generator, they should be classified as fake (negative) and if they come from the data, they should be classified as real (positive).

The generator is trained in two ways.

1. We freeze the weights of the discriminator, as before, and train the generator to produce thins that the discriminator will think are real.
2. We feed it and input from the data, and backpropagate on the corresponding output (using L1 loss).

There are many more details you need to be aware of to train a model like this well, but for now we will just focus on the high-level picture.



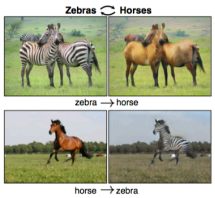
The conditional GAN works really well, but only if we have an example of a specific output that corresponds to a specific input. For some tasks, we don't have paired images. We only have unmatched bags of images in two domains. For instance, we know that a picture of a horse can be turned into a picture of that horse as a zebra (a skilled painter could easily do this), but we don't have a lot of paired images of horses and corresponding zebras. All we have is a large number of horse images and a large number of zebra images.

If we randomly match one zebra image to a horse image, and train a conditional GAN on this, all we get is mode collapse.

CycleGAN (2017)

Transformations are always learned both ways. Ingredients:

- Horse discriminator
- Zebra discriminator
- Horse-to-zebra generator (G)
- Zebra-to-horse generator (F)



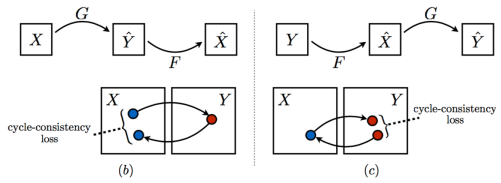
38

CycleGANs solve this problem using two tricks.

First, we train generators to perform the transformation in both directions. We train both a horse-to-zebra generator and a zebra-to-horse generator. Then each horse in our dataset is transformed into a zebra and back again.This gives us a fake zebra picture, which we can use to train a zebra discriminator, together with the real zebra pictures. We do the same thing the other way around: we transform the zebras to fake horses and back again, and use the fake horses together with the real horses to train a horse discriminator.

Second, we add a cycle consistency loss. When we transform a horse to a zebra and back again, we should end up with the same horse again. The more different the final horse picture is from the original, the more we punish the generator networks.

CycleGAN (2017)



Think of the generators as practicing *steganography*.
I.e. hiding a horse picture inside a zebra picture.

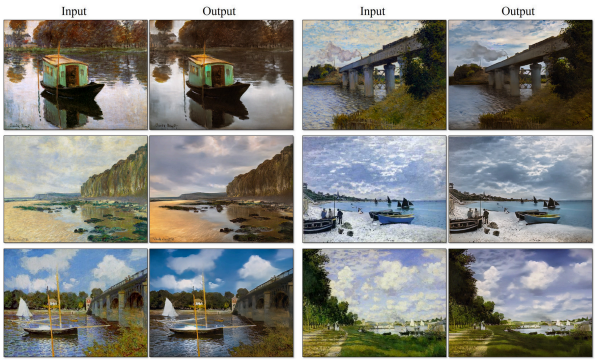
source: **Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (2017)** Zhu et al

39

Here is the whole process in a diagram.

One way to think of this is as the generators practicing steganography: hiding a secret message inside another innocent message. The generators are trying to hide a picture of a horse inside a picture of a zebra. The cycle consistency loss ensures that all the information of the horse picture is in the zebra picture and the horse picture can be fully decoded from the zebra picture. The discriminator's job is to tell which of the zebra pictures it sees have a horse hiding in it.

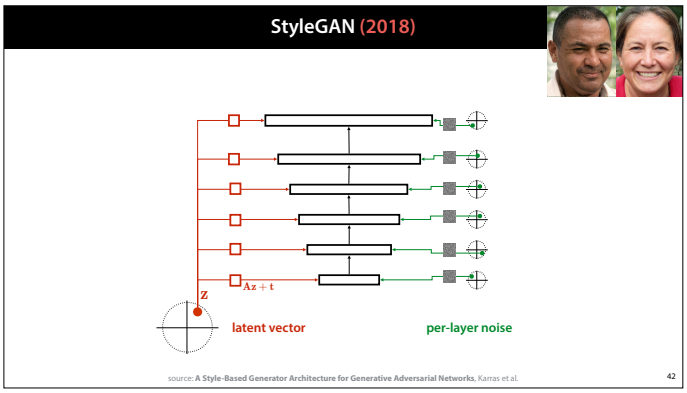
If we have a strong discriminator and the generator can still fool it, then we get very realistic zebra pictures with horses hidden inside. Since the obvious way to make this transformation is to transform the horse into the zebra in the way we would do it, this is the transformation that the network learns.



40

The CycleGAN works surprisingly well. Here's how it maps photographs to impressionist paintings and vice versa.

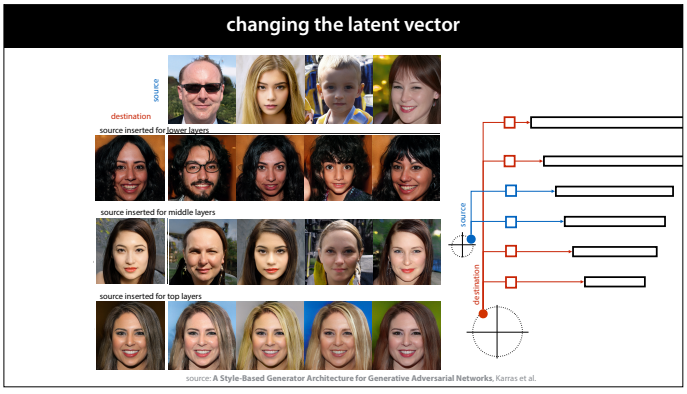
It doesn't always work perfectly, though.



Finally, let's take a look at the StyleGAN, the network that generated the faces we first saw in the introduction. This is basically a Vanilla GAN, with most of the special tricks in the way the generator is constructed. It uses too many tricks to discuss here in detail, so we'll just focus on one aspect: the idea that the latent vector is fed to the network at each stage of its forward pass.

Since an image generator starts with a coarse (low resolution), high level description of an image, and slowly fills in the details, feeding it the latent vector at every layer (transformed by an affine transformation to fit it to the shape of the data at that stage), allows it to use different parts of the latent vector to describe different aspects of the image (the authors call these "styles").

The network also receives separate extra random noise per layer, that allows it to make random choices. Without this, all randomness would have to come from the latent vector.



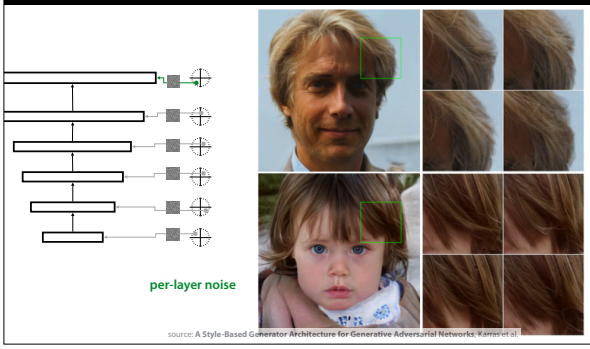
To see how this works, we can try to manipulate the network, by changing the latent vector to another for some of the layers. In this example all images on the margins are people that are generated for a particular single latent vector.

We then re-generate the image for the destination, except that for a few layers (at the bottom, middle or top), we use the source latent vector instead.

As we see, overriding the bottom layers changes things like gender, age and hair length, but not ethnicity. For the middle layer, the age is largely taken from the destination image, but the ethnicity is now override by the source. Finally for the top layers, only surface details are changed.

This kind of manipulation was done during training as well, to ensure that it would lead to faces that fool the discriminator.

changing the noise



source: A Style-Based Generator Architecture for Generative Adversarial Networks, Karras et al

44

Let's look at the other side of the network: the noise inputs.

If we keep all the latent and noise inputs the same except for the very last noise input, we can see what the noise achieves: the man's hair is equally messy in each generated example, but exactly in what way it's messy changes per sample. The network uses the noise to determine the precise orientation of the individual "hairs".

GANs: not discussed

- Wasserstein distance
- Evaluation: Inception score, FID score
- Batch normalisation
- Relativistic GANs

45

We've given you a high level overview of GANs, which will hopefully give you an intuitive grasp of how they work. However, GANs are notoriously difficult to train, and many other tricks are required to get them to work. Here are some phrases you should Google if you decide to try implementing your own GAN.

In the next video, we'll look at a completely different approach to training generator networks: autoencoders.

Deep generative models
Part 3: Autoencoders

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this part, we'll start to lay the groundwork for Variational Autoencoders. This starts with a completely different abstract task: dimensionality reduction. We'll see that given a dimensionality reduction model, we can often turn it into a generative model with a few hacks. In the next part, we will then develop this type of model in a more grounded and theoretical way.

|section|Autoencoders|
|video|https://www.youtube.com/embed/t6GxDo1fSt0|

What can we do with a generator?

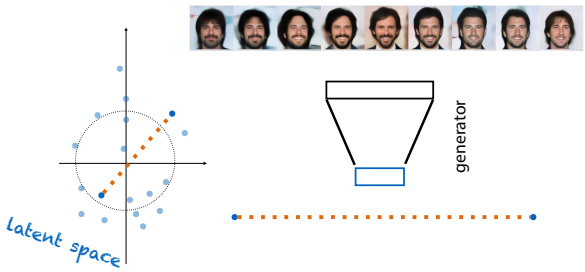
- Generate "new" data
- Interpolation
- Data manipulation
- Dimensionality reduction

47

Before we turn to autoencoders, let's first look at what we can do once we've trained a generator network. We'll look at four use cases.

The first, of course is that we can generate data that looks like it came from the same distribution as ours.

interpolation



source: **Sampling Generative Networks**, Tom White

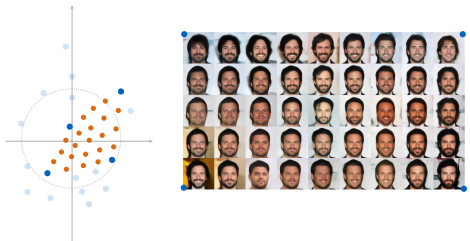
48

Another thing we can do is interpolation.

If we take two points in the input space, and draw a line between them, we can pick evenly spaced points on that line and decode them. If the generator is good, this should give us a smooth transition from one point to the other, and each point should result in a convincing example of our output domain.

Remember that in some contexts, we refer to the input of a generator network as its latent space.

interpolation grid

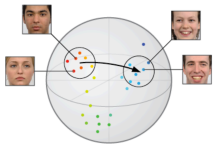


source: **Sampling Generative Networks**, Tom White

49

We can also draw an interpolation grid; we just map the corners of a square lattice of equally spaced points to four points in our input space, and run all points through the generator network.

spherical linear interpolation



source: **Sampling Generative Networks**, Tom White

50

If the latent space is high dimensional, most of the probability of the standard MVN is near the edges of the radius-1 hypersphere (not in the centre as it is in 1, 2 and 3-dimensional MVNs).

High-dimensional MVNs look more like a soap bubble than the dense pointcloud we're used to seeing in low-dimensional visualizations.

For that reason, we get better results if we interpolate along an arcinstead of along a straight line. This is called spherical linear interpolation.

interpolation on real data

source: Abdal, Rameen, Yipeng Qin, and Peter Wonka. "Image2stylegan: How to embed images into the stylegan latent space?." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019.

51

What if we want to interpolate between points in our dataset? It's possible to do this with a GAN trained generator, but to make this work, we first have to find our data points in the input spoace. Remember, during training the discriminator is the only network that gets to see the actual data. We never explicitly map the data to the latent space.

We can tack a mapping from data to latent space onto the network after training (as was done for these images), but we can also learn such a mapping directly. As it happens, this can help us to train the generator in a much more direct way.

What can we do with a generator?

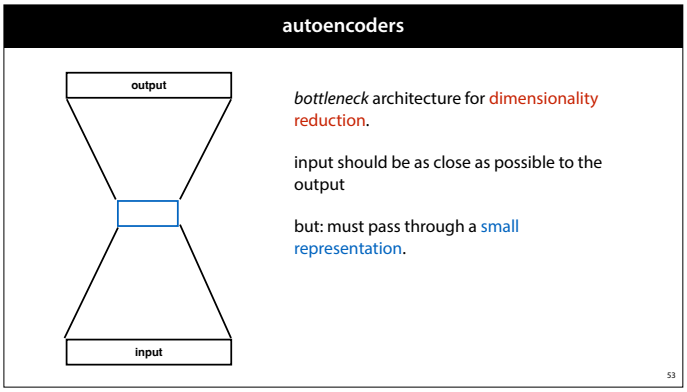
- Generative modelling
- Interpolation
- Data manipulation
  autoencoders only
- Dimensionality reduction
  autoencoders only

requires mapping into the latent space

52

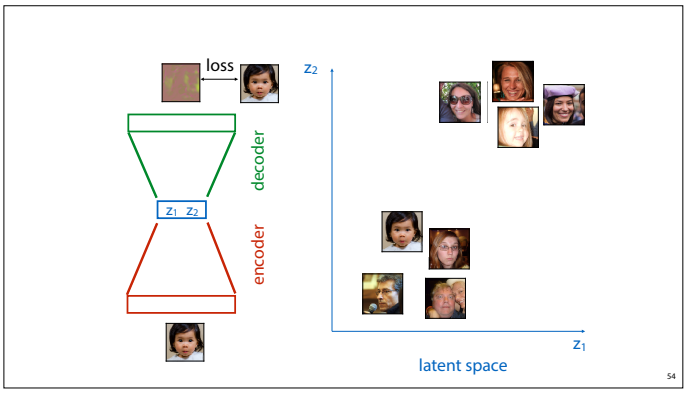
Note that such a mapping would also give us a dimensionality reduction. We can see the latent space representation of the data as a reduced dimensionality representation of the input.

We'll focus on the perspective of dimensionality reduction for the rest of this video, to set up basic autoencoders. We can get a generator network out of these, but it's a bit of an afterthought. In the next video, we'll see how to train generator networks with a data-to-latent-space mapping in a more principled way.



Here's what a simple autoencoder looks like. It's is a particular type of neural network, shaped like an hourglass. Its job is just to make the output as close to the input as possible, but somewhere in the network there is a small layer that functions as a bottleneck.

After the network is trained, this small layer becomes a compressed low-dimensional representation of the input.

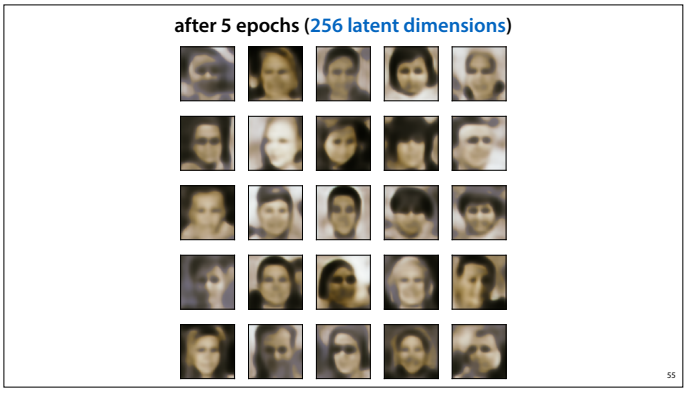


Here's the picture in detail. We call the bottom half of the network the encoder and the top half the decoder. We feed the autoencoder an instance from our dataset, and all it has to do is reproduce that instance in its output. We can use any loss that compares the output to the original input, and produces a lower loss, the more similar they are. Then, we just brackpropagate the loss and train by gradient descent.

Least-squares loss, absolute error loss and binary cross-entropy are popular choices

We call the blue layer the latent representation of the input. If we train an autoencoder with just two nodes on the latent representation, we can plot what latent representation each input is assigned. If the autoencoder works well, we expect to see similar images clustered together (for instance smiling people vs frowning people, men vs women, etc).

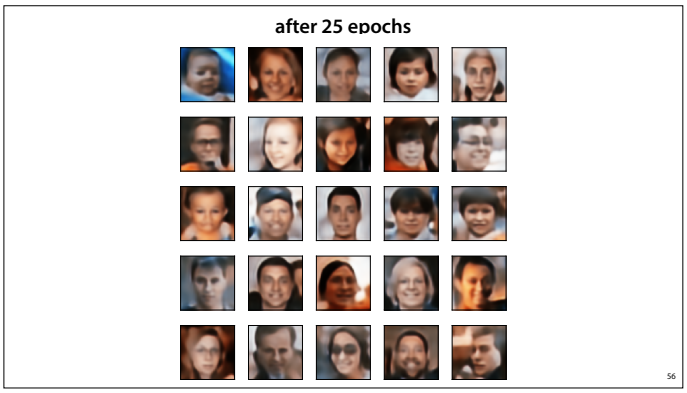
In a 2D space, we can't cluster too many attributes together, but in higher dimensions it's easier. To quote Geoff Hinton: "If there was a 30 dimensional supermarket, [the anchovies] could be close to the pizza toppings and close to the sardines."

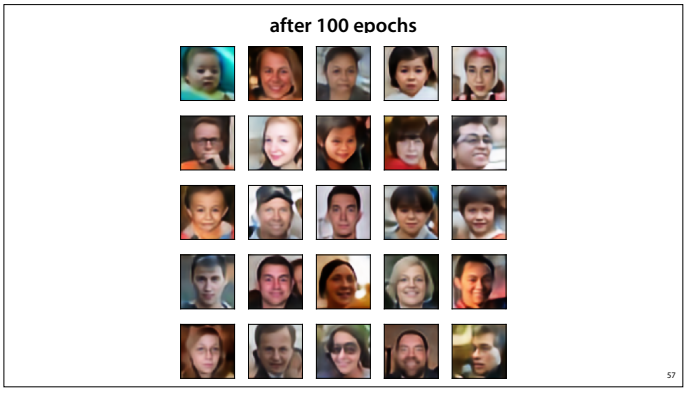


To show what this looks like, we've set up a relatively simple autoencoder consisting of convolutions in the encoder and deconvolutions in the decoder. We train it on a low-res version of the FFHQ dataset of human faces. We give the latent space 256 dimensions.

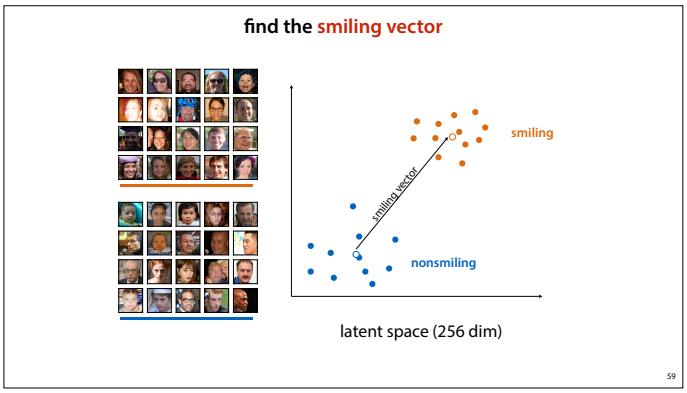
For the PCA lecture we used black and white data to make the task easier. Since we have a more powerful model, we can use more varied full-color data. The StyleGAN was trained on the full-resolution version of this data.

Here are the reconstructions on a very simple network, with MSE loss on the output after 5 full passes over the data.





After 300 epochs, the autoencoder has pretty much converged. Here are the reconstructions next to the original data. Considering that we've reduced each image to just 256 numbers, it's not too bad.

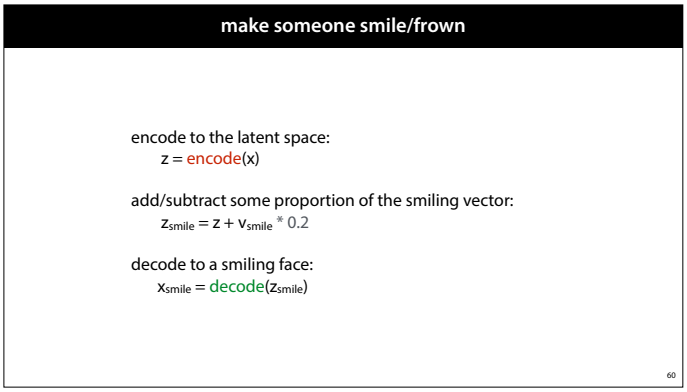


One thing we can now do is to study the latent space based on the examples that we have. For instance, we can see whether smiling and non-smiling people end up in distinct parts of the latent space.

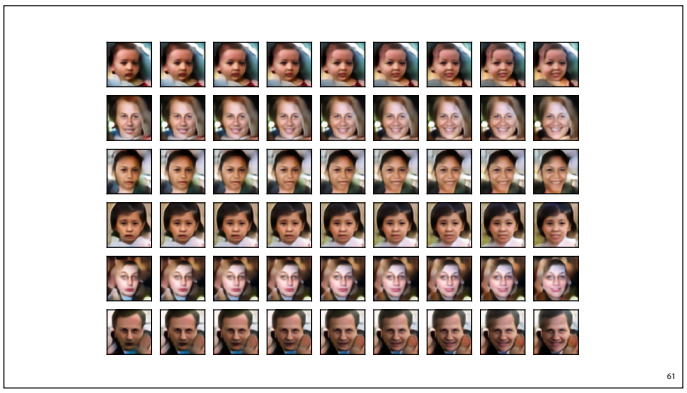
We just label a small amount of instances as smiling and nonsmiling (just 20 each in this case). If we're lucky, these form distinct clusters in our latent space. If we compute the means of these clusters, we can draw a vector between them. We can think of this as a "smiling" vector. The further we push people along this line, the more the decoded point will smile.

This is one big benefit of autoencoders: we can train them on unlabeled data (which is cheap) and then use only a very small number of labeled examples to "annotate" the latent space. In other words, autoencoders are a great way to do semi-supervised learning.

Compare this to what we did in our previous dimensionality reduction method of principal component analysis. There, we found that dimensions in the reduced space corresponded to high-level semantic concepts like gender and expression. Here, the latent space is a little more "entangled", but we can usually still find distinct directions for high-level concepts (they are just not aligned with the axes).



Once we've worked out what the smiling vector is, we can manipulate photographs to make people smile. We just encode their picture into the latent space, add the smiling vector (times some small scalar to control the effect), and decode the manipulated latent representation. If the autoencoder understands "smiling" well enough, the result will be the same picture but manipulated so that the person will smile.



Here is what that looks like for our (simple) example model. In the middle we have the decoding of the original data, and to the right we see what happens if we add an increasingly large multiple of the smiling vector.

To the right we subtract the smiling vector, which makes the person frown.

With a bit more powerful model, and some face detection, we can see what some famously moody celebrities might look like if they smiled.

source: https://blogs.nvidia.com/blog/2016/12/23/ai-flips-kanye-wests-frown-upside-down/

autoencoders

Keep the encoder and decoder: data manipulator.

Keep the encoder, ditch the decoder: dimensionality reduction.

Ditch the encoder, keep the decoder: **generator network**.

63

What we get out of an autoencoder, depends on which part of the model we focus on.

If we keep the encoder and the decoder, we get a network that can help us manipulate data in this way.

If we keep just the encoder, we get a powerful dimensionality reduction method. We can use the latent space representation as the features for a model that does not scale well to too many features (like a non-naive Bayesian classifier).

But this lecture was about generator networks. How do we get a generator out of a trained autoencoder? It turns out we can do this by keeping just the decoder.

---

turning an autoencoder into a generator

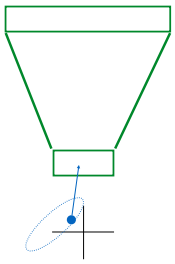
train an autoencoder

encode the data to latent variables Z

fit an MVN to Z

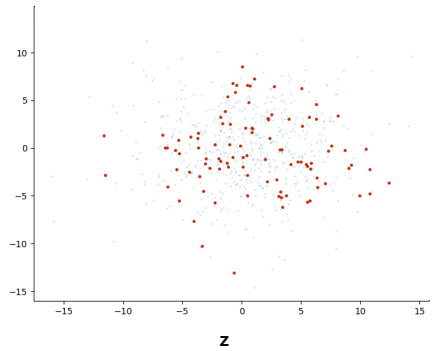
sample from the MVN

"decode" the sample



64

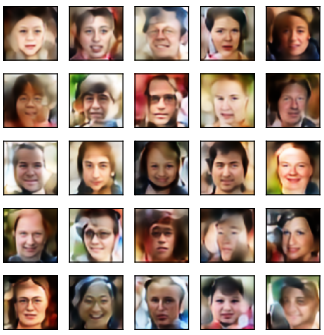
We don't know beforehand where the data will end up in latent space, but after training we can just check. We encode the training data, fit a distribution to this point cloud in our latent space, and then just use this distribution as the input to our decoder to create a generator network.



65

This is the point cloud of the latent representations in our example. We plot the first two of the 256 dimensions, resulting in the blue point cloud.

To these points we, we fit an MVN (in 256 dimensions), and we sample 400 new points from it, the red dots.



generated

66

If we feed these points to the decoder, this is what we get. It's not quite up these with the style gan results, but clearly, the model can generate some non-existant people.

How to control the shape of the latent space?

What are we optimizing? Can we optimize maximum likelihood directly?

Can we optimize for better interpolation directly?

67

This has given us a generator, but we have little control over what the latent space looks like. We just have to hope that it looks enough like a normal distribution that our MVN makes a good fit. In the GAN, we have perfect control over what our distribution on the latent space looks like; we can freely set it to anything. However, there, we have to fit a mapping from data to latent space after the fact.

We've also seen that this interpolation works well, but it's not something we've specifically trained the network to do. In the GAN, we should expect all latent space points to decode to something that fools the decoder, but in the autoencoder, there is nothing that stops the points in between the data points from decoding to garbage.

Moreover, neither the GAN nor the autoencoder is a very principled way of optimizing. Is there a way to train for maximum likelihood directly?

The answer to all of these questions is the variational autoencoder, which we'll discuss in the next video.

Deep generative models
Part 4: Variational autoencoders

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

The video refers to a lecture that is no longer part of the course. See the lecture notes for the missing pieces. We'll create a new video soon.

|section|Variational autoencoders|
|video|https://www.youtube.com/embed/inUJd7f931g|

variational autoencoders

- Force the decoder to also decode points *near* z correctly
- Forces the latent distribution of the data towards N(**0**, **I**)
- Can be derived from first principles
  maximum likelihood

69

The variational autoencoder is a more principled way to train a generator network using the principles of an autoencoder. This requires a little more math, but we get a few benefits in return.

---

maximum (log) likelihood objective

$$\arg\max_{\theta} \quad \ln p_\theta(x)$$

70

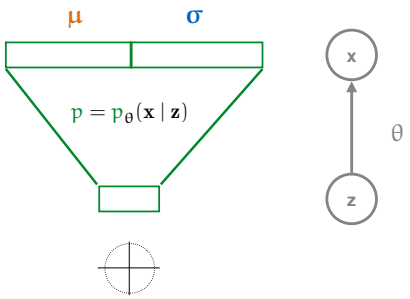
We'll start with the maximum log-likelihood objective. We want to choose our parameters θ (the weights of the neural network) to maximise the log likelihood of the data. We will write this objective step by step until we end up with an autoencoder.

To simplify our notation a little bit, we will put the weights of the network in the subscript of p rather than in the conditional p(x | θ). Both notations mean the same thing (the conditional notation is useful when you want to apply Bayes' rule to get a distribution on the parameters θ, but we won't go into that in this lecture.

We're specifically using the natural logarithm ln here because it will simplify things a little down the road.

---

hidden variable model



71

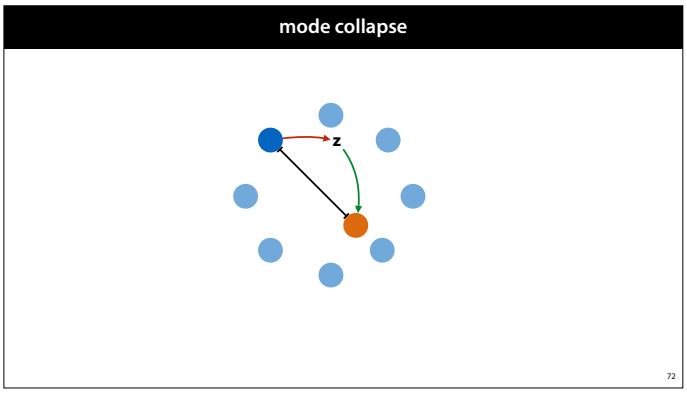
The first insight is that we can view our generator as a hidden variable model. We have a hidden variable z, a standard normally distributed vector, which we then fed to a neural network. The network produces a spherical normal distribution N(μ, σ), from which we sample variable x, which we then observe. We assume that the data came from this process too (or something equivalent to it), and we want to choose the parameters θ of the neural network to mimic the process that generated the data as closely as possible.

The network computes the conditional distribution of x given z: $p_\theta(z|\ x)$. Note that the value we actually want to maximize, $p_\theta(x)$ is not conditioned on z. We want to maximize the probability of x, regardless of what latent vector z generated it. While $p_\theta(z|\ x)$ is easy to compute (by just running the network) $p_\theta(x)$ is not.

To compute $p_\theta(x)$, we would have to somehow integrate over all possible values of z and their prior probabilities.

Note also that we're not thinking of this as an autoencoder

yet. That view will emerge. For now, we are just looking at a generator network, and wondering how we might choose its parameters so that we maximize the likelihood of the data.



Here we see how the hidden variable problem causes our mode collapse. If we knew which z was supposed to produce which x, we could feed that z to the network to compute the loss between the output and x and optimize by backpropagation and gradient descent. In short, we'd have both the input (z) and the target value (x) for our network.

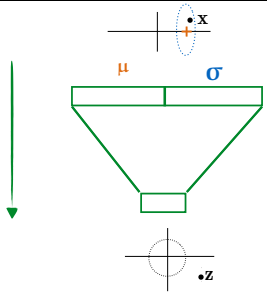
The problem is that we don't have the "complete" data (given the assumptions we made about how the data were generated). We don't know the values of z, only the values of x. This means we don't have the inputs for our network, only the targets.

A common way to solve problems of incomplete data is to generate an approximate completion of your data. That is, you guess the missing part (in this case z), and learn your generator for x on the basis of this guess. That is, you build a generator for x given z, and a guesser for z given x. Then you train both together. The better the guesser gets, the better the generator gets and vice versa.

This is how we will build our autoencoder.

It's no longer part of the exam material, but the Eexpectation Maximization (EM) algorithm works in exactly the same way. Have a look at the extra resources if you're curious. Looking at the differences and commonalities between EM and the VAE is a great way to really get to grips with the details of probabilistic programming.

inverting p?



$$p_\theta(\mathbf{z} \mid \mathbf{x})$$

73

One way to solve this problem, would be to figure out which latent variable z would be likely to generate a given x. That is, for a given generator network, with fixed parameters θ, if we are given an image x how likely is any given input z to lead to the output sample x? This is the function $p_\theta(z|\, x)$.

This, again, is not an easy thing to work out. To compute $p(z|x, \theta)$ we would need to invert the neural network: work out for a given output x, what the input z was. Or more probabilistically: which input values z are likely to have caused the network to output x.

The picture shows the Bayesian reasoning that is required: this particular z leads to an output distribution $N(\mu, \sigma)$ that has its peak probability density very close to x. This suggests z should have a high likelyhood, but z itself is very unlikely under the prior probability N(0, I), from which we sample z.

Network inversion is not impossible to do (we saw something similar when we were discussing GANs), but it's a costly and imprecise business. Just like we did with the GANs, it's best to introduce a network that will learn the inversion for us. We call this network q.

Unlike the GAN setting where we alternate the training of a network and its inversion, we'll figure out a way to train p and q together. We'll try to update the parameters of p to fit the data, and try we'll update the parameters of q to keep it a good approximation to the inversion of p, and we'll do both at the same time.

---

introducing q

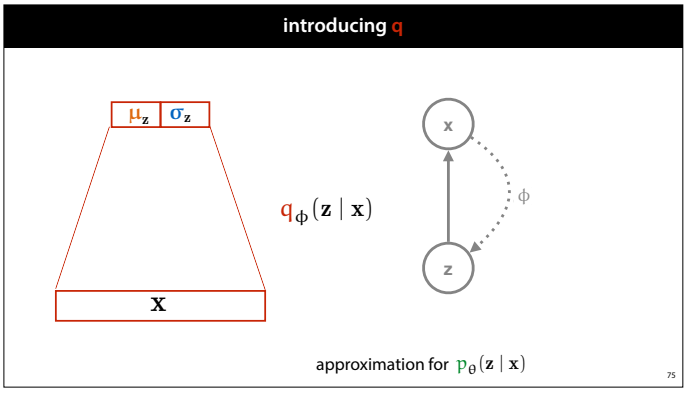
$$q_\phi(\mathbf{z} \mid \mathbf{x}) \approx p_\theta(\mathbf{z} \mid \mathbf{x})$$

74

For this purpose, we'll introduce q. Its function is to approximate $p_\theta(z|\, x)$, the inversion of our generator network.

Note that we're treating q as an approximation, but we're not yet saying that it's a good approximation. It could be a terrible approximation, for instance at the start of learning, but we will set up some equations that hold for any approximation q no matter how good or bad.

Since we will implement q with a neural network, it will have parameters, just like p. We will refer to the set of all its parameters with the letter ϕ (phi).



We will draw the neural network q like this.

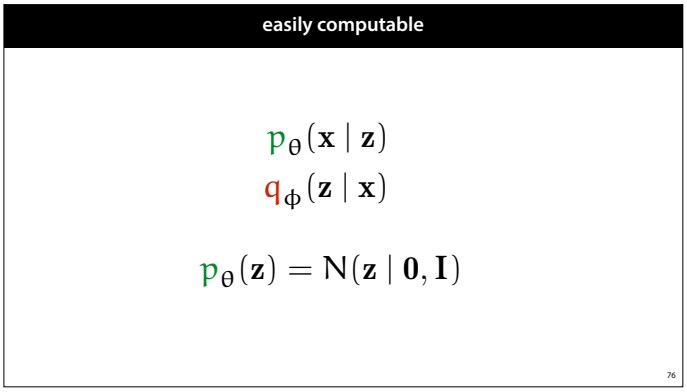
It maps a given instance x to a distribution on the latent space. That is, we're not generating a single z that is the likely latent for a given x, we are creating a distribution that will tell us for every z how likely it is that z would produce the given x.

To make things easy for ourselves, we will assume that this distribution can be well approximated by a single spherical normal distribution. This means that q is a network with a probabilistic output, just like p.

If the network works well, the correct latent representation (the one that decodes to x if we feed it to the generator) will get a high probability density under the distribution produced by this network.

The structure of the q network, does not need to be related in any way to the structure of the network p, and their parameters ϕ and θ are not tied together in any way.

In practice, it's common to make q the rough inverse architecture of p, using deconvolutions in q where p uses convolutions, and so on, but this is not necessarily the best approach.



With that, we can state our problem more precisely. We have a generator network p that allows us to easily compute the conditional probability of x given some z. We want to choose its parameters so that the probability density of the data is maximized.

We have an encoder network q that allows us to easily compute the conditional probability of z given some x. We want to choose its parameters so that the probabilities it generates correspond as closely as possible to the conditional probability z given x under the current weights of the generator network.

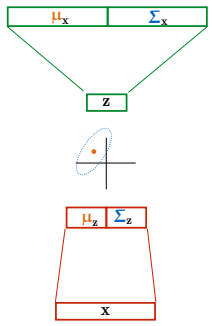
Finally, there is one more function that we can easily compute: the marginal $p_\theta(z)$. This is simply because we defined it ourselves: the input to p is sampled from a standard normal distribution. It's value is chosen completely independently of how p functions or what x it produces, so the marginal distribution on z is simply the standard normal distribution.

This, then, is our challenge: tune the weights of these two

networks to maximize the probability of the data under the generator, and do it in such a way that we only ever need to compute these three functions.

$p_\theta(\mathbf{x} \mid \mathbf{z})$

$q_\phi(\mathbf{z} \mid \mathbf{x})$



77

Putting everything together, this is our model. If we feed q an instance from our data x, we get a normal distribution on the latent space. If we sample a point z from this distribution, and feed it to p we get a distribution on x. If the networks are both well trained, this should give us a good reconstruction of x.

The neural network p is our probability distribution conditional on the latent vector. q is our approximation of the conditional distribution on z.

We are beginning to see the autoencoder emerge. Note that this is relatively incidental. We are just trying to train the generator network, and all we've done is introduce q as an approximation to the inverse of p. At no point did we set out to build a dimensionality reduction method.

$$\arg\max_\theta \quad \ln p_\theta(x)$$

78

Now, we promised that we could optimize these two networks together in a principled way. We will start with the maximum likelihood objective, which is hard to optimize directly, and rewrite step by step into a loss function that affects both networks.

The maximum likelihood loss doesn't involve q yet. But we'll rewrite it later to include q.

a very useful decomposition

$$\ln p(\mathbf{x}) = L(q, p) + KL(q, p)$$

with :

$p = p(\mathbf{z} \mid \mathbf{x})$

$q(\mathbf{z} \mid \mathbf{x})$ *any* approximation to $p(\mathbf{z} \mid \mathbf{x})$

$KL(q, p)$ KL divergence between $p(\mathbf{z} \mid \mathbf{x})$ and $q(\mathbf{z} \mid \mathbf{x})$

$$L(q, p) = \mathbb{E}_q \ln \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})}$$

79

We will first show the following decomposition. This is a very useful property, and it's used often to deal with hidden variable models.

We've dropped the parameter subscripts to simplify the notation.

The idea is that when we introduce an approximation to q(z|x) for p(z|x), we can then look at the Kulback-Leibler (KL) divergence between the two that expresses how good the approximation is. The smaller the KL divergence is, the better the approximation. If the KL divergence is zero, then the approximation is perfect and q(z|x) and p(z|x) express the same function.

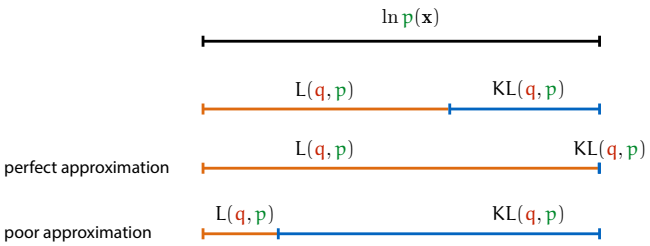
It turns out that the sum of the KL divergence and the function L as defined in the slide, is the log-probability of x.

Note that the components of L are exactly the three functions we can compute. We still need to deal with the fact that it's an expectation, but this hopefully shows that we're getting closer to our target.

---

the role of L

$$\ln p(\mathbf{x}) = L(q, p) + KL(q, p)$$



80

If we take p and its parameters as given, we can use the following reasoning to understand what the function L means. We know that given p, the log-likelihood of the data p(x) is fixed. We also know that the KL divergence is always non-negative. This means that for any q, the KL term must be smaller than the log-likelihood, and the L term is what makes up the difference.

This means that if q is a perfect approximation, L is equal to the log likelihood, the very thing we wanted to approximate. This is relevant, because, as we will show, L

---

what's our loss?

$$\ln p(\mathbf{x}) = L(q, p) + KL(q, p)$$

variational lower bound
or evidence lower bound (ELBO)

$$L(q, p) = \mathbb{E}_q \ln \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})}$$

81

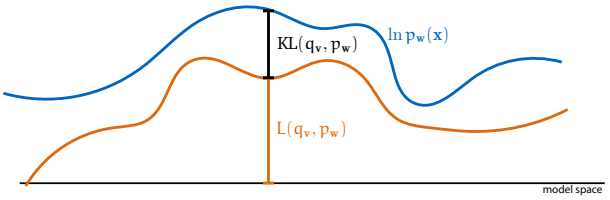
The idea of the variational autoencoder is to take the L term, and use this as our loss.

The thinking is that since it's a lowerbound on the likelihood (the quantity we're trying to maximize), anything that increases L will also increase our likelihood. The better we maximise L, the better our model will do.

Note that this is only a lowerbound because we know that the KL term cannot be negative. If the KL divergence could be negative, the L term could be larger or smaller than the log-likelihood.

lower bound objective

$$\ln p_w(x) = L(q_v, p_w) + KL(q_v, p_w)$$



82

Here's a visualisation of how a lower bound objective works. We're interested in finding the highest point of the blue line (the maximum likelihood solution), but that's difficult to compute. Instead, we maximise the orange line (the evidence lower bound). Because it's guaranteed to be below the blue line everywhere, we may expect to be finding a high value for the blue line as well. To some extent, pushing up the orange line, pushes up the blue line as well.

How well we do on the blue line depends a lot on how tight the lower bound is. The distance between the lower bound and the log likelihood is expressed by the KL divergence between $p_w(z|x)$ and $q_v(z|x)$. That is, because we cannot easily compute $p_w(z|x)$, we introduced an approximation $q_v(z|x)$. The better this approximation, the lower the KL divergence, and the tighter the lower bound.

---

proving our decomposition

$$\begin{aligned}
\ln p(x) &\stackrel{?}{=} L(q,p) + KL(q,p) \\
&= \mathbb{E}_{z\sim q} \ln \frac{p(x \mid z)p(x)}{q(z \mid x)} - \mathbb{E}_{z\sim q} \ln \frac{p(z \mid x)}{q(z \mid x)} \\
&= \mathbb{E}_{z\sim q} \ln p(x \mid z)p(x) - \mathbb{E}_{z\sim q} \ln p(z \mid x) \\
&= \mathbb{E}_{z\sim q} \ln \frac{p(x \mid z)p(x)}{p(z \mid x)} \\
&= \mathbb{E}_{z\sim q} p(x) \stackrel{\checkmark}{=} \ln p(x)
\end{aligned}$$

83

First, for completeness, we need to prove that our decomposition actually holds. This requires only the basic properties of probability and expectations that we already know from the preliminaries.

It's easiest to work backwards: we'll state the decomposition, and then rewrite it into the log-likelihood of the data.

First, we fill in the definition of L and of the KL divergence.

Note the use of expectations. Since z is a continuous variable, the expectation and KL divergence are integrals rather than sums.Since we only use the properties of the expectation that are the same for both the sum and integral version, we never need to deal with integrals explicitly.

Next, we take out the denominator q(z|x) on both sides. Taking this out of the logarithm gives us an exta term inside the expectation, which we can take out of both expectations. This gives us - E q(z|x) from the first term and + E q(z|x) from the second so they cancel out.

Then, we apply the reverse logic to the second term, and move it into the first, giving us a new denominator. The factors p(x|z) in the numerator and denominator cancel out and we are left with an expectation over p(x). Note that the thing we're taking the expectation for (z) doesn't appear in p(x): we're taking the expectation over a constant. So, we remove the expactation, and arrive at our goal.

minimize -L(q, p)

$$
\begin{aligned}
-\mathrm{L}(\mathrm{q},\mathrm{p}) &= -\mathbb{E}_{\mathbf{z}\sim\mathrm{q}} \ln \frac{\mathrm{p}(\mathbf{x}\mid\mathbf{z})\mathrm{p}(\mathbf{z})}{\mathrm{q}(\mathbf{z}\mid\mathbf{x})} \\
&= -\mathbb{E} \ln \mathrm{p}(\mathbf{x}\mid\mathbf{z}) - \mathbb{E} \ln \mathrm{p}(\mathbf{z}) + \mathbb{E} \ln \mathrm{q}(\mathbf{x}\mid\mathbf{z}) \\
&= \mathbb{E} \ln \frac{\mathrm{q}(\mathbf{x}\mid\mathbf{z})}{\mathrm{p}(\mathbf{z})} - \mathbb{E}\, \mathrm{p}(\mathbf{x}\mid\mathbf{z}) \\
&= \mathrm{KL}(\mathrm{q}(\mathbf{z}\mid\mathbf{x}), \mathrm{p}(\mathbf{z})) - \mathbb{E}_{\mathbf{z}\sim\mathrm{q}(\mathbf{z}|\mathbf{x})} \ln \mathrm{p}(\mathbf{x}\mid\mathbf{z})
\end{aligned}
$$

<- N(0,I)

84

With that, we are almost ready to start using L as a loss function. We just need a few tweaks to allow us to compute it efficiently in a deep learning system like Pytorch.

First, since we want to implement a loss function, we want something to minize. Since we want L as big as possible, we'll minimize -L.

All three probability functions we are left with are ones we can easily compute: q(z|x) is given by the encoder network, p(x|z) is given by the decoder network, and p(z) was chosen when we defined (back in part 1) how the generator works, if we marginalize out x, the distribution on z is a standard multivariate normal.

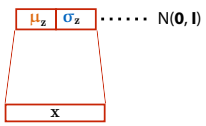
If we break all three out of the expection, and re-arrange we see that we get two very interpretable terms:

- The KL divergence between the distribution q provides for the latent z of x and the prior that p uses for z (which is a standard normal distribution). Note that we're not comparing q to the thing it's approximating here. We're comparing a distribution that is conditioned on x to one that is independent of x. This means that we don't necessarily want this term to go all the way to zero. We'll see later that it functions as a kind of regularizer.
- The log probability of x given z, under the expectation that z is sampled from q(z|x). Here, we see the autoencoder begin to emerge. If we compute q(z|x), sample z from it, and then compute - ln p(x|z), the resulting value should (in expectation) be as low as possible.

---

loss function

$$\mathrm{loss}(\mathrm{q},\mathrm{p}) = \mathrm{KL}(\mathrm{q}(\mathbf{z}\mid\mathbf{x}), \mathrm{p}(\mathbf{z})) - \mathbb{E}_{\mathrm{q}} \ln \mathrm{p}(\mathrm{x}\mid\mathrm{z})$$



85

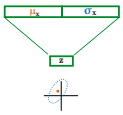
Let's see if this is a loss function we can implement in a system like Pytorch.

The KL term is just the KL divergence between the MVN that the encoder produces for x and the standard normal MVN. This works out as a relatively simple differentiable function of mu and sigma, so we can use it directly in a loss function.

Remember that θ_z is usually restricted to a diagonal matrix, so the network just outputs a vector of the same size as μ_z, which we take to be the diagonal of the covariance matrix.

We'll spare you the working out of the closed-form expression of the KL divergence between Gaussians. If you need it, it's in the slides for the deep learning course. Suffice it to say that it's a simple function of the output of q that's easy to implement in a system like Pytorch.

$$\text{loss}(q, p) = \text{KL}(q(\mathbf{z} \mid \mathbf{x}), p(\mathbf{z})) - \mathbb{E}_q \ln p(\mathbf{x} \mid \mathbf{z})$$



take L samples $\{\mathbf{z}_i\}$ from $q(\mathbf{z}|\mathbf{x})$.

approximate as: $\frac{1}{L} \sum_i \ln p_w(\mathbf{x} \mid \mathbf{z}_i)$

keep things simple: L=1

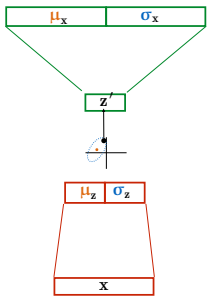
86

The second part of our loss function requires a little more work. It's an expectation for which we don't have a closed form expression. Instead, we can approximate it by taking some samples, and averaging them.

This is the idea of Monte Carlo approximation of expections: to approximate the expectation of f(x) under some probability p, you just take a bunch of samples of x from p and average the resulting values of f(x). IT's what we almost always do implicitly when we use a sample average to represent a population.

To keep things simple, we just take a single sample. We'll be computing the network lots of times during training, so overall, we'll be taking lots of samples, and the optimization method we use (gradient descent) is robust to a little variance.

$$\text{loss}(q, p) = \text{KL}(q(\mathbf{z} \mid \mathbf{x}), N(\mathbf{0}, \mathbf{I})) - \ln p(\mathbf{x} \mid \mathbf{z}')$$



87

So, we replace z (the random variable) by z' (the sample), and remove the expectation.

We now have almost a fully differentiable model. Unfortunately, we still have a sampling step in the middle (and sampling is not a differentiable operation). How do we get from a distribution on z to a sample z', in a way that we can backpropagate through?

---

sampling

$$N^d(\mathbf{0}, \mathbf{1}) \qquad \begin{pmatrix} E_1 \\ \vdots \\ E_d \end{pmatrix} \quad \text{with} \quad E_i \sim N(0, 1)$$

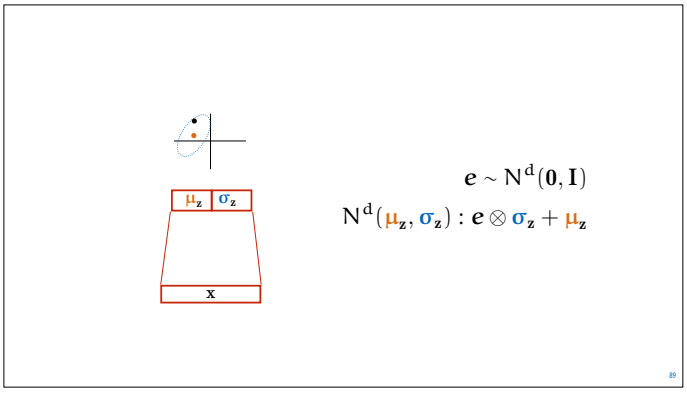
$$N^d(\mu, \sigma) \qquad \boldsymbol{e} \otimes \sigma + \mu \quad \text{with} \quad \boldsymbol{e} \sim N^d(\mathbf{0}, \mathbf{I})$$

The key is to look at the way we normally sample from a normal distribution.

First, sampling from a standard normal distribution in d dimensions is as simple as taking d samples from a one-dimensional standard normal distribution and sticking them into a d-dimensional vector e. If you do this, the resulting vector e is distributed according to a d dimensional multivariate normal distribution.

We'll take the algorithm for sampling from a standard normal as read (we don't need to dig into it). The default approach, if you're interested, is called the Box-Muller transform.

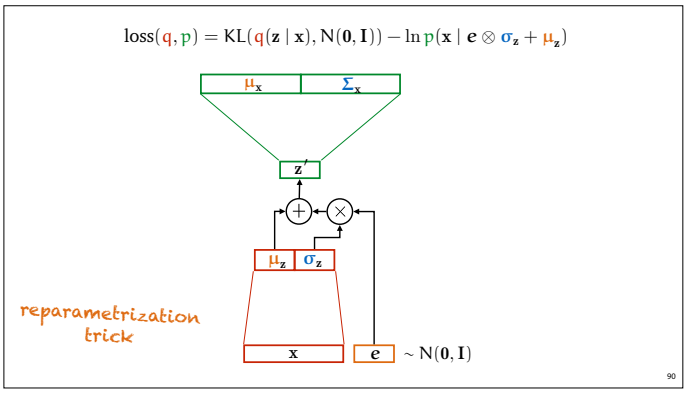
Then, if we want to sample from a diagonal distribution $N(\mu, \sigma)$ with vector mean μ and vector covariance σ, we just take a sample e from the standard normal distribution, and element-wise multiply e by σ and then add μ. The result is a sample from $N(\mu, \sigma)$.



Looking at this algorithm, and applying it to our sample from N(μ_z, σ_z)—the distribution produced by q—we can see two things.

First, the random aspects of the sampling don't depend on the output q. We can do all the random parts (generating e) before we even know what μ_z and σ_z are.

Second, the rest of the algorithm is a simple, differentiable and affine operation. We just take e and multiply it by a vector and add another vector.



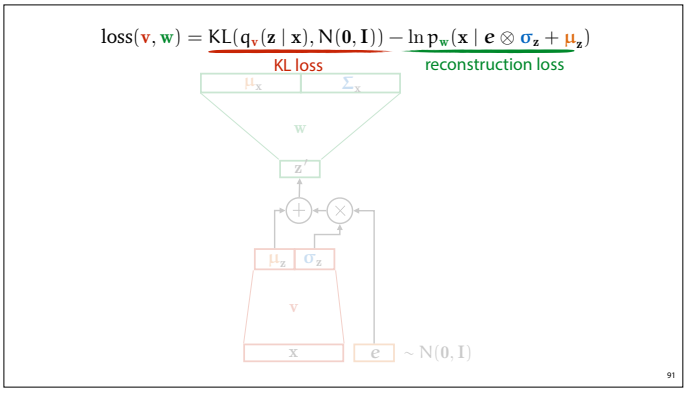
This means that we can basically work the sampling algorithm into the architecture of the network. We provide the network with an extra input: a sample from the standard normal distribution.

Note that this requires the network to produce standard deviations, not variances. So long as the outputs are positive, we can just interpret them as standard deviations, and assume that the network will learn to produce standard deviations.

Why does this help us? We're still sampling, but we've moved the random sampling out of the way of the backpropagation. The gradient can now propagate down to the weights of the q function, and the actual randomness is treated as an input, rather than a computation.

And with that, we have a fully differentiable loss function that we can put into a system like Keras or pytorch to train our autoencoder.

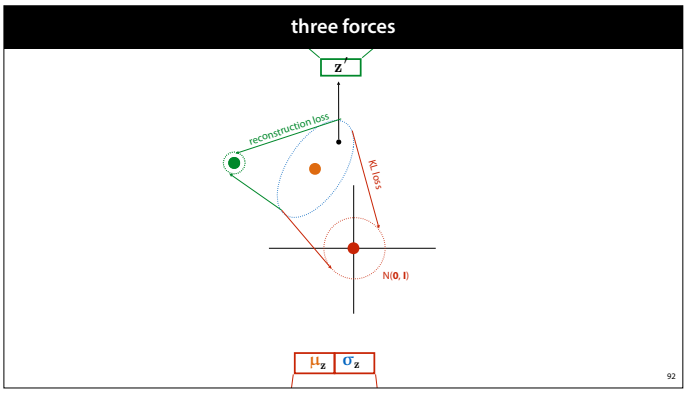
This idea, of working the sampling algorithm into our network, and interpreting the random source of the sampling as another input, is called the reparametrization trick.



The two terms of the loss function are usually called KL loss and reconstruction loss.

The reconstruction loss maximises the probability of the current instances. This is basically the same loss we used for the regular autoencoder: we want the output of the decoder to look like the input.

The KL loss ensures that the latent distributions are clustered around the origin, with variance 1. Over the whole dataset, it ensures that the latent distribution looks like a standard normal distribution.



The formulation of the VAE has three forces acting on the latent space. The reconstruction loss pulls the latent distribution as much as possible towards a single point that gives the best reconstruction. Meanwhile, the KL loss, pulls the latent distribution (for all points) towards the standard normal distribution, acting as a regularizer. Finally, the sampling step ensures that not just a single point returns a good reconstruction, but a whole neighbourhood of points does. The effect can be summarized as follows:

The reconstruction loss ensures that there are points in the latent space that decode to the data.

The KL loss ensures that all these points together are laid out like a standard normal distribution.

The sampling step ensures that points in between these also decode to points that resemble the data.

To understand what happens in the VAE, you should focus on the tension between the reconstruction loss and the other two forces. If we had only the reconstruction loss, the encoder could put the data in a very space set of points in the latent space, and draw very narrow, low variance distributions around these. This leads to a kind of overfitting: the data is very precisely encoder in the latent space, but nothing ensures that the rest of the latent space decodes to something useful.

The KL loss pulls the encoder away from this behavior. It ensures that the decoder wants to output distributions with relatively wide variances. This means that not just a single point decodes to x, but a large region of points does.

question What happens if we have no reconstruction loss to balance the KL loss? Do we ever want the KL loss to reach its minimum of 0? |hide|No, if the KL loss is 0, then the output of the decoder is always equal to N(0, I). This means that the encoder produces a constant output regardless of the input and information makes its way out of the autoencoder.|

The sampling helps the KL loss achieves its aim of spreading out the latent representations. It forces the decoder to generate the data from many points that are spread out over the latent space, not just the ones that are

most likely according to the decoder.

question What happens if we sample a point z' that has high probability under $q(z|x_1)$, but also under $q(z|x_2)$? That is the latent space Gaussians for two points in the data overlap a dit and z' falls in the overlapping region? How does this affect the encoder and the decoder? |hide|The decoder doesn't know whether it's supposed to generate x_1 or x_2, so it will learn to average (mode collapse) between them. This how it learns to interpolate. The encoder probably gets a high score on the KL loss, since the overlapping Gaussians mean they are probably both close to N(0, I). However, the reconstruction loss would get higher if the Gaussians were less overlapping because that would tell the encoder more clearly whether it needs to generate x_1 or x_2.|

---

choosing rec. loss for images

$$-\ln p_w(\mathbf{x} \mid \boldsymbol{\epsilon} \otimes \sigma_z + \mu_z)$$

$-\ln \mathcal{N}(x|\mu, \sigma)$
$-\ln \mathcal{N}(x|\mu, c) \to (x-\mu)^2$ — squared error

$|x-\mu|$ — absolute error
sharper images
Laplace distribution

$H(\text{output}, \text{target})$ — cross-entropy
fast convergence, not a real distribution
cf. Continuous Bernoulli

93

To define an autoencoder, we need to choose the output distribution of our decoder, which will determine the precise form of the reconstruction loss. In these slides, we've used a diagonal normal distribution, but for images, that's not usually the best choice.

We can get slightly better results with a Laplace distribution, but convergence will still be slow.

Better results are achieved with the binary cross entropy. This doesn't correspond to a proper distribution on continuous valued image tensors, but it's often used anyway because of the fact convergence. To fix this problem, you can use something called a continuous Bernoulli distribution, which will give you fast convergence and a theoretically correct VAE.

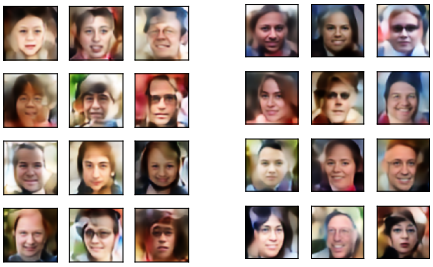
---

after 300 epochs

data ae vae

94

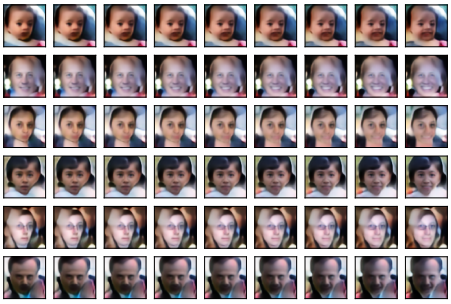
Here are some reconstructions for the regular autoencoder and for the VAE. They perform pretty similarly. There are slight differences if you look closely, but it's hard to tell which is better.



generated

95

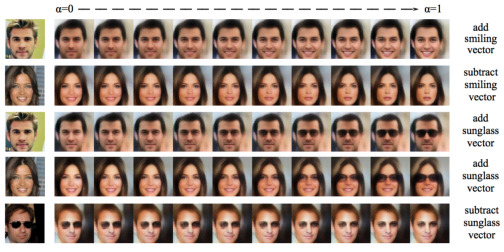
However, if we generate data by providing the generator with a random input, the difference becomes more pronounces. Here we see that the VAE is more likely to generate complete, and coherent faces (although both models still struggle with the background).



96

For completeness, here is the smiling vector, applied to the VAE model.

with VAEs



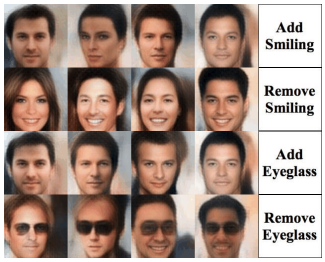
source: Deep Feature Consistent Variational Autoencoder by Xianxu Hou, Linlin Shen, Ke Sun, Guoping Qiu

97

Here are some examples from a more elaborate VAE.

source: Deep Feature Consistent Variational Autoencoder by Xianxu Hou, Linlin Shen, Ke Sun, Guoping Qiu

with VAEs



source: https://houxianxu.github.io/assets/project/dfcvae

98

source: https://houxianxu.github.io/assets/project/dfcvae

from worksheet 5

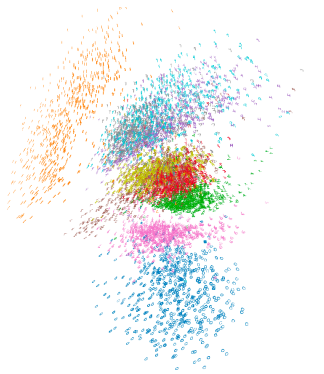
```
for epoch in range(5):
    for images, _ in tqdm(trainloader): # if tqdm gives you trouble just remove it
        b, c, h, w = images.size()

        # forward pass
        z = encoder(images)

        # - split z into mean and sigma
        zmean, zsig = z[:, :latent_size], z[:, latent_size:]
        kl = kl_loss(zmean, zsig)

        zsample = sample(zmean, zsig)

        o = decoder(zsample)
        rec = F.binary_cross_entropy(o, images, reduction='none')
        rec = rec.view(b, c*h*w).sum(dim=1)
        # -- Reconstruction loss. We ask pytorch not to sum the loss, and sum over the
        #    channels and pixels ourselves. This gives us a loss per instance that we
        #    can add to the kl loss

        loss = (rec + kl).mean() # sum the losses and take the mean
        loss.backward()
```

https://github.com/mlvu/worksheets/blob/master/Worksheet%205%2C%20Pytorch.ipynb

99

Here is what the algorithm looks like in Pytorch. Load the 5th worksheet to give it a try.

https://github.com/mlvu/worksheets/blob/master/Worksheet%205%2C%20Pytorch.ipynb



100

In this worksheet, the VAE is trained on MNIST data, with a 2D latent space. Here is the original data, plotted by their latent coordinates. The colors represent the classes (to which the VAE did not have access).

If you run the worksheet, you'll end up with this picture (or one similar to it).

interpolation

AE

i went to the store to buy some groceries .
i store to buy some groceries .
i were to buy any groceries .
horses are to buy any groceries .
horses are to buy any animal .
horses the favorite any animal .
horses the favorite favorite animal .
horses are my favorite animal .

VAE

he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .

101

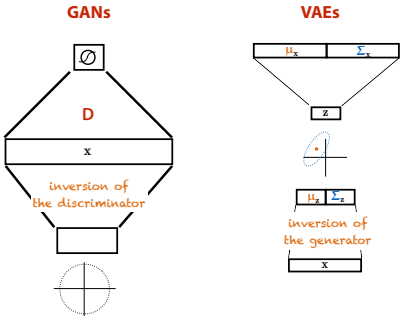
While the added value of the VAE is a bit difficult to detect in our example, in other domains it's more clear.

Here is an example of interpolation on sentences. First using a regular autoencoder, and then using a VAE. Note that the intermediate sentences for the AE are non-grammatical, but the intermediate sentences for the VAE are all grammatical.

source: Generating Sentences from a Continuous Space by Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, Samy Bengio
https://arxiv.org/abs/1511.06349

---

summary: generative modeling



102

We see that GANs are in many ways the inverse of autoencoders, in that GANS have the data space as the inside of the network, and VAEs have it as the outside.

GANs and VAEs

Allow us to train generator networks, avoiding mode collapse

GANs	VAEs
Better for images, often poor in other domains.	Work for language, music, etc.
Ad-hoc model, difficult to establish what is being optimized.	Derived from first principles.
Can't handle discrete data easily.	Allow mapping from data to latent space.
Derived by *inverting* a discriminator.	Can't handle discrete latent variables easily.
	Derived by *inverting* a generator.

103

VAEs and PCA

Mapping to low-dimensional *whitened* latent space (zero, mean, decorrelated).

PCA	VAEs
Linear transformation	Nonlinear transformation
Analytical solution	GD required
Principal components often meaningful, ordered by impact.	Latent dimensions not usually meaningful. *Directions* in latent space meaningful.

104

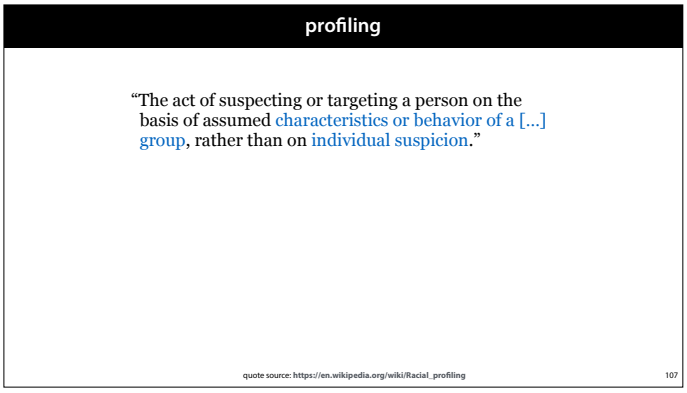
mlcourse@peterbloem.nl

Deep Generative Models
Part 5: Social impact 3

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

This week and the last, we've discussed a lot of probability theory. With these tools in hand, we can go back to our discussion on social impact, and try to make it more precise. We can now talk a lot more precisely about how to reason probabilistically and what kind of mistakes people tend to make. Unsurprisingly, such mistakes have a strong impact on the way machine learning algorithms are used and abused in society.

|section|Social impact 3|
|video|https://www.youtube.com/embed/r4DYGXmbk_E|



Specifically , in this video, we'll look at the problem of profiling.

When we suspect people of a crime or target them for investigation, based on their membership of a group rather than based on their individual actions, that's called profiling.

Probably the most common form is racial profiling; which is when the group in question is an ethnic or racial group. Examples include black people being more likely to be stopped by police, or Arabic people being more likely to be checked at airports.

Other forms of profiling, such as gender or sexual orientation profiling also exist in various contexts.

We saw an example of this in the first social impact video: a prediction system (essentially using machine learning) which predicted the risk of people in prison re-offending when let out. This system, built by a company called Northpointe, showed a strong racial bias.

As we saw then, it's not enough to just remove race as a feature. So long as race or ethnicity can be predicted from the features you do use, your model may be inferring from race.

Profiling doesn't just happen in automated systems. And lest you think this is a typically American problem, let's look a little closer to home.

A few years ago, a Dutch hip-hop artist called Typhoon was stopped by the police. The police admitted that the combination of his skin colour and the fact that he drove an expensive car played a part in the choice to stop him. This caused a small stir in the Dutch media and a nationwide discussion about racial profiling.

The main argument usually heard is "if it works, then it is worth it." That is, in some cases, we should accept a certain amount of racism in our criminal procedures, if it is in some way successful.

This statements hides a lot complexity: we're assuming that such practices are successful, and we're not defining what being successful means in this context. Our responsibility, as academics, is to unpack such statements, and to make it more precise what is actually being said. Let's see if we can do that here.

We'll focus on the supposed pros and cons of profiling and on what it means for a profiling method to be successful, regardless of whether it's an algorithm or a human doing the profiling.

As an example of how automated systems can perform profiling, without being explicitly programmed to, we can also stay in the Netherlands.

Less than a month ago as this section's video was recorded, however, the Dutch government fell. In a parliamentary investigation at the end of last year, it was found that the tax service had wrongly accused an estimated 26 000 families of fraudulent claims for childcare benefits, often requiring them to pay back tens of thousand of euros, and driving them into financial difficulty.

There were many factors at play, but an important problem that emerged was the use of what were called "self-learning systems." In other words, machine learning. One of these, the risk-indicator, candidate lists for people to be checked for fraud. The features for this classification included, among other things the nationality of the subject (Dutch/non-Dutch). The system was a complete black box, and investigators had no insight into why people were marked as high risk. People with a risk level above 0.8 were automatically investigated, making the decision to investigate an autonomous one, made by the system without human intervention.

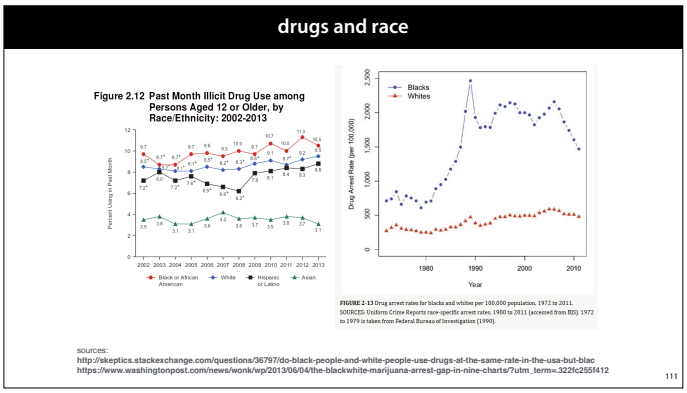
One of the biggest criticisms of the tax service in the child welfare scandal is how few of the people involved understood the use of algorithms in general, and the details of the algorithms they were using specifically.

This hopefully goes some way towards explaining why we've felt it necessary to discuss social impact in these lectures. We're teaching you how to build complex systems, and history has shown again and again that policy makers and project managers are happy to deploy these in critical settings without fully understanding the consequences. If those responsible for building them, that is you and me, don't have the insight and the ability required to communicate the potential harmful social impacts of these technologies, then what chance does anybody else have?

https://www.groene.nl/artikel/opening-the-black-box

https://autoriteitpersoonsgegevens.nl/sites/default/files/atoms/files/

onderzoek_belastingdienst_kinderopvangtoeslag.pdf



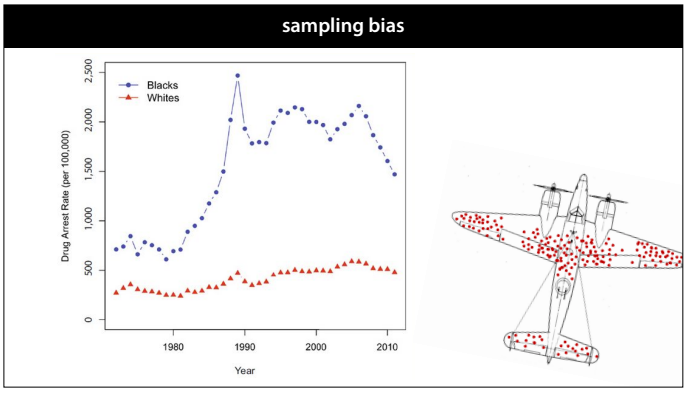
Since this is a sensitive subject, we'll try to make our case as precisely as possible, and focus on a specific instance, where we have all the necessary data available: illicit drug use in the US. The US has a system in place to record race and ethnicity in crime data. The categorization may be crude, but it'll suffice for our purposes.

From these graphs, we see on the left that black people engage in illicit drug use more than people of other ethnicities, and that they are also arrested for it more than people of other ethnicities. However, the rate of use is only marginally higher than that of white people, whereas the arrest rate can be as much as five times as high as that for white people,

This points to one potential problem: racial profiling may very easily lead to disproportionate effects like those seen on the right. Even if there's difference in the proportion with which black people and white people commit a particular crime, it's very difficult to ensure that the profiling somehow honors that proportion. But we shouldn't make the implicit assumption that that's the only problem. If the proportions of the two graphs matched, would profiling then be justified? Is the problem with profiling that that we're not doing it carefully enough, or is the problem that we're doing it at all?

We'll look at some of the most common mistakes made in reasoning about profiling, one by one.

sources:
https://www.washingtonpost.com/news/wonk/wp/2013/06/04/the-blackwhite-marijuana-arrest-gap-in-nine-charts/?utm_term=.322fc255f412



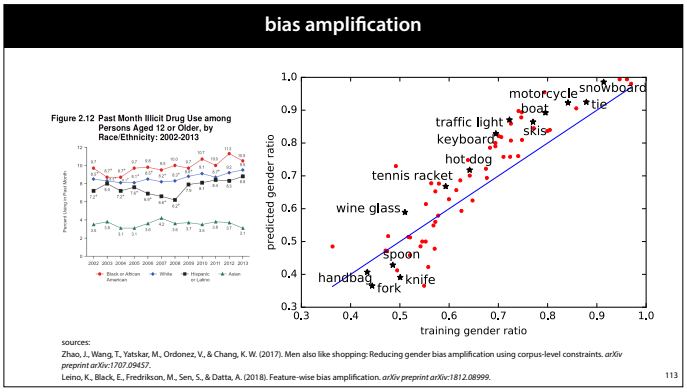
One problem with an automated system like that of Northpointe is that there is a strong risk of data not being sampled uniformly. If we start out with the arrest rates that we see on the right, then a system that predicts illicit drug use will see a lot more black drug users than white ones. Given such a data distribution, it's not suprising that the system learns to associate being black with a higher rate of drug use.

This is not because of any fundamental link between race and drug use, but purely because the data is not representative of the population. We have a sampling bias.

It's a bit like the example of the damaged planes in WWII we saw at the start of the fourth lecture: if we assume a uniform distribution in the data, we will conclude the wrong thing. In that case we weren't seeing the planes that didn't come back. Here, we aren't seeing the white people that didn't get arrested.

Note that it's not just algrithms that suffer from this problem. For instance, if we leave individual police officers to decide when to stop and search somebody, they will likely rely on their own experience, and the experience of a police officer is not uniform. There are many factors affecting human decision making, but one is that if they already arrest far more black than white people, they are extremely likely to end up with the same bias an algorithm would end up with.

So let's imagine that this problem is somehow solved, ande we get a perfectly representative dataset, with no sampling bias. Are we then justified in racial profiling?



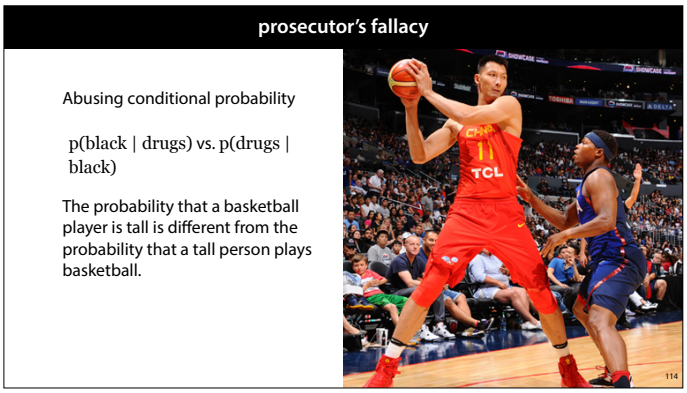
You'd be forgiven for thinking that if a bias is present in the data, that the model simply reproduces that bias. In that case, given a dataset without sampling bias, we would start with the minor discrepancies on the left, and simply reproduce those. Our model would be biased, but we could make the case that it is at least reproducing biases present in society.

However, it's a peculiar property of machine learning models that they may actually amplify biases present in the data. That means that even if we start with data seen on the left, we may still end up with a predictor that disproportionately predicts drug use for black people.

An example of this effect is seen on the right. For an image labeling tasks, the authors measured gender ratios in the training set, for subsets of particular nouns. For instances, for images containing both a wine glass and a person, we see that the probability of seeing a male or female person in the data is about 50/50, but in the predictions over a validation set, the ratio shifts to 60/40.

It's not entirely clear where this effect comes from. The second paper quoted shows that it's related to our choice of inductive bias, so it's a deep problem, that gets to the heart of the problem of induction. Even the Bayes' optimal classifier can suffer from this problem. For our current purposes it's enough to remember, that even if our input has biases that are representative, there's no guarantee that our output will.

It appears that this is a problem that may be impossible to solve. But let's imagine, for the sake of arguments, that we somehow manage it. What if we get a perfectly representative dataset with no sampling bias, and we somehow ensure that our model doesn't amplify bias. Can we then do racial profiling?



Much of racial profiling falls into the trap of the prosecutor's fallacy. In this case the probability that a person uses illicit drugs, given that they're black is very slightly higher than the probability that they do so given that they are white, so the police feel that they are justified in using ethnicity as a feature for predicting drug use (it "works").

However, the probability that a person uses illicit drugs given that they are black is still very much lower than the probability of not using illicit drugs given that they they are black. This probability is never considered.

As we see in the previous slide the rates are around p(drugs|black) = 0.09 vs. p(~drugs|black) = 0.91. If the police blindly stop only black people, they are disadvantaging over 90% of the people they stop.

To help you understand, consider a more extreme example of the prosecutor's fallacy. Let's imagine that you're trying to find professional basketball players. The probability that somebody is tall given that they play professional basketball, p(tall| basketball) is almost precisely 1. Thus, if you're looking for professional basketball players, you are justified in only asking tall people. However, the probability of somebody playing professional basketball given that they're tall, is still extremely low. That means that if you go around asking tall people whether they are profesional basketball players, you'll end bothering a lot of people before you find your basketball player, and probably annoying quite a few of them.

What if

the data is a fair representation of the population

and

the model doesn't amplify bias

and

we've correctly used Bayes' rule?

115

So, have we now covered all our bases? We get a dataset that is a fair representation, our model doesn't amplify biases, and we correctly use Bayes' rule.

Can we then use the model to decide whether or not to stop black people in the street?

The answer is still no.

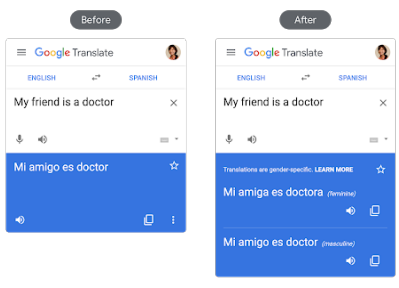
At this point, we may be certain that our predictions are accurate, and we have accurately estimated the probability accurately that a particular black person uses drugs illicitly.

However, the fact that those predictions are accurate tells us nothing about whether the action of then stopping the person will be effective, justified, or fair. That all depends on what we are trying to achieve, and what we consider a fair and just use of police power. The accuracy of our predictions cannot help us guarantee any of this.

Just because your **predictions** are accurate, doesn't make your **actions** sound.

This is an extremely important distinction in the responsible use of AI. There is a very fundamental difference between making a prediction and taking an action based on that prediction.

We can hammer away at our predictions until there's nothing left to improve about them, but none of that will tell us anything about whether taking a particular action is justified. How good a prediction is and how good an action is are two entirely different questions, answered in completely different ways.



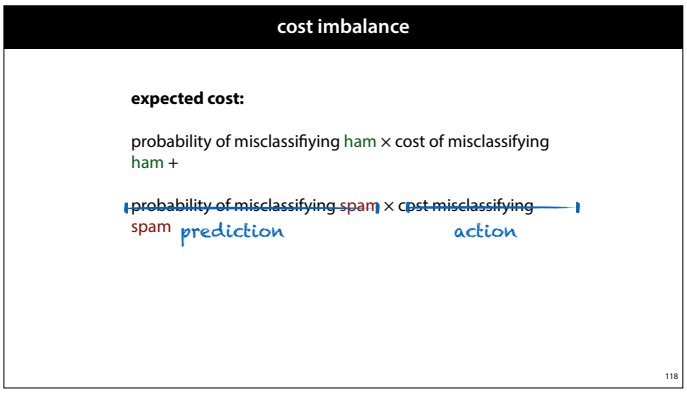
source: https://ai.googleblog.com/2020/04/a-scalable-approach-to-reducing-gender.html

117

Recall the Google translate example from the first lecture. Given a gender neutral sentence in English, we may get a prediction saying that with probability 70% the word doctor should be translated as male in Spanish and with probability 30% it should be translated as female. There are almost certainly biases in the data sampling, and there is likely to be some bias amplification in the model, but in this case we can at least define what it would mean for this probability to be accurate. For this sentence, there are true probabilities, whether frequentist or Bayesian, for how the sentence should be translated. And we can imagine an ideal model that gets those probabilities absolutely right.

However, that tells us nothing about what we should do with those probabilities. Getting a 70/30 probability doesn't mean we are justified in going for the highest probability, or in sampling by the probabilities the model suggests. Both of those options have positive consequences, such as a user getting an accurate translation, and negative consequences, such as a user getting an accurate translation and the system amplifying gender biases.

In this case, the best solution turned out to be a clever interface design choice, rather than blindly sticking with a single output.



This is related to the question of cost imbalance. We may get good probabilities on whether an email is ham or spam, but until we know the cost of misclassification we don't know which action to prefer (deleting the email or putting it in the inbox). The expected cost depends on how accurate our predictions are, but also on which actions we decide to connect to each of the predictions. This is an important point: cost imbalance is not a property of a classifier in isolation: it's a property of a classifier, inside a larger system that takes actions. The cost imbalance for a system that deletes spam is very different from the cost imbalance in a system that moves spam to a junk folder.

Here, we should always be on the lookout for creative solutions in how we use our predictions. Moving spam to a junk folder instead of deleting it, showing users multiple translations instead of just one, and so on. The best ways of minimizing cost don't come from improving the model performance, but from rethinking the system around it.

In questions of social impact, the cost of misclassification is usually extremely hard to quantify. If a hundred stop-and-searches lead to two cases of contraband found, how do we weigh the benefit of the contraband taken off the streets against the 98 stop-and-searches of innocent individuals. If the stop-and-search is done in a biased way, with all black people being searched at least once in their lifetime and most white people never being searched, then the stop-and-search policy can easily have a very damaging effect on how black people are view in society.

It's very easy, and very dangerous to think that we can easily quantify the cost of mistakes for systems like these.

https://twitter.com/OdedRechavi

---

correlation and causation

A and B are correlated: I can predict A from B (and vice versa)

A causes B: changing A causes a change in B (but *not* vice versa.

Correlation does not imply causation.

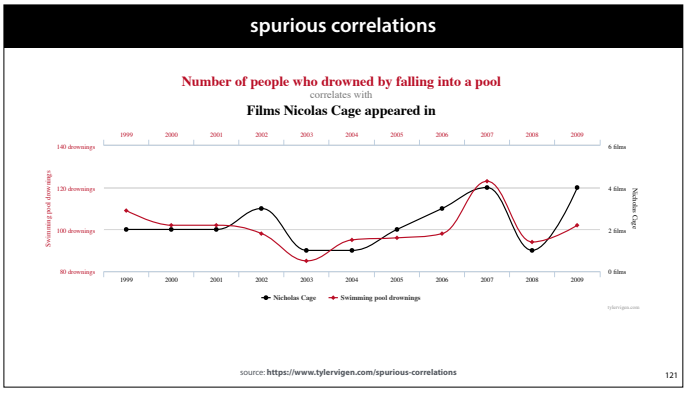
No correlation without causation.

120

A large part of choosing the right action to take based on a prediction, is separating correlation and causation. A lot of social issues, in AI and elsewhere, stem from confusions over correlation and causation, so let's take a cerful look at these two concepts.

Two observables, for instance, being black and using illicit drugs are correlated, if knowing the value of one can be used to predict the value of the other. It doesn't have to be a good prediction, it just has to be better than it would be if we didn't know the value of the first.

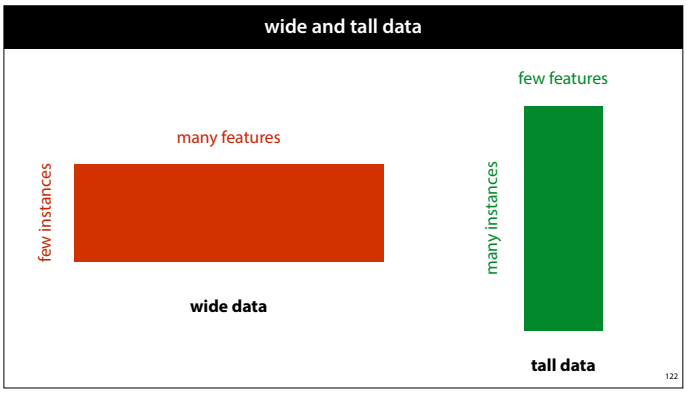
This doesn't mean that the first causes the second. I can from the smoke in my kitchen that my toast has burned, and if somebody tells me that my toaster has been on for half an hour, I can guess that there's probably smoke in my kitchen. Only one of these causes the other. There are many technical definition of what constitutes causaility, but in general we say that A causes B if changing A causes a change in B. Turning off the toaster removes the smoke from my kitchen, but opening a window doesn't stop my toast burning.



When talking correlation, the first thing we need to be on the lookout for is spurious correlations. According to this data here, if we know the number of films Nicolas Cage appeared in in a given year, we can predict how many people will die by drowning in swimming pools.

This is not because of any causal mechanism. Nicolas Cage is not driven by drowning deaths, and people do not decide to jump into their pools just because there are more Nicolas Cage movies (whatever you think of his recent career). It's a spurious correlation. It looks like a relation in the data, but because we have so few examples for each, it's possible to see such a relation by random chance (especially if you check many different potential relations).

The key property of a spurious correlation is that it goes away if we gather more data. If we look at the years 2009-now, we will (most likely) not see this pattern.

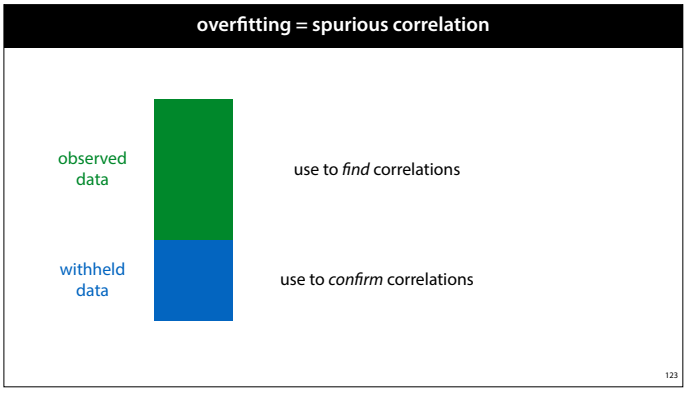


Gathering more data can hurt or help you here.

The more features you have, the more likely it is that one of them can be predicted from the other purely by chance, and you will observe a correlation when there isn't any. We call this wide data.

Adding instances has the opposite effect. The more instances, the more sure we can be that observed correlations are true and not spurious. We call this tall data.

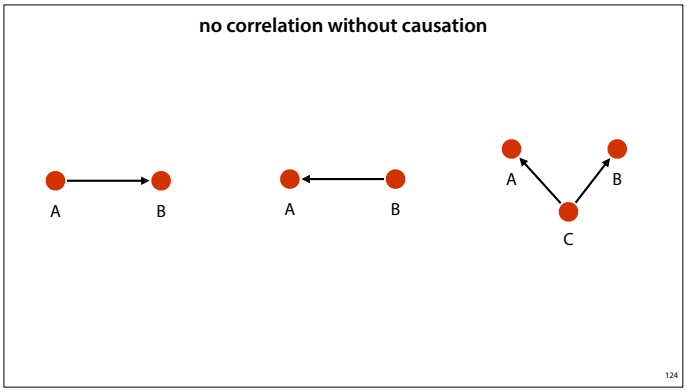
Thus, if we are conservative with our features, and liberal with our instances, we can be more confident that any observed correlations are correct. The litmus test is to state the correlations you think are true and then to test them on new data. In life sciences, this is done through replication studies, where more data is gathered and the stated hypothesis from an existing piece of research is evaluated be the exact same experiment. In machine learning, we withhold a validation set for the first round of experiments, and then a test set for the second (and sometimes a meta-test set for replication studies).



This is essentially a way of guarding against spurious correlations, or in other words, overfitting is just predicting from a spurious correlation. The definition of a spurious correlation is one that disappears when you gather more data, so if our correlation is spurious, it should not be present in the withheld data.

A good machine learning model finds only true correlations and no spurious correlations. How to make that distinction without access to the withheld data, is the problem of induction.

This also tells us that using many features increases the probability of overfitting. If we see the target label as another column in our data, then the more different features we have, the more likely it is that over our small set of instances, one of them if correlated with the training labels. If this correlation is spurious, it goes away if we gather more data: that is, it's not present in the validation and test sets. Predicting from such a spurious correlation is overfitting.



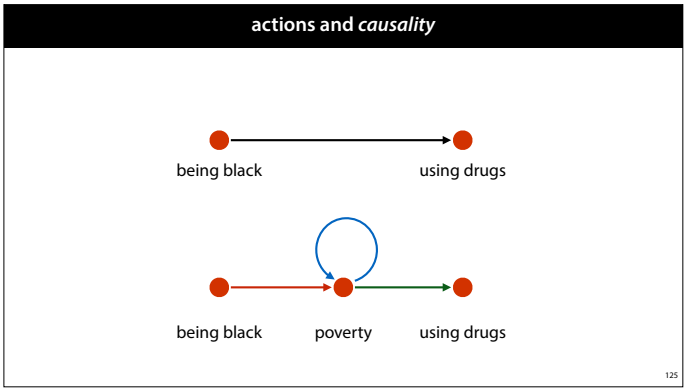
So if we rule out spurious correlations, what can we say that we have learned when we observe a correlation?

If I see you have a runny nose, I can guess you have a cold. That doesn't mean that having a runny nose causes colds. If I make the exam too difficult this year, it affects all grades, so somebody can predict from your failing grade that other students are also likely to have a failing grade. That doesn't mean that you caused your fellow student to fail. This is the cardinal rule of statistics: correlation is not causation. It is one that you've hopefully heard before.

There is another rule, that is just as important, and a lot less famous. No correlation without causation. If we observe a correlation and we've proved that it isn't spurious, there must be a causation somewhere.

Simplifying things slightly, these are the ways a correlation can occur. If A and B are correlated then either A causes B, B causes A, or there is some other effect that causes both A and B (like me writing a difficult exam). A cause like C is called a confounder.

It is important to note that C doesn't have to be a single simple cause. It could be a large network of many related causes. In fact, causal graphs like these are almost always simplifications of a more complex reality.



So let's return to our example of illicit drug use in America. We know that there's a small correlation between race and illicit drug use (even though there is a far greater discrepancy in arrests). What is the causal graph behind this correlation?

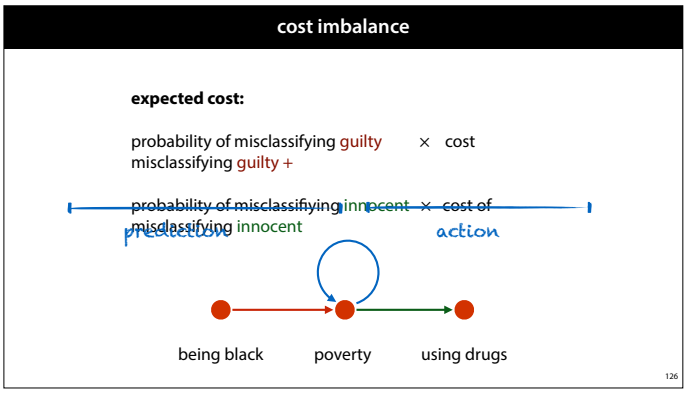
At the top we see what we can call the racist interpretation. That is, racist in the literal sense: seeing race as the fundamental cause of differences in behaviour. Put simply, this interpretation assumes a fundamental, biological difference in black people that makes them more susceptible to drug addiction. Few people hold such views explicitly these days, and there is no scientific evidence for it. But it's important to remember that this kind of thinking was much more common not too long ago.

At the bottom, is a more modern view, backed by a large amount of scientific evidence. Being black makes you more likely to be poor, due to explicit or implicit racism in society, and being poor makes you more likely to come into contact with illicit drugs and makes you less likely to be able to

escape addiction.

There is a third effect, which I think is often overlooked: poverty begets poverty. The less money your parents have, the lower your own chances are to escape poverty. Having to live in poverty means living from paycheck to paycheck, never building up savings, never building up resilience to sudden hardship, and never being able to invest in the long term. This means that on average, you are more likely to increase your poverty than to decrease it.

The reason all this is relevant, is that for interventions to be effective, they must be aligned to the underlying causes. In the world above, racial profiling may actually be effective (although it could still be unjust). However, in the picture below, racial profiling actually increases pressure on black people, pushing them further into poverty. Even though the police feel like they're arresting more drug users, they are most likely strengthening the blue feedback loop (or one similar to it).



If we ignore data bias, and assume a perfect predictor, we still have to deal with the cost of misclassification.

Misclassifying a guilty person can feed into this blue feedback loop. In the best case, it leads to embarrassment and loss of time for the person being searched. But there can also be more serious negative consequences.

One subtle example is being found out for a different crime than the one you were suspected of, due to the search. For instance, imagine that the if the predictor classifies for driving a stolen car, and during the stop, marijuana is found. This may at first seem like a win: the more crimes caught, the better. However, the result of doing this based on profiling is again that we are feeding into the blue feedback loop.

There is a certain level of crime that we, as society allow to pass undetected, because detecting it would have too many negative consequences. It would cost too much to detect more crime, or infringe too much on the lives of the innocent.

This is true for any society anywhere, although every society makes the tradeoff differently. However, if we stop people because they are predicted, through profiling, to be guilty crime X, and then arrest them for crime Y, then we end up setting this level differently for black people than for white people. Essentially, by introducing a profiling algorithm for car theft, we are lowering the probability that people get away with marijuana possession, and we are lowering it further for black people than for white people.

Causality plays a large role in setting the rules for what is and isn't fair. In law this is described as differentiation, justly treating people differently based on their attributes and discrimination, unjustly treating people differently based on their attributes.

For instance, if we are hiring an actor to appear in in an ad for shaving cream, we have a sound reason for preferring a male actor over a female actor, all other qualifications being the same. There is a clear, common-sense causal connection between the attribute of being male and being suitable for the role.

If we are hiring somebody to teach machine learning at a university, preferring a male candidate over a female one, all else being equal, is generally considered wrong, and indeed illegal. This is because there is broad (though not universal) agreement that there is no causal link between your gender and how suitable you are as a teacher of machine learning. There may be correlations, since machine learning is still a male-dominated field, but no causal link.

That is, differentiation is usually allowed, if and only if there is an unambiguous causal link between the sensitive attribute and job suitability.

So what if we:

- Sample a representative dataset,
- Prevent bias amplification,
- Apply Bayesian reasoning correctly,
- Carefully design sensible actions,
- Only follow causal patterns?

Can we then permit ourselves some profiling?

128

Let's take one final look at the profiling question, including everything we've learned.

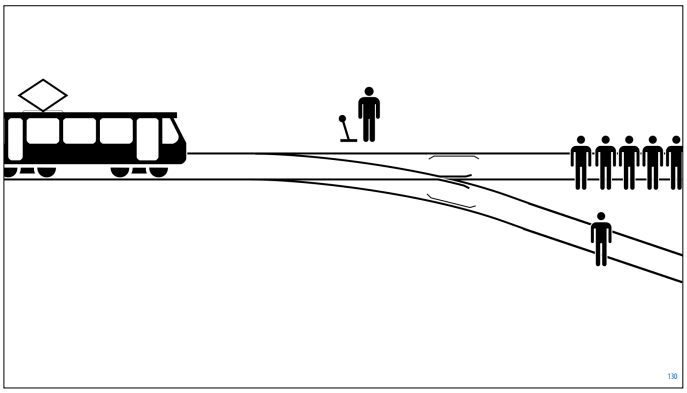
Say we somehow get a representative dataset, which is difficult. We somehow prevent bias amplification, which may be impossible. We apply Bayesian reasoning correctly, which is possible, we carefully design sensible actions based one some quantification of cost, which is very difficult. And we take care to consider all causal relations to avoid inadvertent costs and feedback loops, which is difficult at best.

Imagine a world where we can do all this, and get it right. Are we then justified in applying profiling?

Consequentialism: the consequences of our actions determine how ethical our actions are.

129

What we have taken so far is a purely consequentialist view. The consequences of our actions are what matters. The more positive those consequences, the more ethical the system is, and vice versa.



130

Consider the famous trolley problem: there is a an out of control trolley thundering down the tracks towards five people, and you can throw a switch to divert it to another track with one person on it. This illustrates some of the pitfalls of consequentialist thinking.

The consequentialist conclusion is that throwing the switch is the ethical choice. It saves five lives and sacrifices one.

131

Now imagine a maverick doctor who decides that he will kill one person, harvest their organs, and use them to save five terminally ill people in need of transplants. With two kidneys, two lungs and a heart he should easily be able to find the patients to save.

From a consequentialist perspective, this is exactly the same as the trolley problem. One person dies, five are saved. And yet, we can be certain that many of the people who considered throwing the switch in the trolley problem to be the ethical choice, would not be so certain now.

Without taking a position ourselves, what is it that makes the difference between these two situations? Why is the second example so much less agreeable to many people?

Consequentialism: the consequences of our actions determine how ethical our actions are.

Deontological ethics: moral principles determine how ethical actions are.

Kant: "So act as to treat humanity, whether in thine own person or in that of any other, always as an end, never merely as a means."

The ethics of dignity. John Laird, 1940

132

Without going into details, we can say that some actions are in themselves more morally disagreeable than others, regardless of the consequences. This quality, whatever it is, leads to deontological ethics. Ethical reasoning based on fundamental moral codes, regardless of consequences.

Such codes are often tied to religion and other aspects of culture, but not always. Kant's categorical imperative is an example of a rule that is not explicitly derived from some religious or cultural authority. Broadly, it states that to take an ethical action, you should only follow a rule if you would also accept it as a universal rule, applying to all.

One aspect that crops up in deontological ethics is that of human dignity. This may be an explanation for the discrepancy between the trolley and the doctor. Flipping the switch is a brief action made under time pressure. This is in contrast to the premeditated murder and organ harvesting of an innocent person. The latter seems somehow a deeper violation of the dignity of the victim, and therefore a more serious violation of ethics.

Kant, again, considered this a foundational principle of basic morality, to treat another human being as a means to an end, rather than as an end in themselves is to violate their dignity.

Consider the difference between killing a human being in order to eat them and killing a human being to get revenge for adultery. From a consequentialist perspective, the first has perhaps the greater utility: in both cases, someone dies, but in one of them we get a meal out of it. From the deontological perspective of human dignity, the first is the greater sin. When we cannibalize someone, we treat them as a means to filling our stomach, without regard for their humanity. When we kill out of revenge, even though it may be wrong or disproportional, we treat the other as a human being and our action is directly related to one of theirs.

---

fundamental rights

It is fundamentally unfair to hold an individual responsible for the actions of others that share their attributes.

Everybody has the *right* to to be judged on their own actions.

hold responsible:

subject to a traffic stop, not give parole, search at an airport, not give a credit card, make it more difficult to get a job, subject to financial auditing.

133

To bring this back to our example, we can now say that our analysis of racial profiling is entirely consequentialist. We have been judging the cost of our actions and trying to maximize it by building the correct kind of system. It is perhaps not surprising that a lot of AI ethics follows this kind of framework, since optimizing quantities is what we machine learning researchers do best.

The deontological view, specifically the one focused on human dignity, gives us a completely different perspective on the problem. One that makes the correctness and efficacy of the system almost entirely irrelevant. From this perspective it is fundamentally unjust to hold a person responsible for the actions of another. If we are to be judged, it should be on our own actions, rather than on the actions of another.

To prevent crime from being committed, or to make some reparations after a crime is committed, some people need to suffer negative consequences: this ranges from being subjected to traffic stops to paying a fine. A just system only

subjects those people to these negative consequences, that committed or planned to commit the crime. From this perspective, racial profiling, even if we avoided all the myriad pitfalls, is still a fundamental violation of dignity. It treats the time and dignity of Black people as a means to an end, trading it off against some other desirable property, in this case, a reduction of crime.

While human dignity is often posed as hard constraint: something that should never be violated, in many cases this cannot be reasonably achieved. For instance, any justice system faces the possibility of convicting innocent people for the crimes of others. The only way to avoid this is to convict no one, removing the justice system entirely. So, we allow some violation of human dignity in order that we can punish the guilty.

However, if we do have to suffer a certain probability that our dignity will be violated, we can at least ask that such violations are doled out uniformly.

---

profiling: recap

Data is often not a representative sample

Bias is often amplified by ML models (or people)

Correct Bayesian reasoning is difficult (especially for people)

Predictive models don't tell us what actions to take. For this we need a *causal* model.

Even if none of the mistakes are made, profiling may still be unethical from a deontological perspective.
Specifically, it's usually considered violation of human dignity to hold someone responsible for the actions of others.

134

We won't tell you what to believe about profiling (although you may be able to guess my opinion). You'll need to decide for yourself where to draw the line. The only thing we ask is that you have a clear idea of the arguments for and against. If you argue that profiling is "effective", can you explain what exactly that means? Can you explain why none of the statistical errors above are being made, or why their impact is outweighed by other factors?

Do you understand the difference between consequentialist and deontological arguments? Do you understand why arguments about human dignity cannot be countered with arguments for the effectiveness of profiling?

mlcourse@peterbloem.nl

135

image source: https://www.trouw.nl/nieuws/ouders-bij-debat-toeslagenaffaire-mijn-leven-is-naar-de-klote~bc3f3e52/