1.
Thanks, Scott, for the quick intro.
Hi everyone — I'm Annie, I am from Columbia, I am a quant intern in the New York office this summer.
During my first rotation with QMA, I worked on using deep learning to solve high-dimensional PDEs.
In today's presentation, I'll walk you through how we apply deep learning techniques to solve two types of PDEs: the Feynman-Kac and the Fokker-Planck equations.
Let's get started.

2.
Here's a quick outline of what I'll be covering.
First, I will talk about three core advantages of using deep learning to solve PDEs — things that make it stand out from conventional methods.
Then I'll explain how a deep learning–based PDE solver actually works — what the network is doing, and what the training process looks like.
After that, I'll focus on the Deep Galerkin Method, which we use to solve both backward and forward PDEs. I'll show examples using both the Black-Scholes and SABR models.
Finally, I'll briefly introduce two other methods we explored and touch on some challenges of deep learning approaches
And we will have a short QA session after that.

3.
Let's start with three key advantages of deep learning, things that really set it apart from classical methods like finite difference or Monte Carlo
First — it handles high-dimensional problems.
For inputs, we don't need to discretize the domain into a fixed grid.
Instead, the solver randomly samples input points, making it practical to work in high dimensions.
Here, "dimensions" refer to the number of state variables or model factors — like asset prices, volatilities, or interest rates.
Second — once trained, the model gives you instant outputs.
Like a closed-form formula, you can plug in volatility, strike, or maturity, and get the price and Greeks immediately.
There's no need to rerun simulations like in Monte Carlo and since the model is fully differentiable, we can extract the greeks directly.
Third — the model is flexible and reusable.
I used the same network architecture for both the Black-Scholes and SABR models.
It can generalize across models with minimal code change for different training objective
So overall, we're looking at a solver that's scalable, efficient, and model-agnostic — which makes it a strong fit for our pricing tasks.
4.
Now let's zoom in on how this actually works.
The core idea is simple: use a neural network to learn the solution to a PDE.
We start by feeding in the variables that the PDE depends on — in this toy example, that's time t and space x. In a financial context, these inputs might be things like stock price, volatility, or interest rate.
The neural network acts as a function approximator: It processes the input and outputs a prediction — which is our estimate of the solution at that point
We use a loss function to evaluate the accuracy of the prediction.
It plugs the prediction back into the PDE and checks how well it satisfies the equation.
If it fits the PDE well, the loss is low. If not, the loss is high — and it tells the network:
"Here's how far off you are." The optimizer then takes that error signal, computes gradients using automatic differentiation, and updates the network parameters accordingly.
This forms a training loop:
→ make a prediction
→ check if it fits the PDE
→ adjust the network
→ and try again.

After enough training, the network learns a function that closely satisfies the PDE across the domain — and that becomes our final solution.

5.
Now that we've seen the overall training loop, let's zoom in on how each part is tailored for solving financial PDEs like the Feynman-Kac equation.
DGM draws inspiration from the classical Galerkin method.But instead of fixed basis functions, DGM uses a deep neural network to represent the solution. The training process enforces the PDE and boundary conditions directly — by minimizing a carefully designed loss function at randomly sampled points in space and time.

This brings us to the first component: the input sampler.
DGM uses a mesh-free sampling strategy — meaning we don't define a fixed spatial grid.
Instead, we randomly sample points across the entire input domain, like [t,S], in this case.
This approach avoids the curse of dimensionality and turns the PDE into a machine learning problem. It's one of the most important ideas behind DGM, and we'll come back to it later to see exactly how we generate those random points.

Network:
The next component is the network.
The architecture of a neural network is important to its success. The network we use in DGM is inspired by long short-term networks or LSTM which are typically used in time-series forecasting because they can model long-range dependencies. DGM network borrows ideas like gating and skip connections to help the network train more stably and capture complex time dynamics, which are especially important for financial PDEs.
This is also a **much deeper network** than usual. We use 24 hidden layers with 64 neurons per layer. In most standard machine learning tasks—like image classification or sentiment analysis use less than 10 layers can get good results. The reason for going this deep is to have greater capacity to approximate the solutions, since financial PDEs are often highly nonlinear and time-dependent.
We also tried simpler networks, like MLPs, but they didn't converge well and suffered from vanishing gradients.
In practice, the DGM structure worked well for both PDEs, it is a general and flexible setup.

Next is the loss function.
Unlike typical machine learning setups, we're not training the model to match a set of known answers. Instead, we want the neural network itself to become the solution to the PDE.
So we design the loss function directly from the PDE — including its differential operator, terminal condition, and boundary conditions. In other words, the loss encodes all the rules the solution needs to follow.
It is especially helpful in high-dimensional problems, where analytical solutions don't exist and traditional methods like finite difference break down,the loss value gives us an easy way to track how well the model is doing
Once we have the loss, we use the ADAM optimizer to train the network. ADAM is a gradient-based optimization algorithm that's widely used in deep learning.
It takes the loss value, computes the gradients, and updates the weights and bias in the network.
One powerful thing about ADAM is that it automatically adjusts the learning rate for each parameter, if a parameter is close to convergence, it slows down. If it's far off, it speeds up. That makes training both faster and more stable.
We rely on PyTorch's auto-differentiation to compute the gradients efficiently. That's what makes the whole training loop — forward pass, loss computation, backward pass — fully automatic.
And this whole process repeats until the loss converges

6. INPUT
Let's go back to inputs and take a closer look at where the training points come from.
We only generate training inputs from three key regions: each one used to enforce a different part of the PDE
As u can see from the graph,
The blue points cover the interior domain where we want the network to learn the overall solution shape.
The red points are sampled at the final time, ensure the network output matches known payoffs at maturity
The green points sampled at the spatial edges Smin and Smax. These help enforce boundary conditions like vanishing option value or linear behavior as price grows.

We don't sample time points beyond maturity, or spot levels far outside the range because there's no payoff, no boundary, no PDE defined there. These sampled regions define the space where the model learns the solution surface. So the model only knows how the function behaves within this domain.
Anything outside — it's not just uncertain, it's undefined.

7. LOSS
Now let's look at how the loss function is actually set up when we solve the Black-Scholes PDE.
The core idea is to define a loss term for each group of points we just talked about, based on the equation or condition it needs to satisfy.
The PDE loss handles the interior points — it checks whether the prediction satisfies the Black-Scholes equation across the surface. This shapes the overall structure.
The boundary loss focuses on the top and bottom edges — when the stock price is very low or very high — and keeps the model's behavior reasonable at the extremes.
And the terminal loss pins down the surface surface at the final time, making sure the network matches the option payoff at maturity.
So all three losses work together, guide sampled points to form a surface which is the PDE solution surface for faymankac's black scholes.
8. DGM BS Benchmark
Here's how well the network performs
You can see that DGM recovers the full pricing surface — smooth and accurate.
Across different time slices, the predicted values stay consistent with the analytical solution.
And because the network is fully differentiable, we can extract Greeks like delta directly using auto-diff — and they match closely with the true values.
So the model isn't just fitting prices — it's learning the full structure and sensitivities, which is key for practical use.

9. Generalized DGM BS
This is our generalized setup for the Black-Scholes model.
All key parameters — time, spot, strike, interest rate, and volatility — are now inputs to the network. There are 5 in total. The architecture remains the same and the loss structure still follows the same components. By feeding in all relevant parameters, the model can now learn to price a wide range of market trades. This generalization allows us to use a single model to capture the full pricing surface across different contracts and market conditions.
10. **Generalized DGM BS Result**
Here we evaluate the model under different market settings.
The model handles different market conditions well — even with changing rates and volatilities, predictions remain accurate. So we get reliable results without retraining.

11. Generalized DGM SABR
Now we move to the SABR model.
The only thing we changed here is the implementation of the Loss Function
The PDE loss for SABR is based on its backward equation.
The terminal loss still match the option payoff at maturity
And the boundary loss are still the edges of the domain.
We don't need to change the network architecture — just update the equation and adjust the sampling strategy.
This flexibility means we can reuse the same codebase and model design across different stochastic models, which makes it easier to extend to more complex products without rebuilding everything from scratch.

12. Generalized DGM SABR Result
Our generalized DGM replicates Hagan's Normal pricing extremely well.
Both at time zero and across future time steps.
As you can see from the plots —
in the top row, we test at t=0 under different SABR parameters,
and in the bottom row, we vary t over time — the model predictions remain nearly indistinguishable.
This gives us confidence that the network captures both the initial price surface and its time evolution correctly.

13. DGM - Neural Network Summary

This table is a summary and comparison of our Black-Scholes and SABR models.

14. DGM - Fokker Planck General

Now we turn to solving forward PDEs the Fokker-Planck equation.
At first, I tried applying the same DGM for backward PDEs.
But the challenge here is the initial condition: it's a delta function — a sharp spike concentrated at a single point — which is hard for neural networks to approximate.
I tried using a Gaussian approximation, but training was unstable and couldn't capture the sharpness effectively.
So I started looking into alternatives and came across a recent paper published in Management Science in April 2025.
It proposed a clever idea: instead of learning the PDF directly, we train the model to learn the CDF, and then recover the PDF by differentiation.
The trick here is the CDF satisfies a backward Kolmogorov equation, even though we're solving a forward problem. So we turned a forward PDE into a backward one — which is exactly what DGM is built to handle.
The terminal condition under this formulation:
At time T, the process has already landed at a known value x.
So the CDF becomes a step function — it's 1 if x is less than or equal to y, and 0 otherwise.
Once the network learns the CDF, we simply differentiate it with respect to y to recover the full transition density.

15. DGM - Fokker Planck BS

The only change from the standard DGM setup is the input — we add a Target Value. That's because we're now learning the cumulative probability that the asset price ends up below a target value at maturity. To do that, the model needs to know what that target is, so we include it as part of the input. Everything else — the network architecture, the optimizer, and the training process — stays exactly the same. And once training is done, we can get the probability density by taking the derivative of the output with respect to this new input.

16. Benchmark 1

As a quick benchmark, we compare the model output to the analytical solution at two time steps — one very early and one much later.
As you can see, the model aligns very closely with both the CDF and the PDF at each point, which shows that it's robust over time. We're not just solving for one price, we're recovering the entire distribution of outcomes, which gives us much richer information about market uncertainty.

17. Benchmark 2

Here's the full joint distribution.On the left is the DGM result, and on the right is the analytical solution.
As you can see, the shapes match really well across the entire surface. So this confirms that DGM doesn't just work at a few points — it recovers the full evolution of the density over time.

18. DGM - Fokker Planck SABR

Just like with Black-Scholes, we can apply the same DGM framework to the SABR model.
The network architecture stays the same — what changes are the inputs and the loss again
This time, the input includes forward price, instantaneous volatility, and all SABR parameters: t, F, α (alpha, 阿尔法),ρ (rou),β (beta 贝塔),ν (nu, 妞儿) as well as the target F and alpha .
The loss function is updated to reflect the backward Kolmogorov PDE under SABR, a new terminal condition,
Once the model learns this CDF, we can differentiate it to recover the full transition density under SABR.
So once again, we didn't change the network — we just swapped in the right ingredients for this model.

19. Learned CDF of Joint Distribution

This is the joint CDF the model learned under SABR.This surface is a direct output from the trained network — no post-processing involved. You can see it transitions smoothly from 0 to 1 over the 2D domain,
which confirms the model has captured the joint behavior of forward and volatility.

20. Joint PDF

What you see here is the evolution of the joint PDF under the SABR dynamics — from initial time to later snapshots.
As time progresses, the density gradually spreads out, becomes more skewed, and eventually stabilizes.
Again, this is fully learned — no simulation is involved — all snapshots come from the network.

21. Comparison with MC
I also compare it with a standard Monte Carlo simulation to check the accuracy.
On the left is the MC estimate, in the middle is the DGM-based output,
and on the right is their pointwise difference.
As you can see, the network not only captures the dynamics,
but also aligns very closely in shape and location of the probability mass.
This agreement is entirely learned from the PDE.DGM output was trained without any simulation data
22. Summary
This table is a quick summary setting of our Black-Scholes and SABR models for Fokker Planck Equation.

23. Method I Implemented
So far I've shown you the method that ended up working best — but DGM actually wasn't where I started. We began with Deep BSDE, a popular approach for solving backward PDEs published in 2018. It gave decent results at time zero, and even allowed us to estimate delta through gradients.
But it had a serious limitation: it could only predict the option price at t = 0, and had to be retrained for each payoff or parameter. There was no way to recover the full pricing surface across time — so we pivoted.
That's when I turned to DGM. It was much more powerful: it captured the entire solution surface, supported different payoffs, and even gave us Greeks through auto-diff.
But — it didn't perform well when applied to the forward PDE form, like the Fokker-Planck equation.
After reading a few papers, I decided to try a sliced Mixture Density Network — a method that first appeared in the physics community and was later explored in 2020. I adapted it to fit our setting.
This approach fits the density at each time step using Monte Carlo reference points — it smoothed out the noise and helped us visualize the dynamics more clearly.

It worked — but it wasn't scalable. We needed one network per time step, and it didn't extend well to high dimensions.

So I went back to the literature, read through recent papers, and eventually found a way to adapt DGM for forward PDEs — by switching the output from density to CDF.
This let us recover the entire transition density from a single model, fully learned from the PDE, with no simulation data required.

24. Challenges
All mothers exhibit common challenges with deep learning technique.
First, the model doesn't extrapolate well — it only learns within the training region.
Second, training can be sensitive — small tweaks to learning rate, batch size, or even initialization can change how well it converges.
Interpretability is still an issue — the network gives good results, but we don't always know what it's actually learning.
And finally, the computational cost can be high — especially when the model gets deeper and the sample size grows.