Score-based generative models are a different formulation of diffusion models, might be considered a wider family of generative models that diffusion models belong to. While its derivation is motivated differently, both score-based generative models and diffusion models will come out at the same point.

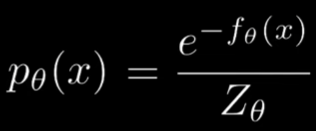
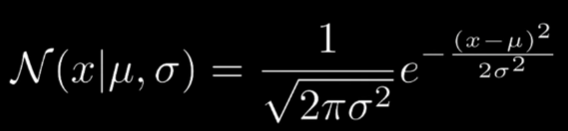
Score-based generative models help us understand a lot of the derivations in diffusion.

Table of content

1. Score
2. Score Matching
3. Noise Perturbation
4. Denoising Score Matching
5. Sampling
6. Multiple Noise Perturbation
7. Differential Equations
8. Link to Diffusion Models

Introduction (Motivation)

Suppose that we have a dataset and we want to learn the probability distribution function (PDF) of the dataset so that we can sample from it to produce new datapoints. We can try to learn an approximated PDF of the dataset, the approximated PDF is written as p\_theta to the real PDF p. A good way to parameterise it is by the following:

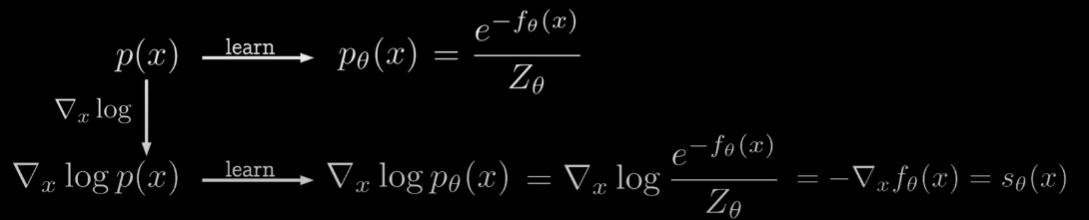
 , can see similarities with 

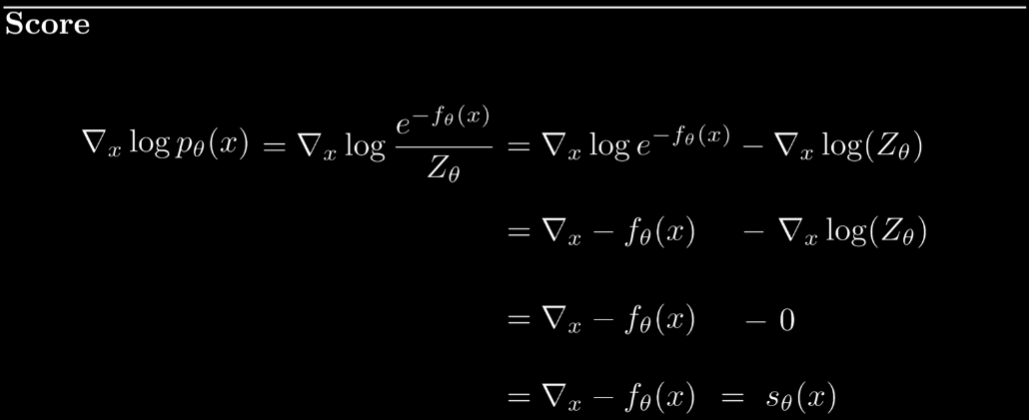
This is a good way as the negative exponent enforces the PDF constraint of being non-negative (<=0), and then dividing it by a learnable normalising constant Z\_theta (integrate pdf wrt x for the whole range of x) ensures the PDF constraint of being within [0, 1].

The exponent, f\_theta() (the model), can be learned to output how dense the probability is (controls PDF value wrt to x, for eg, the higher the pdf value for x means we are at a dense area for that x). However, trying to learn p\_theta with this formula gives computation issues as to find normalising constant Z\_theta, we have to integrate the PDF, p\_theta, for all values of x. (too high dim and intractable to compute).

Going forward from this approach, finding an approximated p\_theta with the above formula, requires certain restrictions to the model f\_theta or approximating Z\_theta. But there is a trick we can do, and that is called the score.

1. Score



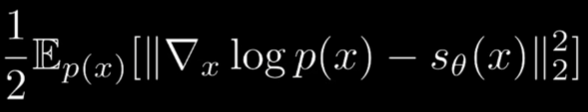


The score refers to taking the gradient of the log of the pdf p\_theta(x) with respect to x. After a series of simplifications, we are just left with a much more simplified expression, the gradient of -f\_theta(x) with respect to x.

This is way better because the normalising constant Z\_theta that is tedious to get is removed, which introduces many restrictions to the model.

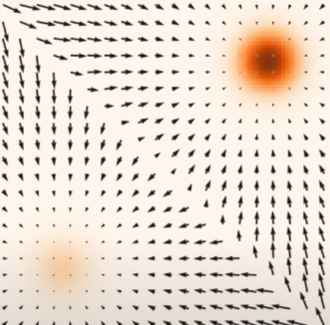
1. Score Matching

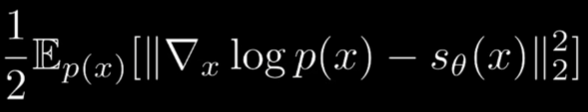
Score matching refers to approximating of the scores of the original distribution (original PDF, p(x)) with the predicted scores from s\_theta(x):



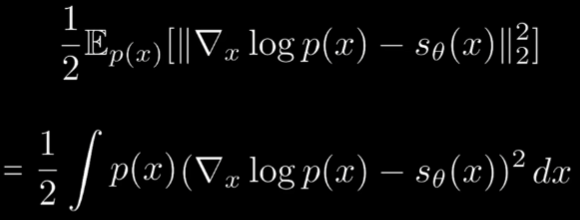
We are just minimising the above expression, the difference between the original score (score of p(x)) and predicted score (score of p\_theta(x)).

Intuitively, the score function  makes sense too, it tells you the direction to move the datapoint to maximise the probability of it, to get to the data point.  
Scores actually represent, for a given point x in the dataspace, the direction to go to get closer to real datapoints. (Scores represented by the arrows in the figure below, orange blobs = real training datapoints and saturation = density)

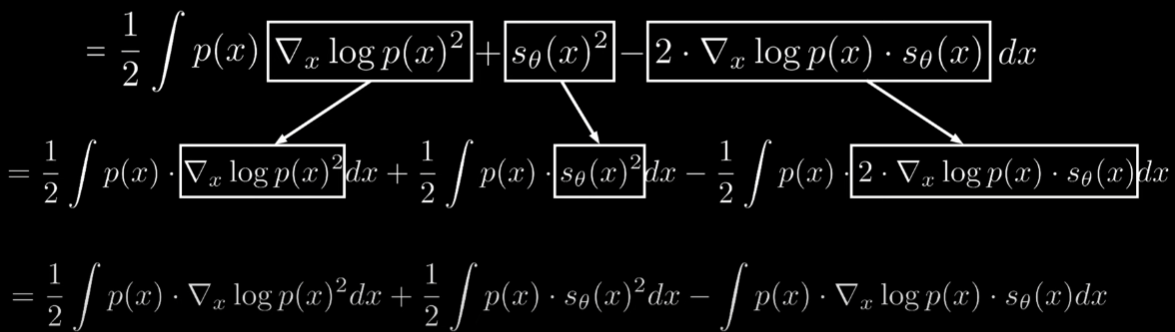


So, we should just try to minimise the objective as above: 

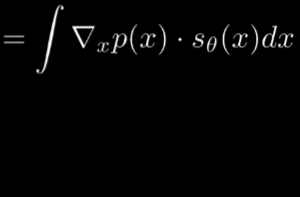
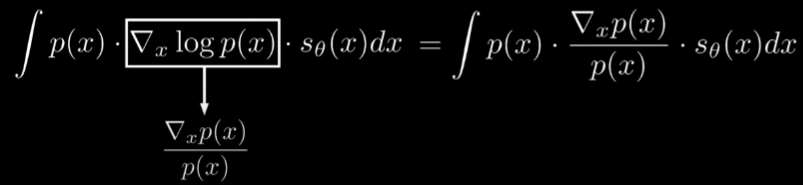
But, we do not know the true PDF, p(x), and thus we do not know the term . Hence, we perform some tricks and simplifications below to solve this problem.

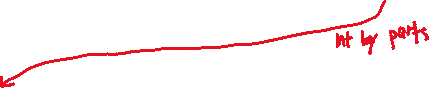


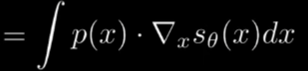
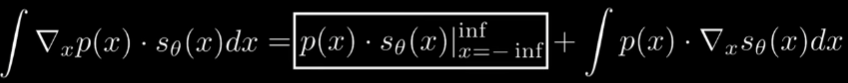




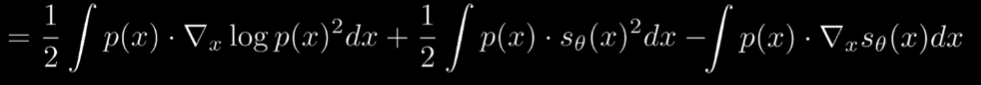




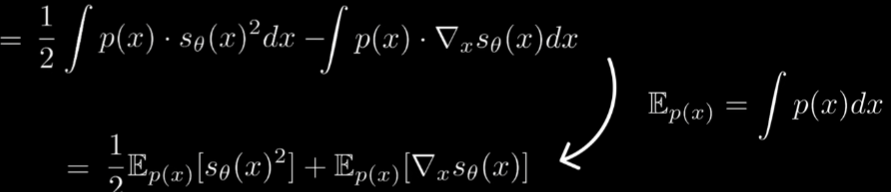




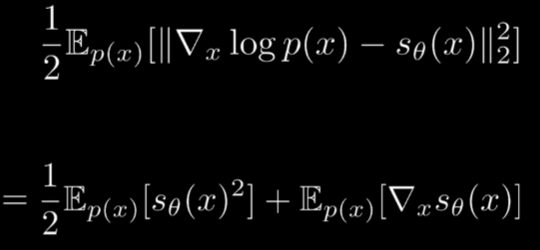


since the first term does not include the learned score model s\_theta(x), we can effectively ignore it for our objective function of minimising with respect to theta (model’s learnable parameters)





Results:

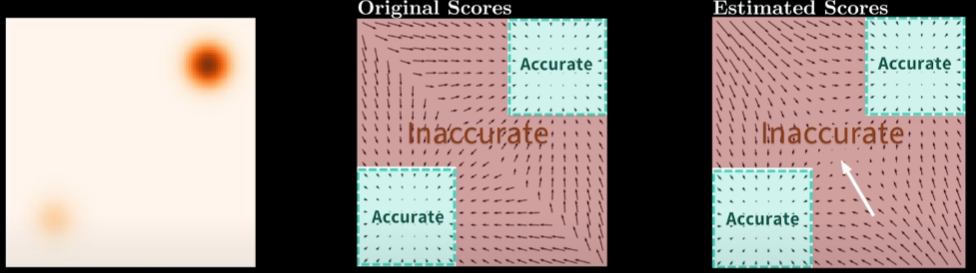




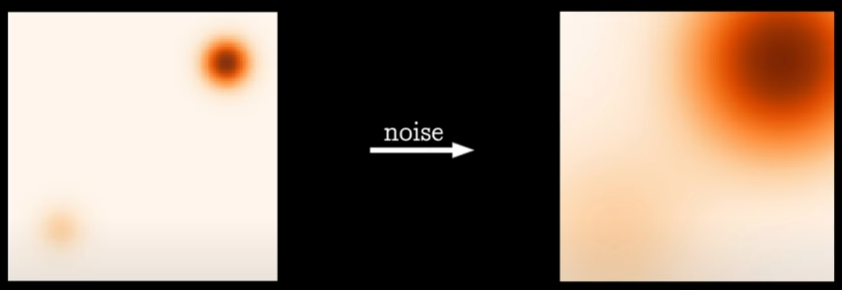
The final objective does not have the unknown p(x) terms.

However, this final objective has 2 problems:

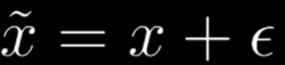
1. The 2nd term is computationally expensive because we are have to the Jacobian of s\_theta(x), which requires back propagation for every individual input predictor variable x
2. There is low data coverage. During training we mostly see data that comes from the training samples distribution. So, the model is very good at learning the score function in these areas (range of seen training instance values of x) and is poor at unseen spaces (when we make a prediction, sample randomly and land at an unseen spaces)



One way to solve the 2nd issue, low data coverage problem, is to add noise to the training images, like in diffusion. As such, we get more variety in training samples and cover more of the data space

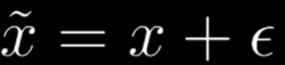


This is simply done by taking all of our training samples and adding Gaussian noise to them:

 , where 

1. Noise Perturbation

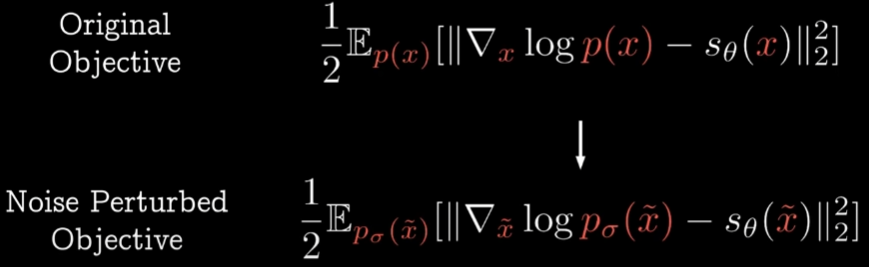
To partially solve problem 2 of the Minimising the Score Matching Objective (Low Data coverage), we need to find a way to increase data coverage (more variation in training samples)

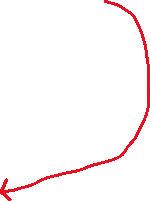
By taking all of our training samples and adding Gaussian noise to them, for each original training sample x, we obtain  , 

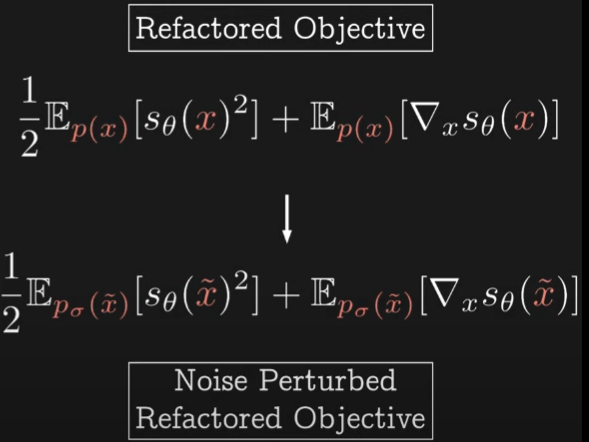
Thus, for the PDF, we get the new the original training images + random Gaussian noise as:

, sigma being from the error distribution’s variance.

The original objective function is modified as such. x to noisy x and p(x) to p\_sigma(noisy x)









How much noise we add for the images depend on sigma in , but unlike in diffusion where we use a noise scheduler to control the variance of the noise, we will choose sigma as a single number as a hyperparameter.

* If sigma is too large (higher amount of noise added), the perturbed distribution p\_sigma(x) will be very different from the original distribution p(x), causing inaccurate training and sampling
* If sigma is too small (smaller amount of noise added), then the variety in training samples will be still too small and there will still be the low data coverage problem.

But the expensive computation problem for the refactored objective function (2nd term, due to the Jacobian), problem 1, as mentioned above still persists.

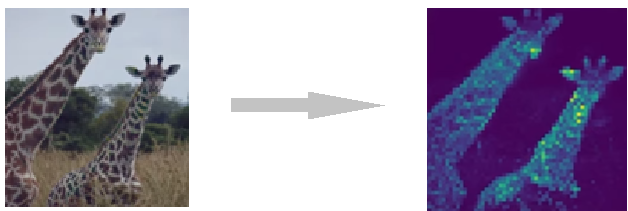
1. Denoising Score Matching

To solve problem 1 of the Minimising the Score Matching Objective (Expensive computation due to the Jacobian in the 2nd term), we need to find another objective less expensive to calculate.

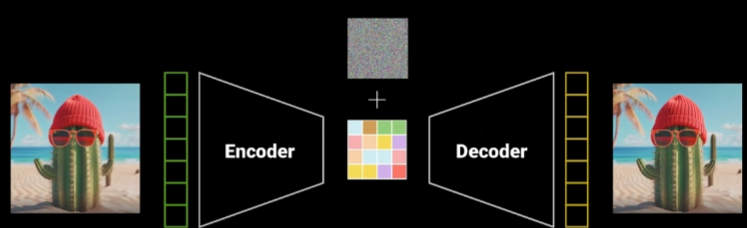
It was found that there exists a connection between score matching and denoising autoencoders.

Denoising Autoencoders

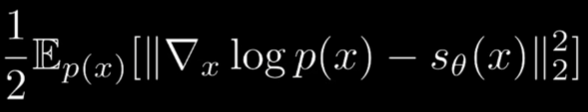
The goal is of denoising autoencoders is to learn the representative features of data.



Same structure as an autoencoder, but noise is added to the latent vector at the bottle neck. The autoencoder is still task to predict the original image. The goal is of denoising autoencoders is feature extraction. By learning how to predict the original input after adding noise, separating noise from data, the autoencoders learn representative features of the data (to be able to predict an input despite being noisy)

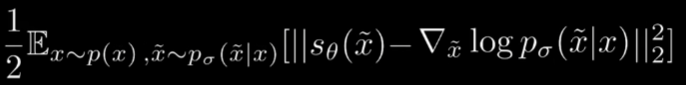


New Denoising Score Matching Objective Function

Recap: Score Matching = approximating of the scores of the original distribution (original PDF, p(x)) with the predicted scores from s\_theta(x), objective function = 

**Denoising** Score Matching Objective Function:





* The term  from the original score matching objective is removed while the term  is new.



* This makes a big difference as we can calculate  directly but not 



Final Denoising Score Matching Objective Function



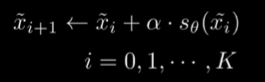


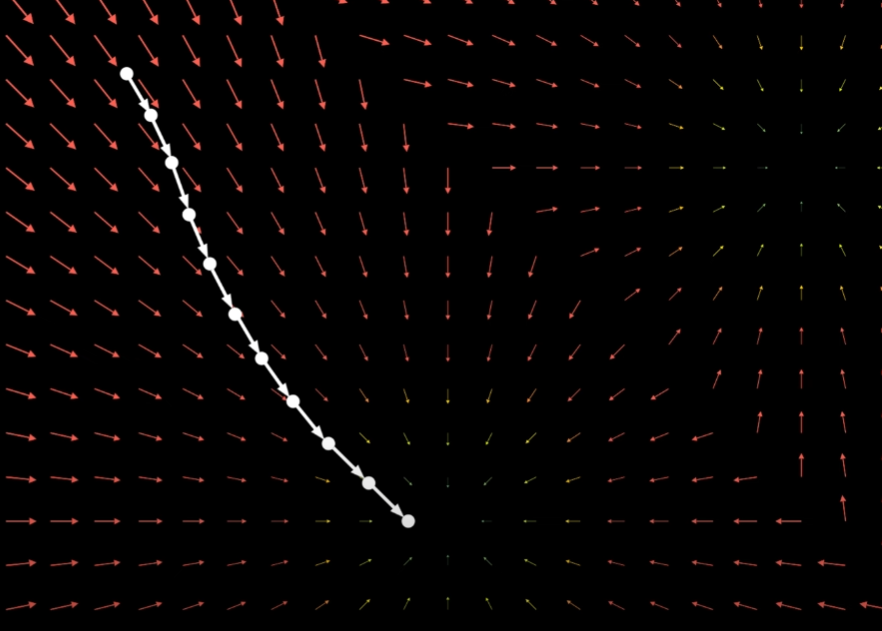
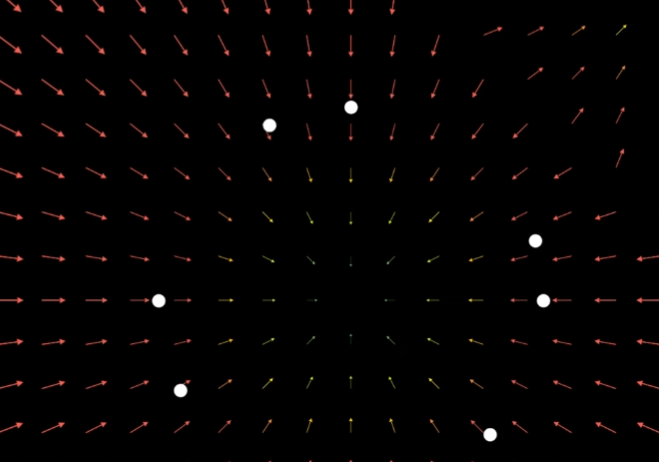
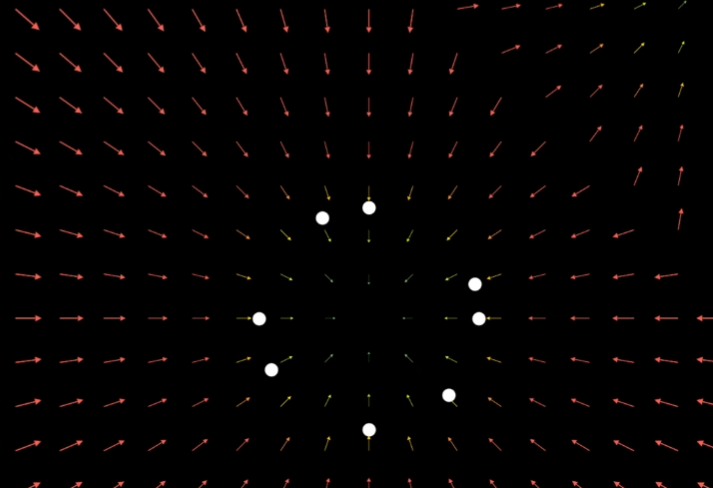
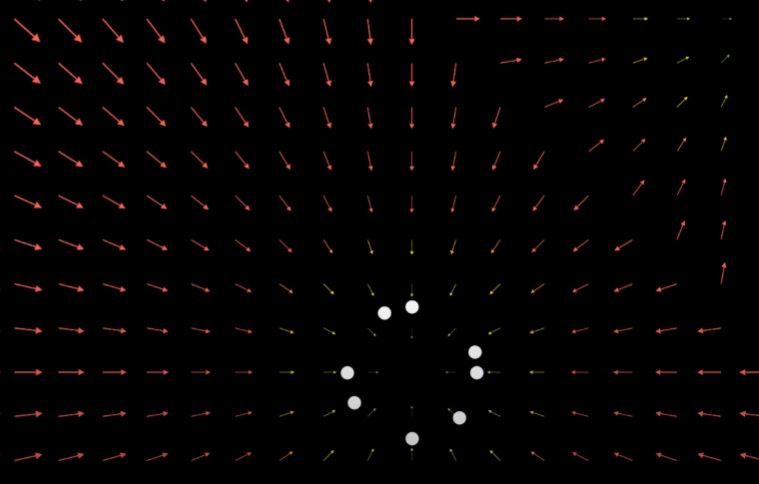
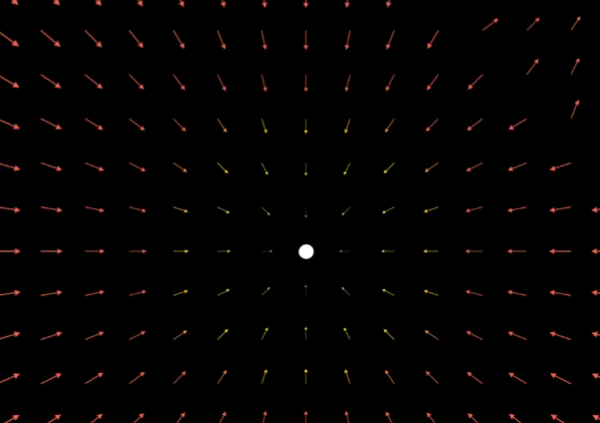
This new training objective makes a lot of sense as it is just essentially trying to minimise the score ,which is the step towards the real data (the “difference”)

It also provides direct and less expensive computation than the original score matching refactored objective function that has a Jacobian in its 2nd term.

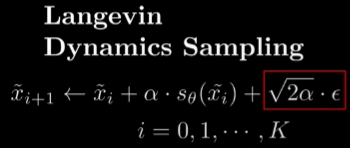
1. Sampling

To train the model, we need to perform sampling. The idea is to sample a random point on the dataspace and move towards the direction of a real datapoint in repetitive small steps (move via scores at the different positions)

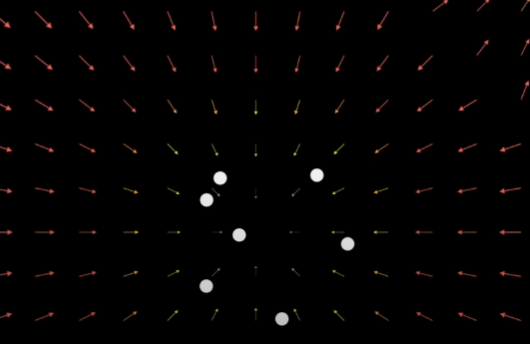
The naïve approach is to do simple sampling . But these result in a problem. Different instances of x, will all collapse to the same point (mean of the training data). The reason is because the scores always point to a higher likelihood, which is the mean of the training data as it has the highest density of data.

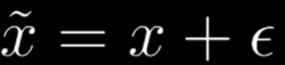
To solve this problem, we sample with:

, where epsilon is random Gaussian noise 

This improved sampling will result in different datapoints hovering around the mean instead, which is actually what could belong in the dataset, instead of all sampled points being the same mean:

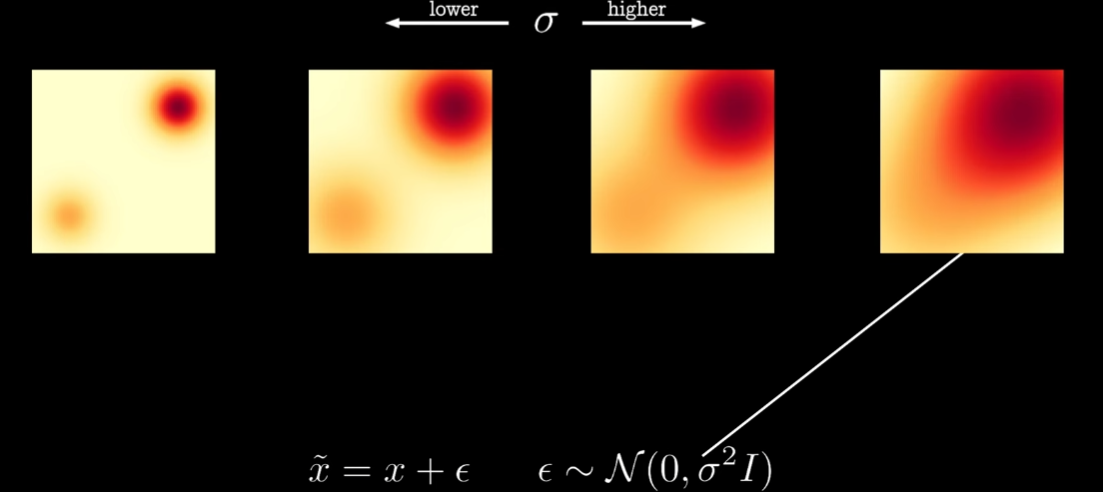


1. Multiple Noise Perturbations

Remember that we still have the problem of a trade-off when adding a different amount of noise to the images for noise perturbations,  to get  to solve the low data coverage problem.

If the noise to be added is given as such , and sigma is a hyperparameter. Adding more noise = covers more of the dataspace but also changes the data distribution:

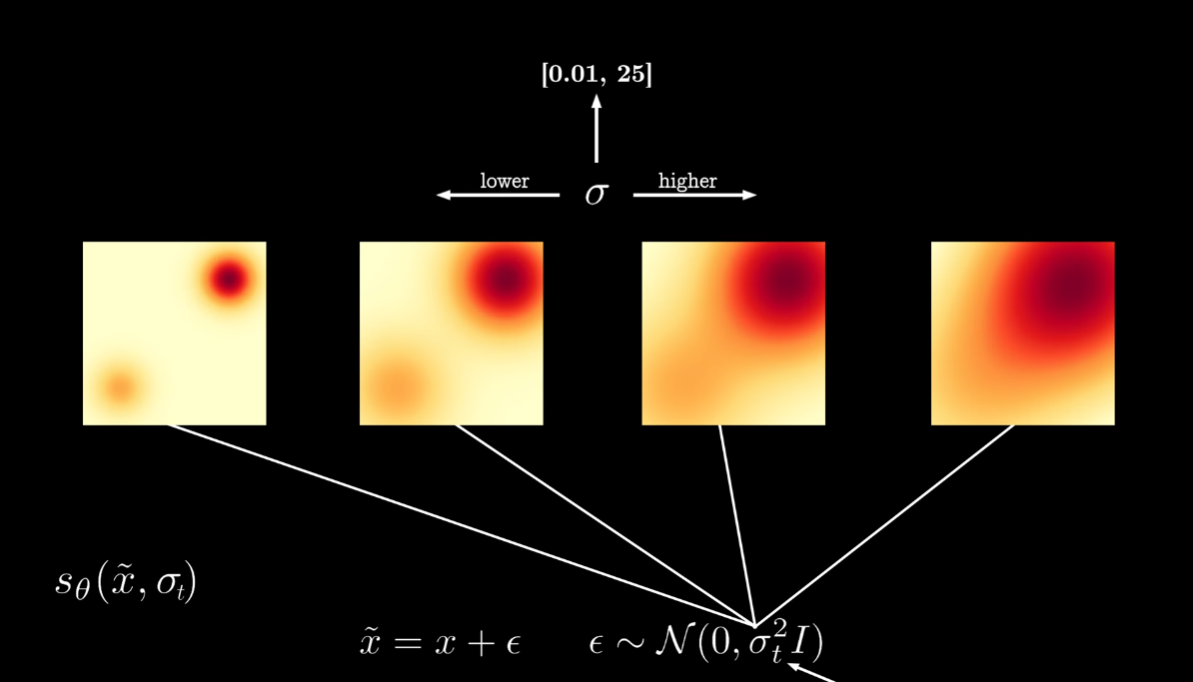
1. Add too much noise, sigma is high
   1. Data distribution changes too much. End up approximating p(x) too inaccurately with p\_sigma(x) that is greatly different
2. Add too little noise, sigma is low
   1. Noisy input X tilda will barely contains any noise, very similar to the original input X, resulting in the original low data coverage problem, training has high variance



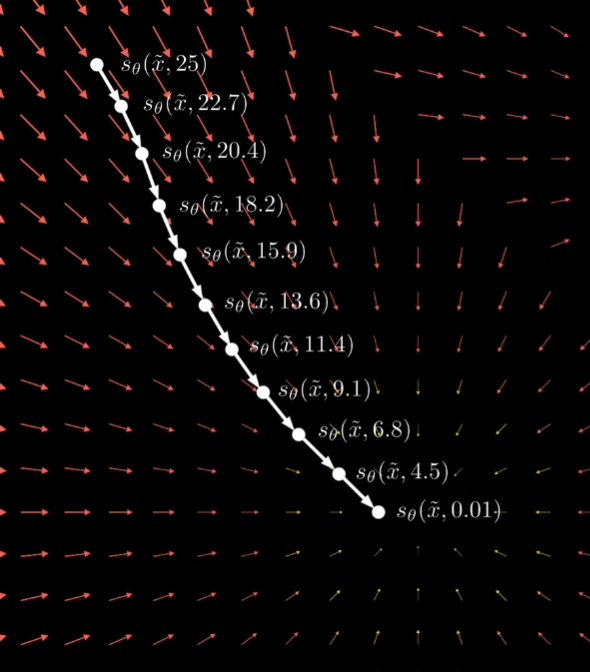
To solve this, we perform multiple noise perturbation. We do both extremes, high sigma and low sigma, and also everything in between. We train the model on low, middle and high noise levels. In practice, we typically vary sigma over some pre-defined range, e.g. [0.01, 25]. So the model, s\_theta will see datapoints all over the space.

We also condition the model on the noise level specific to each input, sigma\_t, which provides the model with additional information with predicting the scores.

Notation change, sigma\_t instead of sigma to indicate that sigma is no longer a fixed value anymore, but a sequence of values, low and high, varying for different inputs.



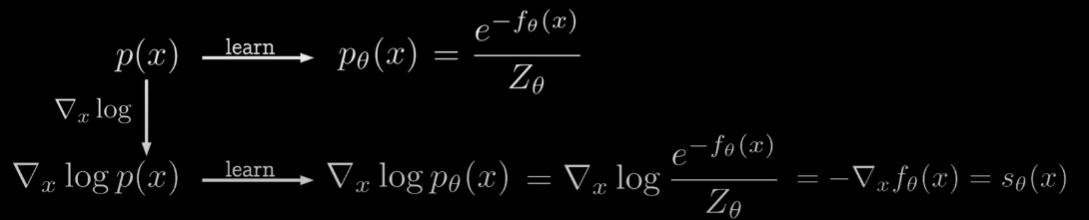
During sampling, we can start from the largest sigma value of the noise interval, and as the data moves towards the real data point, Xt to Xt+1, we lower the sigma value. But how many noise levels to use? As high as possible, as the levels go to infinite, the noise perturbations becomes a stochastic process, more importantly, sampling will also become a stochastic process (stochastic process = systems that evolve overtime that inherently has randomness), solving the low data coverage problem as generation is stochastic (random and time-evolving). This improves the quality of the samples, easier implementation and compute the exact log likehoods.

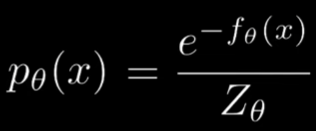


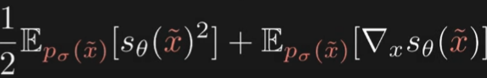
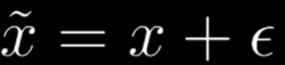
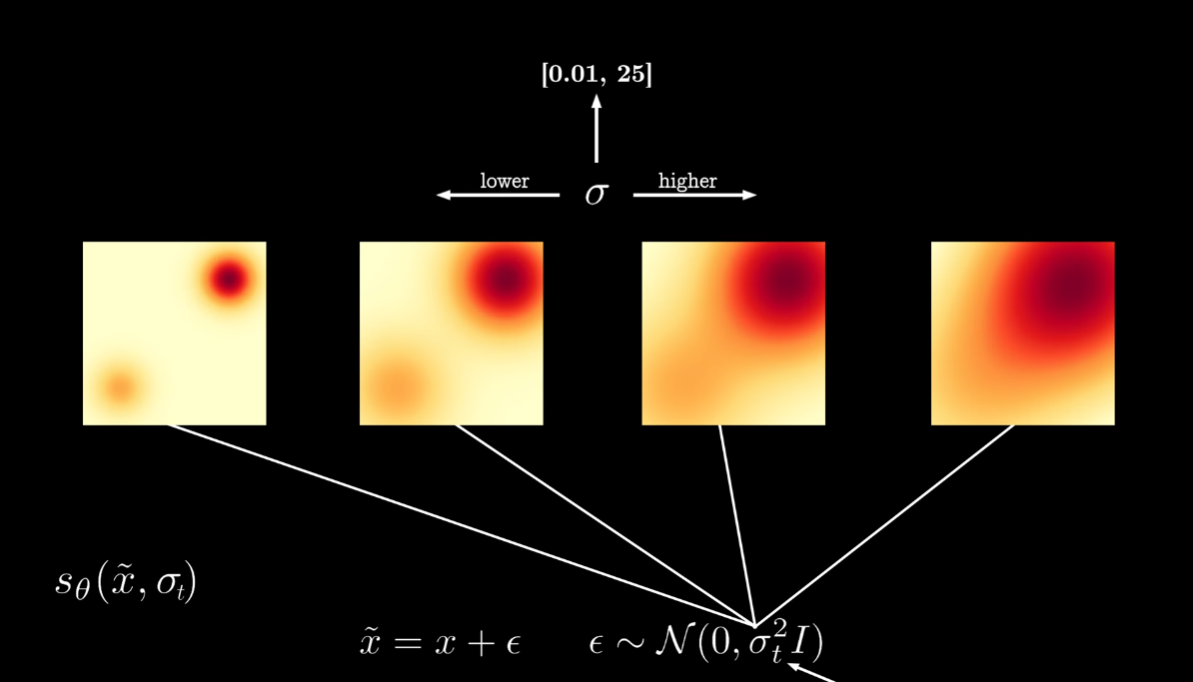
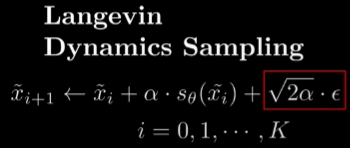


The number of noise levels correspond to some form of scheduler (?) deciding how the noise varies and how many levels.

Overview



 🡪 approximation of original PDF to be learned, but  too expensive

1. Scores
   1.  = , proposed as an easier quantity to be learned   
      (The model)
2. Scores Matching (Training)
   1. Provides us with a final refactored objective function, but has 2 problems:
      1. 1) Expensive to calculate – due to Jacobian for 2nd term
      2. 2) Low data coverage – function only predicts does not generalise well, it samples (predicts) datapoints that are seen in the training samples and does not go beyond it (limited distinct outputs)
3. Denoising Score Matching (Training)
   1. Find an alternative objective function to solve the 1st problem of expensive calculation.
   2. Final Objective Function
      1. 
4. Noise Perturbation (Training)
   1. Aims to solve the 2nd problem of low data coverage by adding noise to all the training samples.  , 
   2. We get  adding more noise = covers more of the dataspace but also changes the data distribution.
   3. Sigma is a fixed hyperparameter that controls the noise, but this results in a trade-off problem:
      1. Add too much noise, Sigma is high 🡪 Data distribution changes too much. End up approximating p(x) too inaccurately with p\_sigma(x) that is greatly different
      2. Add too little noise, Sigma is low 🡪 Noisy input X tilda will barely contains any noise, very similar to the original input X, resulting in the original low data coverage problem, training has high variance
5. Multiple Noise Perturbations (Training)
   1. Aim to solve Noise Perturbation trade-off, to ultimately solve the problem of low coverage
   2. Do both extremes, high sigma and low sigma, and also everything in between. We train the model on low, middle and high noise levels. In practice, we typically vary sigma over some pre-defined range, e.g. [0.01, 25]. So the model, s\_theta will see datapoints all over the space.
   3. We also condition the model on the noise level specific to each input, sigma\_t, which provides the model with additional information with predicting the scores.
   4. Notation change, sigma\_t instead of sigma to indicate that sigma is no longer a fixed value anymore, but a function of t, t = [0, 1], ranges correspond to min and max noise
   5. 
   6. The amount of noise variations, example noise steps for each sampling Xt+1 from Xt, is the noise levels and as noise levels -> infinity, sampling becomes a stochastic process, meaning that there will no longer be low data coverage and samples will become random
6. Sampling
   1. Prediction/Sampling formula 

**Annex:**

Score-based Generative Models like to Diffusion

