Neural Network Applications in Financial PDEs

Adam Thorne

March 2026

Contents

Chapter 1

Set Up: Finance and Stochastic Methods

1.1 Introduction to Financial Contracts

1.1.1 Long v.s. Short

In finance, **long** and **short** describe the direction of a position, whether you profit when the price goes up or when it goes down.

- **Long**: you own the asset (or the right to buy it). You profit if the price *rises*.
  - Example: You buy 100 shares of XYZ at \$50. If the price rises to \$60, you profit \$1,000.
- **Short**: you borrow an asset you do not own, sell it immediately at the current price, and are obligated to buy it back later and return it to the lender. You profit if the price *falls*, because you can buy it back cheaper than you sold it.

How shorting works step by step.

- **Step 1: Borrow.** You borrow 100 shares of XYZ from a broker. The broker lends you shares it holds on behalf of other clients.
- **Step 2: Sell.** You immediately sell those 100 shares on the market at the current price of \$50, receiving $100 \times \$50 = \$5{,}000$ in cash. You do not own the shares anymore; you owe them to the broker.
- **Step 3: Wait.** You hold the \$5,000 cash and wait for the price to fall.
- **Step 4: Buy back ("cover").** Suppose the price falls to \$40. You buy 100 shares back on the market for $100 \times \$40 = \$4{,}000$ and return them to the broker.
- **Profit:** $\$5{,}000 - \$4{,}000 = \$1{,}000$ (minus any borrowing fees charged by the broker).

1.1.2 Calls and Puts

An **option** is a financial contract between a buyer and a seller. The buyer pays an upfront *premium* (denoted by C) to obtain the *right*, but not the obligation, to trade an underlying asset at a set *strike price* K on or before an *expiration date* T. The seller (or *writer*) receives the premium and is obligated to honor the trade if the buyer chooses to exercise the option.

Call: A call option gives the holder the right to *buy* the underlying asset at the strike price K.

Put: A put option gives the holder the right to *sell* the underlying asset at the strike price K.

1.1.3 Selling Options

Selling an option is the opposite of buying it. The seller receives the premium and is obligated to honor the trade if the buyer chooses to exercise the option. The seller profits from the premium received, but risks unlimited losses if the option is exercised against them.

Call: The seller of a call option profits from the premium received, but risks unlimited losses if the option is exercised against them.

Put: The seller of a put option profits from the premium received, but risks unlimited losses if the option is exercised against them.

Why would anyone sell an option? Despite the risk, selling options is attractive because the seller collects the premium upfront and keeps it entirely if the option expires worthless. In fact, most options do expire worthless: if the stock never crosses the strike price, the buyer has no reason to exercise and the seller walks away with the full premium as pure profit.

How do sellers hedge their risk, so they don't go broke? The seller of an option faces potentially large losses if the market moves against them. To manage this risk, sellers use a strategy called *delta hedging*. The idea is to hold a position in the underlying stock that offsets the risk of the option. I will cover an example of this in section **1.1.5**.

1.1.4 European v.s. American Options

- **European options:** Can only be exercised at the fixed expiration date ⇒ fixed boundary condition. Less flexibility ⇒ lower price.
- **American options:** Can be exercised at any time before expiration ⇒ free boundary. More flexibility ⇒ higher price.

1.1.5 Examples of Options Contracts

Each contract below covers 100 shares. One contract = 100× the per-option price.

Sell a Call

- **Setup:** $S_0 = \$100$, $K = \$105$, $T = 1$ month, premium $C = \$3$
- **Harnur collects:** $100 \times \$3 = \$300$ upfront
- **Risk:** If XYZ rises above \$105, Harnur must sell 100 shares at \$105 no matter how high the price goes
- **Hedge:** Harnur buys 40 shares of XYZ at \$100. If the stock rises, her shares rise in value, covering her obligation
- **If $S_T = \$110$:**
  - Option costs her: $(110 - 105) \times 100 = \$500$
  - Her 40 shares gained: $40 \times \$10 = \$400$
  - Premium collected: +\$300
  - **Net PnL:** $-\$500 + \$400 + \$300 = +\$200$
- **If $S_T = \$100$ (expires worthless):** Harnur keeps the full \$300 premium

Sell a Put.

- **Setup:** $S_0 = \$100$, $K = \$95$, $T = 1$ month, premium $C = \$3$
- **Lucas collects:** $100 \times \$3 = \$300$ upfront
- **Risk:** If XYZ falls below \$95, Lucas must buy 100 shares at \$95 no matter how low the price goes
- **Hedge:** Lucas *short sells* 35 shares of XYZ at \$100. This guarentees that if the stock drops, he can cover his obligation.
- **If $S_T = \$85$:**
  - Option costs him: $(95 - 85) \times 100 = \$1{,}000$
  - His short position gained: $35 \times \$15 = \$525$
  - Premium collected: +\$300
  - **Net PnL:** $-\$1{,}000 + \$525 + \$300 = -\$175$
- **If $S_T = \$100$ (expires worthless):** Lucas keeps the full \$300 premium

Buy a Call.

- **Setup:** $S_0 = \$100$, $K = \$105$, $T = 1$ month, premium $C = \$3$
- **Clara pays:** $100 \times \$3 = \$300$ upfront
- **If $S_T = \$115$:** Contract worth $= (115 - 105) \times 100 = \$1{,}000$, **PnL** $= \$1{,}000 - \$300 = +\$700$
- **If $S_T \leq \$105$ (expires worthless):** Clara loses only her \$300 premium

Buy a Put.

- **Setup:** $S_0 = \$100$, $K = \$95$, $T = 1$ month, premium $C = \$3$
- **David pays:** $100 \times \$3 = \$300$ upfront
- **If $S_T = \$80$:** Contract worth $= (95 - 80) \times 100 = \$1{,}500$, **PnL** $= \$1{,}500 - \$300 = +\$1{,}200$
- **If $S_T \geq \$95$ (expires worthless):** David loses only his \$300 premium

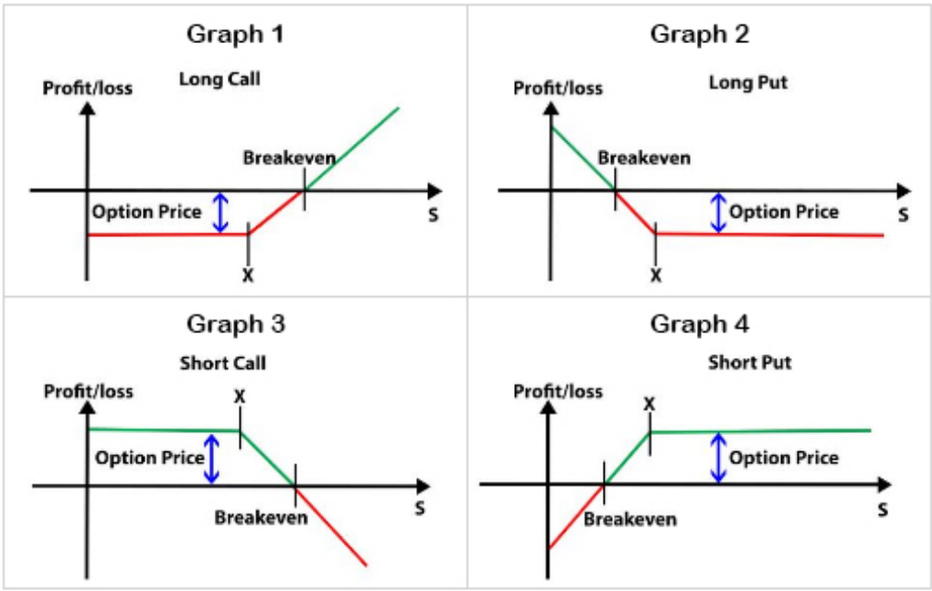

Figure 1.1: PnL Graphs for Calls and Puts

1.1.6 How do investors use options?

Options let investors control financial risk and opportunity by locking in the right to buy or sell an asset at a predetermined price in the future. Three main uses are **leverage**, **hedging**, and **non-linear payoffs**.

Leverage. Control more shares with less capital by buying calls instead of stock.

- **Setup:** $S_0 = \$100$, call premium $C = \$6$, one contract $= 100$ shares
- **Bruno** buys 100 shares outright: costs \$10,000
- **Thomas** buys 1 call contract (controls 100 shares): costs \$600
- **If $S_T = \$0$:**
  - Bruno loses \$10,000
  - Thomas loses only \$600 (his maximum loss is always capped at the premium)
- **Key idea:** Options amplify upside exposure for a fraction of the capital

Hedging. Use puts to insure an existing stock position against a crash.

- **Setup:** Own 100 shares at $S_0 = \$100$, buy 1 put contract with $K = \$95$, $C = \$4$
- **Cost of insurance:** $100 \times \$4 = \$400$
- **If $S_T = \$0$:**
  - Shares lose: \$10,000
  - Puts pay off: $(95 - 0) \times 100 = \$9{,}500$
  - Net from puts after premium: $\$9{,}500 - \$400 = +\$9{,}100$
  - **Combined PnL:** $-\$10{,}000 + \$9{,}100 = -\$900$ (vs. $-\$10{,}000$ unhedged)
- **Key idea:** Puts act like insurance, small premium limits the downside

Non-linear payoffs. Options can deliver much higher percentage returns than stock for the same directional view.

- **Setup:** Both Fischer and Myron expect XYZ to rise from $S_0 = \$100$
- **Fischer** buys 100 shares: costs \$10,000
- **Myron** buys 1 call contract: $K = \$100$, $C = \$5$, costs \$500
- **If** $S_T = \$120$**:**
  - Fischer's PnL: $100 \times \$20 = +\$2{,}000$ (+20% on \$10,000)
  - Myron's PnL: $(20 - 5) \times 100 = +\$1{,}500$ (+300% on \$500)
- **Key idea:** Options offer asymmetric, non-linear returns: large gains for a small outlay

1.2 Random Processes

1.2.1 Random Walk

To simulate a stock price, we can start from a completely random process. Define a stock price M at step k by

$$M_k = \sum_{n=1}^{k} X_n,$$

where

$$X_n = \begin{cases} 1, & \text{with probability } \frac{1}{2} \\ -1, & \text{with probability } \frac{1}{2} \end{cases}.$$

So at each step the price moves up or down by a fixed amount with equal probability. Properties of this model include:

- Time moves in discrete steps ($n = 1, 2, 3, \ldots$).
- Jumps are fixed size (each X_n is ± 1).
- Independent increments: each X_n is independent of the others.
- Variance grows linearly: $\mathrm{Var}(M_n) = n$.

This process is **not** differentiable at the time steps, which is an important limitation when moving toward continuous-time models such as Brownian motion.

1.2.2 Scaled Random Walk

We fix the discrete random walk by making time more continuous. Let $k = nt$ and scale the process down by a factor of $1/\sqrt{n}$. Define

$$W^{(n)}(t) = \frac{1}{\sqrt{n}} M_{nt}.$$

This fits more steps into the same time interval. The scaling by $1/\sqrt{n}$ is chosen so that the variance is preserved: $\mathrm{Var}[W^{(n)}(t)] = t$.

1.2.3 Brownian Motion

Taking $n \to \infty$ yields a continuous-time process: Brownian motion $W(t)$. We can write it in terms of increments, e.g.,

$$W(t_1) = W(t_1) - W(t_0), \quad W(t_2) - W(t_1), \quad \ldots, \quad W(t_m) - W(t_{m-1}).$$

By the Central Limit Theorem, not only is $W(t)$ normally distributed for every t, but every increment $W(s) - W(t)$ is also normal (with mean zero and variance t for standard Brownian motion). A key fact: Brownian motion is **not** differentiable anywhere.

1.2.4 Geometric Brownian Motion

Plain Brownian motion is not how stocks move. If we set $S_t = W_t$, then $dS(t) = dW(t)$, but this ignores that stock price changes are *proportional* to the price. A more realistic model is

$$dS(t) = S(t)\,dW(t).$$

This is **geometric Brownian motion**. Stocks also differ in how much they move: some are much more volatile. We capture that with a *volatility* parameter σ:

$$dS(t) = \sigma S(t)\,dW(t).$$

Finally, the average movement of a stock is not zero; over time prices tend to drift in a direction (e.g., certain sectors may trend up or down). Adding a *drift* term α gives the standard model

$$dS(t) = \alpha S(t)\,dt + \sigma S(t)\,dW(t),$$

where α is the drift and σ is the volatility. This stochastic differential equation is the basis for models such as Black-Scholes, this is called an *Itô Process*.

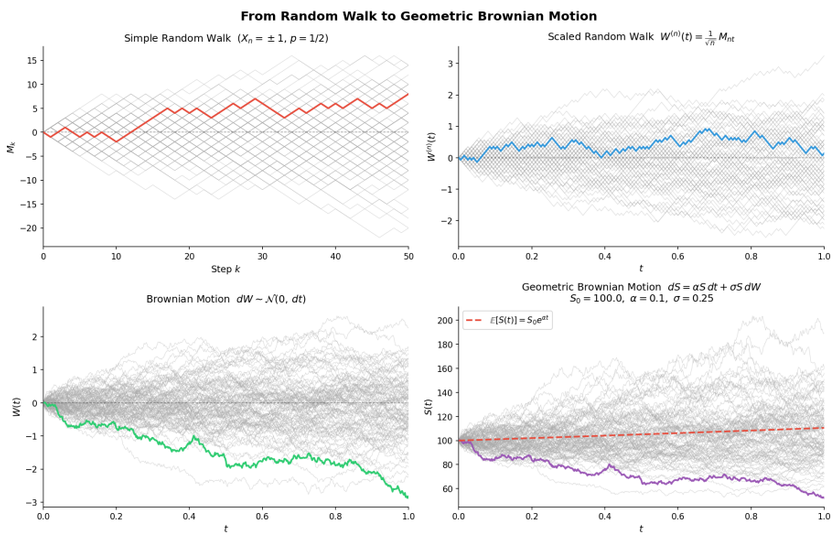

Figure 1.2: Stochastic Processes

1.3 Stochastic Calculus

Stochastic calculus provides the tools to work with random processes that evolve in time, such as stock prices modeled by geometric Brownian motion [11]. Because these paths are not differentiable, the usual rules of calculus are replaced by Itô's calculus, which allows us to formulate and solve equations involving both deterministic drift and random diffusion.

1.3.1 The Basics

Suppose the portfolio value at time t is $X(t)$. Part of the wealth is in a bank account earning a constant interest rate r, and the rest is in a stock whose price follows the GBM

$$dS(t) = \mu S(t)\,dt + \sigma S(t)\,dW(t).$$

Let $\Delta(t)$ denote the number of shares of stock held at time t. The change in the portfolio value is then

$$\begin{aligned} dX(t) &= \Delta(t)\,dS(t) + r\big(X(t) - \Delta(t)S(t)\big)\,dt \\ &= \Delta(t)\big(\mu S(t)\,dt + \sigma S(t)\,dW(t)\big) + r\big(X(t) - \Delta(t)S(t)\big)\,dt \\ &= \underbrace{rX(t)\,dt}_{\text{(i)}} + \underbrace{\Delta(t)(\mu - r)S(t)\,dt}_{\text{(ii)}} + \underbrace{\Delta(t)\sigma S(t)\,dW(t)}_{\text{(iii)}}. \end{aligned} \tag{1.1}$$

Where:

(i) Average rate of return r on the entire portfolio (risk-free growth).

(ii) Risk premium $\mu - r$ from investing in the stock.

(iii) Volatility term proportional to the stock holding.

1.3.2 Itô's Lemma

Gaining Some Intuition In standard calculus, if we have a function $f(t, X)$ and want to find its differential df, we use the chain rule (a first-order Taylor expansion):

$$df = \frac{\partial f}{\partial t}\, dt + \frac{\partial f}{\partial X}\, dX.$$

This works because any smooth function looks like a straight line when you zoom in far enough, the first-order term captures it exactly. The problem with stochastic processes is that this is never true: no matter how small the time increment, the path remains jagged and cannot be linearised. The first-order expansion misses a correction term, and that is precisely what Itô's lemma supplies.

So how can we fix this? We must add the second derivative so we can catch *all* parts of the stochastic part. Thus we get:

$$df = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial X}dX + \frac{1}{2}\frac{\partial^2 f}{\partial t^2}(dt)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial X^2}(dX)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial t\, \partial X}dtdX + \frac{1}{2}\frac{\partial^2 f}{\partial X\, \partial t}dXdt$$

$$df = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial X}dX + \frac{1}{2}\frac{\partial^2 f}{\partial t^2}(dt)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial X^2}(dX)^2 + \frac{\partial^2 f}{\partial t\, \partial X}dtdX$$

From standard calculus $(dt)^2 \to 0$. Looking back to when Brownian Motion was introduced, importantly it's variance is t. Using this we can see that $(dW)^2 \to dt$ because dW is a small change in W, and variance is a measure of squared difference. While this is not a rigorous proof, I find it to be more intuitive. Finally, $dt\, dW = dt\sqrt{dt} = dt^{1.5} \to 0$.

Finding the $(dX)^2$ term Since X is an Itô Process, $dX = \alpha dt + \sigma dW$, thus

$$(dX)^2 = (\alpha dt + \sigma dW)^2 = \alpha^2 dt^2 + 2\alpha\sigma dt dW + \sigma^2 dW^2$$

Using what we just saw

$$(dX)^2 = \sigma^2 dt$$

Then plugging this back in above we arrive at Itô's lemma

$$\boxed{df = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial X}(\alpha dt + \sigma dW) + \frac{1}{2}\frac{\partial^2 f}{\partial X^2}(\sigma^2 dt)}$$

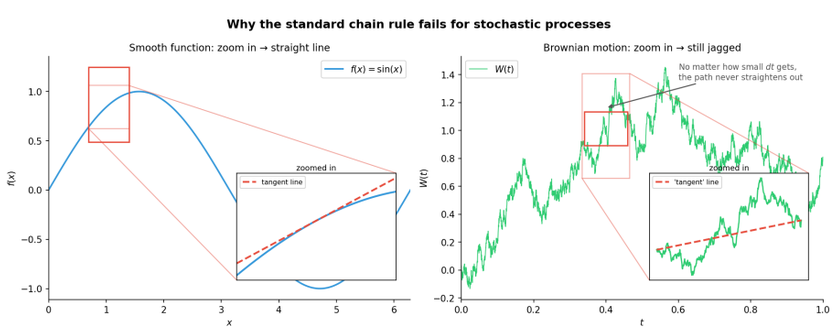

Figure 1.3: Standard v.s. Stochastic Calculus

Chapter 2

Option Pricing Frameworks

2.1 The Black-Scholes Framework

The Black-Scholes framework [1] is a cornerstone of mathematical finance. It provides a closed-form formula and a partial differential equation for the fair value of a European option under a set of ideal market conditions. The key idea is to replicate the option payoff with a continuously rebalanced portfolio of the underlying stock and a risk-free bond, leading to a deterministic PDE that does not depend on the drift of the stock.

2.1.1 Introduction and Background

We consider the value $V(S,t)$ of an option as a function of the stock price S and time t. The stock is assumed to follow the geometric Brownian motion $dS(t) = \alpha S(t)\,dt + \sigma S(t)\,dW(t)$ introduced earlier. The derivation below shows how a perfectly hedged portfolio and Itô's lemma lead to the famous Black-Scholes PDE.

2.1.2 Assumptions

The Black-Scholes valuation formula relies on the following ideal conditions:

(a) The short-term interest rate r is known and constant through time.

(b) The stock price follows a random walk in continuous time with a variance rate proportional to the square of the stock price. Thus the distribution of possible stock prices at the end of any finite interval is log-normal, and the variance rate of the return on the stock is constant.

(c) The stock pays no dividends or other distributions.

(d) The option is European (exercisable only at maturity).

(e) There are no transaction costs in buying or selling the stock or the option.

(f) It is possible to borrow any fraction of the price of a security to buy or hold it at the short-term interest rate.

(g) There are no penalties for short selling. A seller who does not own the security receives the price from the buyer and agrees to settle on a future date by paying an amount equal to the security's price on that date.

2.1.3 Deriving the PDE

Construct a perfectly hedged portfolio: long one option and short Δ shares of stock,

$$X = V(S,t) - \Delta S.$$

We want to see how X changes over time and what risks appear. The change in the portfolio is

$$dX = dV - \Delta\, dS.$$

By Itô's lemma,

$$dV = \frac{\partial V}{\partial t}\,dt + \frac{\partial V}{\partial S}\,dS + \frac{1}{2}\,\frac{\partial^2 V}{\partial S^2}\,(dS)^2.$$

Substitute $dS(t) = \alpha S(t)\,dt + \sigma S(t)\,dW(t)$ into $(dS)^2$. Using $dt \cdot dt = 0$, $dt \cdot dW = 0$, and $dW \cdot dW = dt$, only the $\sigma^2 S^2\,dt$ term remains, so

$$dV = \frac{\partial V}{\partial t}\,dt + \frac{\partial V}{\partial S}\,dS + \frac{1}{2}\,\sigma^2 S^2\,\frac{\partial^2 V}{\partial S^2}\,dt.$$

Plugging this into dX and grouping terms,

$$dX = \left(\frac{\partial V}{\partial t} + \frac{1}{2}\,\sigma^2 S^2\,\frac{\partial^2 V}{\partial S^2}\right) dt + \left(\frac{\partial V}{\partial S} - \Delta\right) dS.$$

The "delta hedge": choose $\Delta = \partial V/\partial S$. Then the dS term vanishes,

$$dX = \left(\frac{\partial V}{\partial t} + \frac{1}{2}\,\sigma^2 S^2\,\frac{\partial^2 V}{\partial S^2}\right) dt.$$

The portfolio is now deterministic and risk-free, so it must earn the risk-free rate: $dX = r(V - \Delta S)\,dt = r\big(V - S\,\frac{\partial V}{\partial S}\big)\,dt$. Equating the two expressions for dX and cancelling dt,

$$\frac{\partial V}{\partial t} + \frac{1}{2}\,\sigma^2 S^2\,\frac{\partial^2 V}{\partial S^2} = rV - rS\,\frac{\partial V}{\partial S}.$$

Rearranging yields the Black-Scholes PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\,\sigma^2 S^2\,\frac{\partial^2 V}{\partial S^2} + rS\,\frac{\partial V}{\partial S} - rV = 0.$$

2.1.4 Solving the PDE for Option Price

The boundary condition

The Black-Scholes PDE must be solved subject to a *boundary condition*: a rule specifying the value of the option at expiration. For a European call with strike K and maturity T, the payoff at expiration is

$$V(S,T) = \max(S - K,\ 0).$$

The holder exercises only if the stock price S finishes above the strike K; otherwise the option expires worthless.

Change of variables: reducing to the heat equation

The Black-Scholes PDE,

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0,$$

can be transformed into the classical *heat equation* through a change of variables. Let $k = 2r/\sigma^2$ and introduce:

$$S = Ke^x, \qquad \tau = \tfrac{1}{2}\sigma^2(T-t), \qquad V(S,t) = K\,e^{-\frac{1}{2}(k-1)x - \frac{1}{4}(k+1)^2\tau}\,u(x,\tau).$$

Here $x = \ln(S/K)$ is the log-moneyness (how far the stock is from the strike, in log-scale), τ is a rescaled time-to-expiration (runs forward from 0 to $\frac{1}{2}\sigma^2 T$ as the option approaches expiry), and the exponential prefactor removes the drift and discounting terms. Substituting into the PDE and simplifying, all terms involving r and the mixed x-τ derivatives cancel, leaving the standard heat equation:

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2}.$$

Transforming the boundary condition

Under the substitution above, the call payoff $V(S,T) = \max(S-K,0)$ at $t = T$ (i.e. $\tau = 0$) becomes the initial condition for u:

$$u(x,0) = \max\Big(e^{\frac{1}{2}(k+1)x} - e^{\frac{1}{2}(k-1)x},\ 0\Big).$$

This is a one-sided exponential: it equals $e^{\frac{1}{2}(k+1)x} - e^{\frac{1}{2}(k-1)x}$ when $x > 0$ (stock above strike) and 0 when $x \le 0$.

Solving via Standard Solution

The heat equation $\partial u/\partial\tau = \partial^2 u/\partial x^2$ on the whole real line has the well-known solution via convolution with the Gaussian kernel. Given initial data $u(x,0) = u_0(x)$, the solution at time $\tau > 0$ is:

$$u(x,\tau) = \frac{1}{2\sqrt{\pi\tau}}\int_{-\infty}^{\infty} u_0(s)\,e^{-\frac{(x-s)^2}{4\tau}}\,ds.$$

Substituting our initial condition $u_0(s) = \max(e^{\frac{1}{2}(k+1)s} - e^{\frac{1}{2}(k-1)s}, 0)$ and splitting into two integrals (one for each exponential term):

$$u(x,\tau) = \frac{1}{2\sqrt{\pi\tau}}\int_0^{\infty} e^{\frac{1}{2}(k+1)s}e^{-\frac{(x-s)^2}{4\tau}}\,ds \;-\; \frac{1}{2\sqrt{\pi\tau}}\int_0^{\infty} e^{\frac{1}{2}(k-1)s}e^{-\frac{(x-s)^2}{4\tau}}\,ds.$$

Each integral has the form $\int_0^\infty e^{as} e^{-(x-s)^2/(4\tau)}\, ds$, which is evaluated by completing the square in s inside the exponent:

$$as - \frac{(x-s)^2}{4\tau} = -\frac{1}{4\tau}(s - (x + 2a\tau))^2 + ax + a^2\tau.$$

After completing the square the integral over s becomes a Gaussian integral over a half-line, which evaluates to $\sqrt{\pi\tau}\, e^{ax+a^2\tau}\, N\Big(\frac{x+2a\tau}{\sqrt{2\tau}}\Big)$ where N is the standard normal CDF. Applying this to each term with $a = \frac{1}{2}(k+1)$ and $a = \frac{1}{2}(k-1)$ respectively, then undoing the change of variables ($x = \ln(S/K)$, $\tau = \frac{1}{2}\sigma^2(T-t)$, removing the exponential prefactor) produces the Black-Scholes formula.

The Black-Scholes formula

Undoing the change of variables and simplifying, the unique solution gives the price of a European call option as [1]:

$$C = S\,N(d_1) - Ke^{-rT}N(d_2),$$

where

$$d_1 = \frac{\ln(S/K) + \big(r + \frac{1}{2}\sigma^2\big)T}{\sigma\sqrt{T}}, \qquad d_2 = d_1 - \sigma\sqrt{T}.$$

Each piece has a concrete meaning:

- $N(\cdot)$ is the cumulative distribution function of the standard normal distribution: $N(d)$ gives the probability that a standard normal random variable is less than d. It enters the formula because the stock price at expiration is log-normally distributed.
- $N(d_2)$ is the (risk-neutral) probability that the option will be exercised, i.e. that $S_T > K$ at expiration.
- Ke^{-rT} is the present value of the strike price: the amount to be paid at expiration, discounted back to today at the risk-free rate r.
- $S\,N(d_1)$ is the present value of receiving the stock conditional on the option being exercised.
- The formula says: the call is worth the expected value of the stock you will receive minus the expected cost of paying the strike, both under the risk-neutral measure.
- Crucially, the formula does not depend on the expected return α of the stock. The drift cancels out during the delta-hedging argument; only the volatility σ and risk-free rate r determine the price.

European put formula

For a European put (right to sell at strike K), the boundary condition at expiration is $V(S,T) = \max(K-S, 0)$. The Black-Scholes PDE is the same; only the boundary condition changes. Using the relationship between call and put prices (put-call parity: buying a call and selling a put replicates owning the stock on margin), the put price is:

$$P = Ke^{-rT}N(-d_2) - S\,N(-d_1),$$

where d_1 and d_2 are defined identically to the call formula. Here $N(-d_2)$ is the risk-neutral probability that $S_T < K$ (i.e. the put is exercised), and $N(-d_1) = 1 - N(d_1)$.

2.1.5 Addressing its Limitations

The Black-Scholes model assumes that volatility σ is constant for all strikes K and maturities, which does not match real markets [4]. In practice, implied volatility across strike K often forms a "smile": high for deeply out-of-the-money puts, lower near the money, and high again for out-of-the-money calls. This limitation is addressed by the local volatility framework of Dupire, discussed in the next section.

2.2 Dupire Equation

2.2.1 Introduction and Background

In Black-Scholes, a single number σ describes how much the stock price fluctuates. But when we look at the market prices of options across all strikes K (the agreed buy/sell price) and maturities T (when the option expires), and back out what constant σ each option is implying, we do not get the same number. This is the *implied volatility smile*: implied volatility is high for deep out-of-the-money and in-the-money options, and lower for at-the-money options.

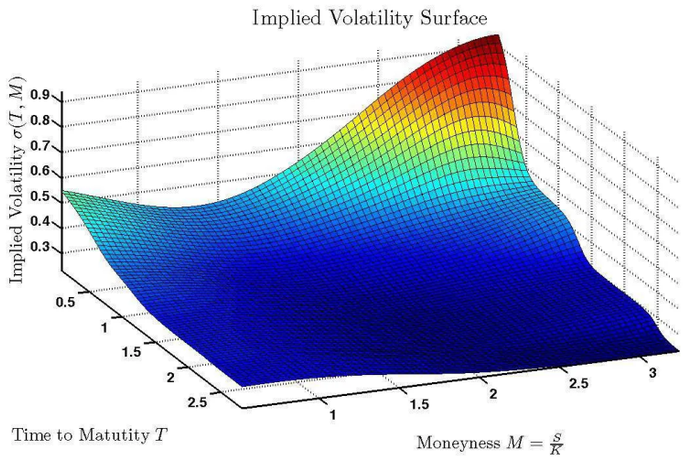

Figure 2.1: Volatility smile.

The existence of the smile creates a logical contradiction. If Black-Scholes were the correct model, there would be one true σ for the stock and every option would imply the same value. In practice, a six-month option might imply $\sigma = 20\%$ while a one-year option implies $\sigma = 18\%$. This is nonsensical: the stock cannot simultaneously have a constant volatility of both 20% and 18%.

Bruno Dupire [3] resolved this by asking: instead of forcing σ to be constant, can we find a single *process* for the stock price that is consistent with *all* observed option prices at once, while keeping the model mathematically complete (meaning every risk can be perfectly hedged)? His answer was yes, provided we allow σ to be a deterministic function of the current stock price S and time t, written $\sigma(S, t)$. This is called *local volatility*.

2.2.2 Assumptions and Setup

Before deriving the equation, we state the key assumptions and define all terms.

- The **stock price** S follows a diffusion process (a continuous random process with no sudden jumps):
$$\frac{dS}{S} = r(t)\,dt + \sigma(S, t)\,dW$$
where $r(t)$ is the instantaneous risk-free interest rate (the rate earned on a perfectly safe investment at each instant), $\sigma(S, t)$ is the local volatility we want to find, and dW is a standard Brownian motion increment (the random shock).

- The model is **complete**: since the only source of randomness is the single Brownian motion dW, every option payoff can be replicated by trading the stock, and prices are uniquely determined by no-arbitrage.

- We are given the market prices $C(K, T)$ of **European call options** for all strikes $K > 0$ and maturities $T > 0$. A European call pays $\max(S_T - K, 0)$ at expiration: the holder receives the stock price minus the strike if the stock finishes above K, and nothing otherwise.

- For clarity of the derivation, we follow Dupire and set $r = 0$ (zero interest rate). The full version with non-zero r adds one extra term but the logic is identical.

2.2.3 Deriving the PDE

Step 1: From option prices to probability densities

The price of a European call at zero interest rate is the expected value of its payoff:

$$C(K, T) = \int_0^\infty (x - K)^+ \, \phi_T(x)\,dx$$

where $(x - K)^+ = \max(x - K, 0)$ and $\phi_T(x)$ is the *risk-neutral density*: the probability density function of the stock price S_T at time T, under the risk-neutral measure. Intuitively, $\phi_T(x)\,dx$ is the probability (adjusted for risk) that the stock ends up between x and $x + dx$ at time T.

To extract ϕ_T from the call prices, differentiate $C(K, T)$ twice with respect to K. The first derivative gives:

$$\frac{\partial C}{\partial K} = -\int_K^\infty \phi_T(x)\,dx$$

which is the (negative) probability that the stock ends above K. Differentiating again:

$$\phi_T(K) = \frac{\partial^2 C}{\partial K^2}$$

So the second derivative of the call price with respect to strike directly gives the risk-neutral density. Both derivatives are positive by no-arbitrage: call prices must be convex in K and increasing in T, otherwise one could construct a riskless profit.

Step 2: The Fokker-Planck (forward Kolmogorov) equation

Any diffusion process $dx = a(x,t)\,dt + b(x,t)\,dW$ evolves its probability density $f(x,t)$ according to the **Fokker-Planck equation**:

$$\frac{\partial f}{\partial t} = \frac{1}{2}\frac{\partial^2}{\partial x^2}\big(b^2 f\big) - \frac{\partial}{\partial x}(af)$$

This equation describes how the density spreads out over time as the process evolves. Under risk-neutrality (zero drift, $a = 0$), and using $f = \partial^2 C/\partial x^2$ from Step 1, this simplifies to:

$$\frac{\partial}{\partial t}\frac{\partial^2 C}{\partial x^2} = \frac{1}{2}\frac{\partial^2}{\partial x^2}\left(b^2 \frac{\partial^2 C}{\partial x^2}\right)$$

Step 3: Solving for the diffusion coefficient b

Integrating both sides twice in x (and using the boundary condition that $\partial C/\partial t \to 0$ as $x \to \infty$, since a call with an infinitely high strike is worthless), the integration constants vanish and we obtain:

$$\frac{1}{2}\,b^2\,\frac{\partial^2 C}{\partial x^2} = \frac{\partial C}{\partial t}$$

Since $\partial^2 C/\partial x^2 = \phi_T > 0$, we can solve for b^2:

$$b(x,t)^2 = \frac{\partial C/\partial t}{\frac{1}{2}\,\partial^2 C/\partial x^2}$$

Recalling that x denotes the strike K and t the maturity T, and writing $b = \sigma_{\text{loc}} \cdot S$ (since σ_{loc} is the *proportional* volatility of the stock), we arrive at the **Dupire equation**:

2.2.4 Solving for Local Volatility

From the derivation above, the local volatility $\sigma_{\text{loc}}(K,T)$ that makes the model consistent with all observed European call prices $C(K,T)$ is:

$$\sigma_{\text{loc}}(K,T) = \sqrt{\frac{\dfrac{\partial C}{\partial T} + rK\,\dfrac{\partial C}{\partial K}}{\dfrac{1}{2}\,K^2\,\dfrac{\partial^2 C}{\partial K^2}}}$$

Each term has a clear interpretation:

- $\dfrac{\partial C}{\partial T}$: how fast the call price increases as the maturity grows. Longer-dated options are worth more because the stock has more time to move; this captures the *term structure* of volatility.
- $rK\,\dfrac{\partial C}{\partial K}$: a correction for the risk-free rate (zero when $r = 0$). It accounts for the cost of carrying the position to expiration.
- $\dfrac{\partial^2 C}{\partial K^2}$: the convexity of call prices with respect to strike. This equals the risk-neutral density $\phi_T(K)$: a high value means many paths end near strike K, contributing a lot of probability mass there.
- The ratio numerator/denominator measures how much volatility is "needed" near the point (K,T) in order for the model to match the observed market price.

This is the key practical result: given a grid of market call prices, compute the partial derivatives numerically and plug into the formula to recover $\sigma_{\text{loc}}(K,T)$ at each point. The local volatility surface then fully specifies the stock process, which can be used to price and hedge any exotic or path-dependent option consistently with the market.

2.2.5 Why Dupire Equation is Ill-Posed

According to Hadamard, a PDE or inverse problem is *well-posed* if:

1. A solution exists.
2. The solution is unique.
3. The solution depends *continuously* on the data: small changes in the input produce only small changes in the output.

The equation for local volatility does not satisfy the third criterion [8]. Market options pricing data is not continuous and is extremely noisy. Taking the second derivative of this data (as required in the local volatility formula) becomes wild, so the Dupire/local volatility problem is an ill-posed PDE.

2.2.6 What does Ill-Posed mean for us?

Ill-posed problems are challenging to solve numerically because small changes in the input data can lead to large changes in the solution. This makes it difficult to find a stable and accurate solution.

Chapter 3

Neural Network Methods

3.1 Neural Network Basics

Neural networks are parameterized functions that approximate input–output mappings using layers of nonlinear transformations and are trained from data.

3.1.1 Architecture

A neural network is a layered computational graph that maps an input vector to an output vector through a sequence of parameterized transformations. Every network has three kinds of layers: an **input layer**, one or more **hidden layers**, and an **output layer**.

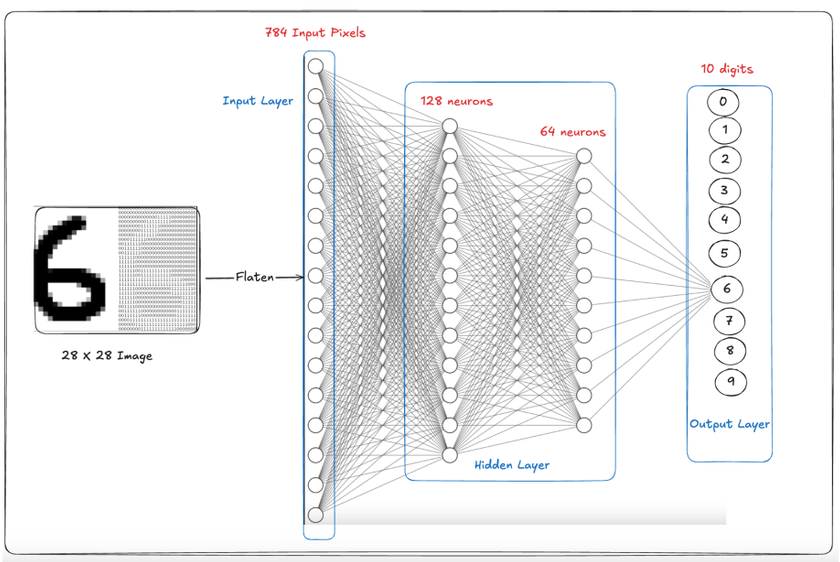

Figure 3.1: Architecture for handwritten digit recognition

Input layer. The input layer receives the raw data and passes it into the network. Each *node* (also called a *neuron*) in the input layer holds one feature of the input. Say we want to do digit detection from an image. Let's say our images are 28×28 (standard for handwritten digits), our input would be the image itself. We would *flatten* the image (turn it into a column vector), where each value in this vector corresponds to the grey-scale value of a *pixel* on the image. This is seen in figure 3.2 in the input layer. Similarly, if we are trying to price an option, the input might be (S, K, T, r, σ), five nodes, one per variable. The input layer performs no computation; it simply holds the values.

Hidden layers. The hidden layers are where the actual computation happens. To understand what each hidden layer does, let us trace the computation for a *single neuron*. Suppose the previous layer has n neurons with output values $x_1, x_2, \ldots, x_n$ (for input layer this is just the values of our inputs), which we think of as a column vector:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

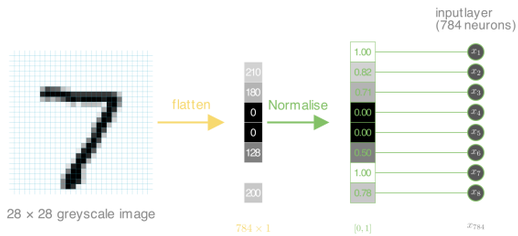

Figure 3.2: Input Layer

Step 1: Weights. Each neuron in the current layer has its own *weight vector*. For neuron i, the weight vector is

$$\mathbf{w}_i = \begin{pmatrix} w_{i1} \\ w_{i2} \\ \vdots \\ w_{in} \end{pmatrix},$$

where w_{ij} is the weight neuron i places on neuron j from the previous layer. Larger $|w_{ij}|$ means neuron j's output has more influence on neuron i.

Step 2: Dot product. The neuron computes a weighted sum of all inputs by taking the dot product of its weight vector with the input vector:

$$\mathbf{w}_i \cdot \mathbf{x} = \begin{pmatrix} w_{i1} \\ w_{i2} \\ \vdots \\ w_{in} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = w_{i1}x_1 + w_{i2}x_2 + \cdots + w_{in}x_n.$$

This is a single number summarising how strongly the previous layer's activity aligns with this neuron's weights.

Step 3: Bias. We may not want the neuron to activate unless the weighted sum clears some threshold. To capture this, each neuron has a single learnable *bias* term b_i, which shifts the weighted sum:

$$\mathbf{w}_i \cdot \mathbf{x} + b_i = w_{i1}x_1 + w_{i2}x_2 + \cdots + w_{in}x_n + b_i.$$

A large negative b_i makes the neuron hard to activate; a large positive b_i makes it fire even for weak inputs.

Step 4: Activation function. Without any further transformation, every neuron's output would be a linear function of the inputs, and stacking linear layers would only ever produce a linear map no matter how deep the network. To introduce nonlinearity, the pre-activation value is passed through an *activation function* σ. The choice of σ is arbitrary (discussed below); what matters is that it is nonlinear. The final output of neuron i is:

$$h_i = \sigma(w_{i1}x_1 + w_{i2}x_2 + \cdots + w_{in}x_n + b_i) = \sigma\left(\begin{pmatrix} w_{i1} \\ w_{i2} \\ \vdots \\ w_{in} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b_i\right).$$

Every neuron in the hidden layer performs exactly this computation with its own weight vector $\mathbf{w}_i$ and bias b_i. The outputs $h_1, h_2, \ldots$ of all neurons in the layer are then collected into a vector and passed to the next layer.

Generalization: the whole layer at once. Suppose the layer has m neurons. Each neuron i has its own weight vector $\mathbf{w}_i^\top$ (a row), so stacking all m rows gives a weight matrix W:

$$W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

The m dot products are all computed simultaneously by the matrix–vector product $W\mathbf{x}$:

$$W\mathbf{x} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} w_{11}x_1 + \cdots + w_{1n}x_n \\ w_{21}x_1 + \cdots + w_{2n}x_n \\ \vdots \\ w_{m1}x_1 + \cdots + w_{mn}x_n \end{pmatrix}.$$

Adding the bias vector and applying σ element-wise gives the full layer output in one expression:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b}) = \sigma\left(\begin{pmatrix} w_{11}x_1 + \cdots + w_{1n}x_n + b_1 \\ w_{21}x_1 + \cdots + w_{2n}x_n + b_2 \\ \vdots \\ w_{m1}x_1 + \cdots + w_{mn}x_n + b_m \end{pmatrix}\right) = \begin{pmatrix} \sigma(w_{11}x_1 + \cdots + w_{1n}x_n + b_1) \\ \sigma(w_{21}x_1 + \cdots + w_{2n}x_n + b_2) \\ \vdots \\ \sigma(w_{m1}x_1 + \cdots + w_{mn}x_n + b_m) \end{pmatrix}.$$

Each row of W is one neuron's weight vector; each entry of $\mathbf{h}$ is that neuron's output. This is simply the single-neuron computation repeated m times, packed into a matrix multiply.

Activation functions. Without a nonlinear activation function, stacking multiple linear layers would still produce a linear map, no matter how deep the network. The activation function introduces nonlinearity, allowing the network to approximate complex functions. Common choices are:

- **ReLU** (Rectified Linear Unit): $\phi(z) = \max(0, z)$. Zero for negative inputs, linear for positive. Fast to compute and widely used. Can suffer from "dying ReLU" (neurons that always output zero).
- **Sigmoid**: $\phi(z) = 1/(1 + e^{-z})$. Squashes output to $(0, 1)$. Historically popular but can cause vanishing gradients in deep networks.
- **Tanh**: $\phi(z) = \tanh(z)$. Squashes output to $(-1, 1)$. Centred at zero, often preferred over sigmoid.
- **Softplus**: $\phi(z) = \ln(1 + e^z)$. A smooth approximation to ReLU. Useful when differentiability of the activation is required (e.g. in PINNs).

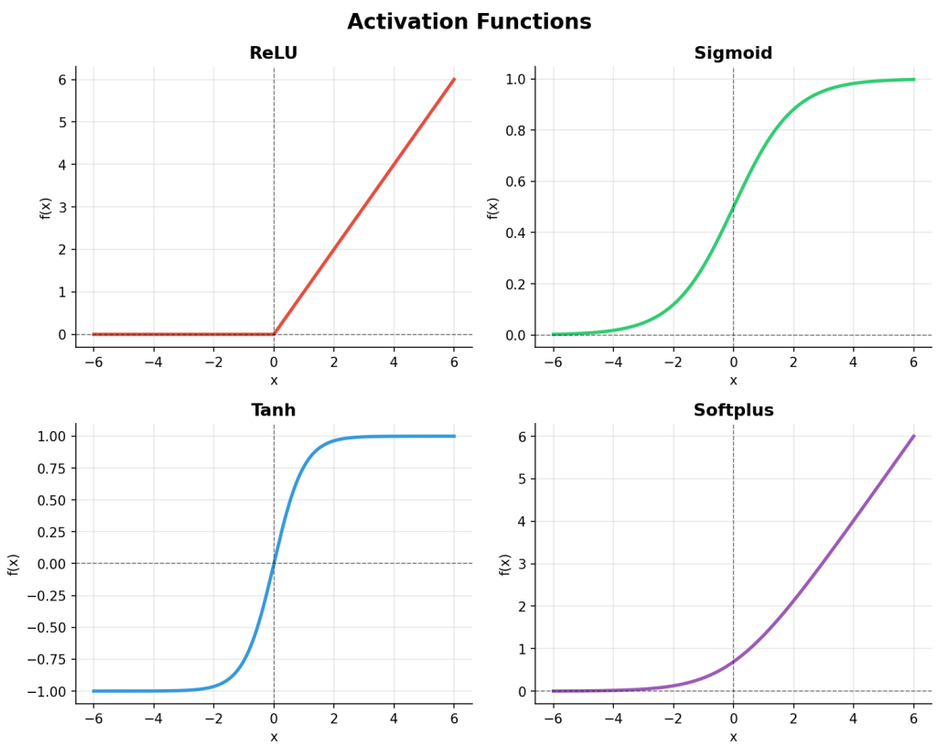

Figure 3.3: Common activation functions used in neural networks.

Output layer. The output layer produces the final prediction. Its width equals the number of outputs required. For option pricing, this might be a single node (the option price V). For local volatility recovery, it might output σ_{loc} at a given (K, T). The output layer often has no activation function (or a linear one), so the network can predict any real number rather than being squashed into a fixed range.

Depth and width. The **depth** of a network is the number of layers (input + hidden + output). The **width** of a layer is the number of neurons in it. A *deep* network has many layers; a *wide* network has many neurons per layer. Depth allows the network to compose simple features into complex ones (e.g. edges → shapes → objects in image recognition). Width allows each layer to

represent more patterns simultaneously. The total number of **parameters** (weights and biases) in a fully connected network with layers of sizes $n_0, n_1, \ldots, n_L$ is:

$$\text{parameters} = \sum_{\ell=1}^{L} \big(n_{\ell-1} \cdot n_\ell + n_\ell\big) = \sum_{\ell=1}^{L} n_\ell(n_{\ell-1}+1).$$

For example, a network with input size 5, two hidden layers of width 64, and output size 1 has $5 \cdot 64 + 64 + 64 \cdot 64 + 64 + 64 \cdot 1 + 1 = 4{,}545$ parameters.

3.1.2 Forward Pass

The *forward pass* is the process of pushing an input through every layer of the network in sequence to produce a prediction. The architecture section showed what a single layer computes; here we chain all L layers together into one unified notation and see what comes out the other end.

Full network notation

Label the layers $\ell = 0, 1, \ldots, L$, where $\ell = 0$ is the input, $\ell = 1, \ldots, L-1$ are hidden layers, and $\ell = L$ is the output. Each layer ℓ has a weight matrix $W^{(\ell)}$ and bias vector $\mathbf{b}^{(\ell)}$. Define the activations recursively:

$$\mathbf{h}^{(0)} = \mathbf{x}, \qquad \mathbf{h}^{(\ell)} = \sigma\big(W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\big) \quad \text{for } \ell = 1, \ldots, L-1.$$

The pre-activation (the value before σ is applied) at layer ℓ is written $\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$, so $\mathbf{h}^{(\ell)} = \sigma(\mathbf{z}^{(\ell)})$. This separation is useful for backpropagation later.

Output layer and softmax

The output layer $\ell = L$ is treated differently depending on the task. For **regression** (predicting a continuous number, such as an option price), the output layer applies no activation:

$$\hat{y} = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)} \in \mathbb{R}.$$

For **classification** (predicting one of C discrete classes, such as identifying which digit is in an image), the output layer produces a vector of C raw scores called *logits*, $\mathbf{z}^{(L)} \in \mathbb{R}^C$, and then the **softmax** function converts them into probabilities:

$$\hat{p}_c = \text{softmax}(\mathbf{z}^{(L)})_c = \frac{e^{z_c^{(L)}}}{\displaystyle\sum_{j=1}^{C} e^{z_j^{(L)}}}, \qquad c = 1, \ldots, C.$$

Intuition for softmax. The raw logits $z_1, \ldots, z_C$ can be any real numbers and do not sum to one. Softmax exponentiates each logit and then normalises by the total, producing a valid probability distribution ($\hat{p}_c > 0$, $\sum_c \hat{p}_c = 1$). The exponentiation has two important effects: it is always positive (so no negative probabilities), and it *amplifies differences*, a logit that is only slightly larger than the others ends up with a noticeably larger probability. In the limit, if one logit dominates, softmax approaches a one-hot vector (all probability on that class). This makes softmax a smooth, differentiable approximation to the operation "pick the most likely class."

Loss function

The network has parameters $\theta = \{W^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$. Training means choosing θ so that the network's predictions are close to the true targets. The **loss function** $\mathcal{L}(\theta)$ measures this error on a training set of N examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$.

For **regression**, the standard choice is *mean squared error*, and we will use this as well:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\big(\hat{y}^{(i)} - y^{(i)}\big)^2.$$

While there are other loss functions, for the purposes of this paper, mean squared error will work well. The training problem is simple then: find θ^* such that $\mathcal{L}(\theta)$ is minimized. Gradient descent and backpropagation (the next two sections) are the tools used to find θ^*.

Numerical example

Consider a tiny network with:

- Input: $\mathbf{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ (2 features)

- One hidden layer of width 2, with ReLU activation
- Output layer of width 2, with softmax (classifying into 2 classes)

and parameters:

$$W^{(1)} = \begin{pmatrix} 1 & -1 \\ 0 & 2 \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \qquad W^{(2)} = \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix}, \quad \mathbf{b}^{(2)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Step 1: Hidden pre-activation.

$$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)} = \begin{pmatrix} 1 & -1 \\ 0 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Step 2: Hidden activation (ReLU).

$$\mathbf{h}^{(1)} = \mathrm{ReLU}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Both values are positive so ReLU leaves them unchanged.

Step 3: Output pre-activation (logits).

$$\mathbf{z}^{(2)} = W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)} = \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}.$$

Step 4: Softmax.

$$e^{z_1} = e^2 \approx 7.389, \quad e^{z_2} = e^{-1} \approx 0.368, \quad \text{sum} \approx 7.757.$$

$$\hat{\mathbf{p}} = \begin{pmatrix} 7.389/7.757 \\ 0.368/7.757 \end{pmatrix} \approx \begin{pmatrix} 0.953 \\ 0.047 \end{pmatrix}.$$

The network assigns 95.3% probability to class 1 and 4.7% to class 2.

Step 5: Loss. If the true label is class 1, the MSE loss is:

$$\mathcal{L} = \frac{(1 - 0.954)^2 + (0 - 0.047)^2}{2} \approx 0.0022$$

This is a small loss because the network is already very confident in the correct class. If the true label were class 2 instead, the loss would be ≈ 0.9082, a large penalty for being confidently wrong (considering the max error in this network is 1).

3.1.3 Gradient Descent

Once we have a loss $\mathcal{L}(\theta)$, training is an optimisation problem: find the parameters θ that make $\mathcal{L}$ as small as possible. Gradient descent does this by repeatedly taking small steps in the direction that decreases the loss the fastest.

The gradient $\nabla_\theta \mathcal{L}$ is a vector that points in the direction of *steepest increase* of the loss. So to decrease the loss, we move in the *opposite* direction. The update rule is:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta \mathcal{L}(\theta),$$

where $\eta > 0$ is the **learning rate**: a small scalar controlling how large each step is. If η is too large, the steps overshoot and the loss may diverge; if too small, training is slow. This is an important *hyperparameter* that gets changed during machine learning R&D. Hyperparameters are any part of the configuration that is set *before* the training process begins (model depth, model width, batch size, etc), compared to model parameters that are *learned*.

This is repeated for many iterations. At each step the parameters move a little closer to a (local) minimum of $\mathcal{L}$, and the network's predictions gradually improve.

3.1.4 Backpropagation

Gradient descent requires $\nabla_\theta \mathcal{L}$, the derivative of the loss with respect to every weight and bias in the network. Backpropagation is the algorithm that computes all of these derivatives efficiently by working backwards through the network using the chain rule.

The network

To keep the algebra transparent, consider the smallest non-trivial network: one input x, two hidden layers each containing a single neuron, and one output $\hat{y}$ (regression, no final activation).

The forward pass computes:

$$\begin{aligned} z_1 &= w_1 x + b_1, & h_1 &= \sigma(z_1), \\ z_2 &= w_2 h_1 + b_2, & h_2 &= \sigma(z_2), \\ \hat{y} &= w_3 h_2 + b_3. \end{aligned}$$

The loss (MSE with one example) is $\mathcal{L} = (\hat{y} - y)^2$.

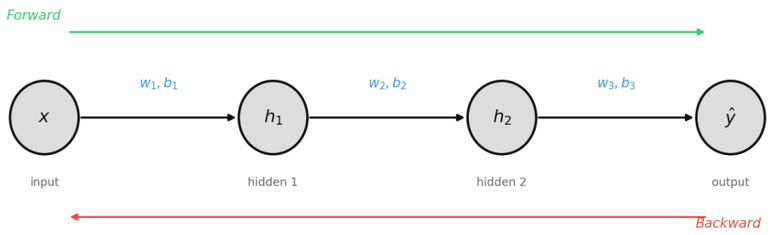

Figure 3.4: A minimal network with two hidden neurons. The green arrow shows the forward pass direction; the red arrow shows the backward (backpropagation) direction.

Backward pass: output layer

We want $\partial\mathcal{L}/\partial w_3$, $\partial\mathcal{L}/\partial b_3$, and so on, back to w_1. Start at the output and apply the chain rule one step at a time.

The loss depends on $\hat{y}$, and $\hat{y}$ depends on w_3:

$$\frac{\partial\mathcal{L}}{\partial w_3} = \frac{\partial\mathcal{L}}{\partial\hat{y}} \cdot \frac{\partial\hat{y}}{\partial w_3} = 2(\hat{y}-y)\cdot h_2, \qquad \frac{\partial\mathcal{L}}{\partial b_3} = 2(\hat{y}-y).$$

Define the shorthand $\delta_{\hat{y}} = \partial\mathcal{L}/\partial\hat{y} = 2(\hat{y}-y)$. This is the error signal at the output.

Backward pass: hidden layer 2

To reach w_2, the chain now passes through h_2 and then through the activation σ:

$$\frac{\partial\mathcal{L}}{\partial h_2} = \delta_{\hat{y}} \cdot \frac{\partial\hat{y}}{\partial h_2} = \delta_{\hat{y}} \cdot w_3.$$

$$\frac{\partial\mathcal{L}}{\partial z_2} = \frac{\partial\mathcal{L}}{\partial h_2} \cdot \sigma'(z_2) = \delta_{\hat{y}}\, w_3\, \sigma'(z_2).$$

Define $\delta_2 = \partial\mathcal{L}/\partial z_2 = \delta_{\hat{y}}\, w_3\, \sigma'(z_2)$. Then:

$$\frac{\partial\mathcal{L}}{\partial w_2} = \delta_2 \cdot h_1, \qquad \frac{\partial\mathcal{L}}{\partial b_2} = \delta_2.$$

Backward pass: hidden layer 1

Continuing one more step back:

$$\frac{\partial\mathcal{L}}{\partial h_1} = \delta_2 \cdot w_2, \qquad \delta_1 = \frac{\partial\mathcal{L}}{\partial z_1} = \delta_2\, w_2\, \sigma'(z_1).$$

$$\frac{\partial\mathcal{L}}{\partial w_1} = \delta_1 \cdot x, \qquad \frac{\partial\mathcal{L}}{\partial b_1} = \delta_1.$$

The pattern

Each layer's error signal δ_ℓ is formed by taking the next layer's error signal, multiplying by the connecting weight (passing the error backwards through the weight), and then multiplying by $\sigma'(z_\ell)$ (passing the error backwards through the activation). The gradient for each weight is then just δ_ℓ times the activation that fed into it. This recursive structure is why the algorithm is called *back*propagation: the error signal δ propagates from output to input, and each layer uses it to compute its own gradients. In short, it's a very long chainrule.

Once all gradients are in hand, gradient descent updates every parameter simultaneously:

$$w_i \leftarrow w_i - \eta\,\frac{\partial\mathcal{L}}{\partial w_i}, \qquad b_i \leftarrow b_i - \eta\,\frac{\partial\mathcal{L}}{\partial b_i}.$$

Note on computing derivatives. In practice, no one computes these gradients by hand. They are evaluated automatically using *automatic differentiation*: a foundational software that traces every elementary operation the network performs, then applies the chain rule back through them exactly.

Let's see the forest from the trees

That was a lot of notation, I find it can be hard to see the big picture with a bunch of new notation, and for neural networks, I find that the notation is rather trivial to understanding what's *really* happening.

1. **Inputs & Outputs** The first and most important step in the process: what are we finding (output), and through what information (input). For instance, I want to classify handwritten digit images. Input: image. Output: digit.
2. **Model Architecture** Choose starting hyperparameter for the model. This can be arbitrary to start out with, though it is beneficial to start with standard model architecture choices and tune them as the model is being tested and trained.
3. **Initialize model parameters** Not to be confused with hyperparameters, model paramaters are the *learned* paramaters θ. These are initialized completely randomly to start, but it is smart to keep them within a (-1,1) normally distributed range.
4. **Forward pass** Put an input through the network. That is: calcualted the dot products between all nodes plus the biases, apply activation function, continue till the output layer, and get the networks initial prediction. This will almost certaintly be extremely wrong.
5. **Play the blame game** Now we find *who* is causing the network to be inaccurate. Go through the giant chain rules to find gradiant of the loss function $\mathcal{L}_\theta$.
6. **Correct Errors** Move the paramaters towards the direction of the loss functions greatest decrease.
7. **Repeat**

Once this process is done, often we will tune hyperparameters, check code correctness, or potentially rethink our inputs / outputs for this problem.

3.1.5 Spectral Bias

Neural networks can in principle fit any function, including wildly oscillating ones. Yet in practice, when trained by gradient descent, they do not learn all frequencies at the same rate. Rahaman et al. [9] demonstrated through Fourier analysis of ReLU networks that *lower frequencies are learned first*. A network trained on a target function $\phi(z) = \sum_i A_i \sin(2\pi k_i z)$ fits the small-k (slowly varying) components early in training and only reaches the large-k (rapidly oscillating) components much later, regardless of the amplitudes A_i. This ordering is not a choice, it is a structural consequence of how gradient descent moves through parameter space. Because the Lipschitz constant of the network (which controls how fast the function can oscillate) grows only gradually during training, high-frequency structure simply cannot be expressed until the parameter norm is large enough.

A second consequence reported in [9] is *robustness*: after training, the low-frequency components of the learned function are far more stable under random perturbations of the weights than the high-frequency components. High-frequency modes require the parameters to be finely coordinated, so they occupy a small volume in parameter space and are easily disrupted.

Why this is useful for our problem. The volatility surface $\sigma_{\text{loc}}(K, T)$ we want to recover is a smooth, slowly varying function, it has no sharp spikes or rapid oscillations. Spectral bias means the network will naturally prioritise fitting that smooth global shape before attempting to fit any noise in the data. This acts as an *implicit regulariser*. For an ill-posed problem like local volatility calibration, where the data is noisy and second derivatives amplify that noise, having a method that structurally prefers smooth solutions is precisely what is needed. I'm gonna use some pictures to try to illustrate this process.

3.2 Neural Adjoint Method

The neural-adjoint method [5] trains a network to solve inverse PDE problems by combining a neural network parameterization with the adjoint method to compute gradients efficiently.

3.2.1 Adjoint Method

When we train a neural network, gradient descent needs $\frac{\partial \mathcal{L}}{\partial \theta}$: how does the loss change with respect to each parameter? For standard networks, backpropagation handles this efficiently. But when the network is coupled to a differential equation, the dynamics of that equation sit between the parameters and the loss, and backpropagating through every step of a numerical ODE solver becomes extremely expensive in memory.

The adjoint method is a classical technique that computes the same gradients without storing any intermediate solver states. The key insight is that instead of differentiating through the forward

solve, you run a separate *backward* ODE, called the adjoint equation, whose solution tells you exactly how sensitive the loss is to the initial conditions and parameters.

A simple example. Suppose we have the scalar ODE

$$\frac{dy}{dt} = f(y, t; \theta), \quad y(t_0) = y_0,$$

and after solving forward to time T we compute a scalar loss $\mathcal{L} = L(y(T))$.

Step 1: Solve forward. Integrate the ODE from t_0 to T to get $y(T)$ using any numerical solver. This is the normal forward pass.

Step 2: Define the adjoint. The adjoint variable is $\lambda(t) = \frac{\partial \mathcal{L}}{\partial y(t)}$, which tracks how the loss depends on the state at each time. It satisfies the adjoint ODE, run *backwards* in time:

$$\frac{d\lambda}{dt} = -\lambda(t)\frac{\partial f}{\partial y}, \quad \lambda(T) = \frac{\partial L}{\partial y(T)}.$$

The terminal condition at $t = T$ is just the derivative of the loss with respect to the final state, which is easy to compute.

Step 3: Solve backward. Integrate the adjoint equation from T back to t_0. This is another ODE solve, but crucially it requires no memory of the forward trajectory.

Step 4: Recover the gradient. The gradient of the loss with respect to the parameters is then

$$\frac{\partial \mathcal{L}}{\partial \theta} = -\int_{t_0}^{T} \lambda(t)\frac{\partial f}{\partial \theta}\, dt,$$

which is computed in the same backward pass. This gives the exact gradient that gradient descent needs, at the cost of one additional ODE solve rather than storing the full computational graph of the forward solver [19].

Intuitively. While I don't entirely understand the math behind the ODE, for the sake of neural networks this is how I like to think of this: you roll a ball down a mountain, and at the bottom you see that the ball didn't arrive where you wanted it to. Instead of tracing the ball's path down the mountain to see exactly how each part of the mountain influenced its path, you somehow have a way to always perfectly throw the ball from your desired destination *back* to the top. This way, you can *follow* the ball's exact path, and know how you should toss it from the top.

3.2.2 Using Adjoint in Neural Net

A Neural ODE is a model where the hidden state of a network evolves continuously according to a differential equation, rather than through a discrete sequence of layers. Instead of stacking transformations $h_{t+1} = h_t + f(h_t, \theta)$ as in a residual network, we parameterize the dynamics directly:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta),$$

where f is a neural network with parameters θ. The output is simply $h(T)$, the solution to this ODE at some terminal time T, computed by any numerical solver [20].

The challenge is training. Backpropagating through every internal step of the ODE solver would require storing the entire forward trajectory, which is expensive in memory and introduces extra numerical error. The adjoint sensitivity method, described in the previous section, solves this exactly. The ODE solver is treated as a black box, and gradients are recovered by solving a second augmented ODE backwards in time, with constant memory cost regardless of how many steps the forward solver took [18].

Concretely, the backward pass computes three quantities simultaneously in a single reverse ODE solve: the state $h(t)$ recomputed backwards, the adjoint $a(t) = \partial \mathcal{L}/\partial h(t)$, and the parameter gradient

$$\frac{d\mathcal{L}}{d\theta} = -\int_{T}^{t_0} a(t)^\top \frac{\partial f}{\partial \theta}\, dt.$$

All three are concatenated into one augmented state vector and integrated together, so the total cost is two ODE solves: one forward, one backward [18].

For our purposes, the key takeaway is that a Neural ODE can represent a continuously evolving process governed by learnable dynamics, and it can be trained end-to-end with exactly the same gradient descent machinery as any other neural network, thanks to the adjoint method handling the gradient computation efficiently. This turns memory needed in backpropagation from $\mathcal{O}(N) \to \mathcal{O}(1)$.

3.2.3 Strict Enforcement

The Neural Adjoint Method enforces boundary conditions strictly by construction: the network output is analytically modified so that it satisfies the constraints for any choice of parameters. For example, if the boundary condition requires $u = 0$ at $t = 0$, the network output can be written as $\hat{u}(x, t) = t \cdot \mathrm{NN}(x, t)$, which is identically zero at $t = 0$ regardless of what the network outputs. This means the boundary conditions are never part of the loss, they are simply never violated.

3.2.4 Benefits

The Neural Adjoint method is more complicated to implement and understand, but it has major benefits:

1. **Constant memory**: Normally, backprop through an ODE solver would force you to save every step of the forward solve so you can trace gradients backwards. The more steps, the more memory. The adjoint method sidesteps this: the backward pass is just another ODE solve, so you never store the forward trajectory. Memory stays constant no matter how many steps the solver takes [18].

2. **Finds solutions precisely**: Instead of sampling random guesses, the NA walks directly toward the best input using gradients from a trained forward model. Ren, Padilla, and Malof [17] show that this makes it beat other inverse methods (invertible nets, VAEs) on benchmark tasks, especially when you allow more candidate solutions. Other methods either miss good regions or land somewhere close but not quite right; the NA descends straight to the optimum.

3. **No inverse training data needed**: You only train a forward model (input $\rightarrow$ output), which is easy to get from a simulator. You never need examples of inverse solutions, which are often costly or impossible to collect. To solve an inverse problem, you just run gradient descent on the inputs until the forward model gives you the output you want [17].

3.2.5 Drawbacks

1. **It takes forever to train**: When I was training this type of network, it took over 10$\times$ the amount of time. This will be shown in the final section.

2. **Only solves inverse problems**: You could call this a specialization, but for me it is a drawback. The method is built to find inputs that produce a desired output; it cannot naturally be turned around for forward problems, where you have inputs and want the solution. PINNs can do both in one framework; the NA is built only for inverse.

3.3 Physics Informed Neural Networks (PINN)

3.3.1 Basics and Use Cases

Physics-informed neural networks [10] incorporate the PDE (and optionally boundary/initial conditions) into the training loss so that the network learns to satisfy the governing equations.

3.3.2 How to make sure Neural Networks follow the laws of Physics

A standard neural network learns by minimising the gap between its outputs and known data. But a PDE solution must also satisfy the governing equation everywhere in the domain, something data-fitting alone does not enforce. The fix is simple in principle: penalise the network not just for wrong outputs, but also for violating the PDE. The network is then trained to satisfy both at once.

3.3.3 Cost Function and PDE Residual

Take the 1D heat equation as a concrete example:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}.$$

The network takes a point (x, t) as input and outputs a single number $\hat{u}(x, t)$, its best guess for the solution at that location and time. This is analogous to a regression network predicting a function value. Thus, we consider the distance from the governing equation as well.

Standard data loss. If we have some observed or labelled values of u (say, from measurements or boundary/initial conditions), the network is penalised for deviating from them in the usual way using mean squared error. This alone is not enough it only constrains the network at the points where data is available, and says nothing about whether the heat equation holds anywhere else.

PDE residual loss. If $\hat{u}$ truly solves the heat equation, then plugging it in gives zero everywhere:

$$r(x,t) = \frac{\partial \hat{u}}{\partial t} - \frac{\partial^2 \hat{u}}{\partial x^2} = 0.$$

This quantity r is the **PDE residual**: it measures how badly the network's output violates the governing equation at a given point. We scatter a large set of collocation points $\{(x_j, t_j)\}$ across the domain, evaluate r at each one, and add the mean squared residual to the loss:

$$\mathcal{L}_{\text{PDE}} = \frac{1}{M} \sum_{j=1}^{M} r(x_j, t_j)^2.$$

Total loss. The network minimises the sum of both terms:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{PDE}}.$$

This forces the network to match known values *and* satisfy the differential equation between them. This is exactly what we do when we solve a PDE by hand.

3.3.4 Other internal changes

This seems very simple, so there must be other differences right? Actually, not really. For standard PINN's the only other notable difference is the use of autodifferentiaion during the forward pass, rather than just during backpropogation. This is because, again, derivatives don't need to be computed by hand. So rather than computing $\frac{\partial \hat{u}}{\partial t}$ by hand, we can use autodifferentiaion to compute it. This would essentially trace the output of the network back to it's t input for us. PINNs are quite a simple and intuitive extension of neural networks.

3.3.5 Soft Enforcement

PINNs are really simple, so what's the catch? Soft enforcement adds the PDE residual and any boundary or initial conditions as *penalty* terms in the loss; the network is trained to minimise these terms but does not satisfy them exactly.

This distinction matters. Because the PDE is only a penalty, the network can "cheat". It may find a set of parameters that keeps the loss small without the physics being satisfied to any meaningful precision. For example, imagine solving the heat equation with a fixed boundary temperature of $u = 0$ at $x = 0$. A standard PINN is penalised when $\hat{u}(0,t) \neq 0$, but if violating that condition slightly allows the network to reduce the residual loss more elsewhere, it will. The result is a solution that looks reasonable globally but drifts away from the true boundary value, with no guarantee of how large the error is.

3.3.6 Benefits

PINNs have numerous benefits; here are the most important:

1. **Need less data**: A typical neural network needs thousands of labelled examples. A PINN can get away with far fewer. The PDE itself acts as extra training signal: once you have some known values, you can check whether the physics holds at as many points as you like across the domain, with no labels. The PDE teaches the network the structure of the solution, so it learns reliably from fewer observations.

2. **No grid required**: Classical solvers (finite element, finite difference) need a mesh over the domain. Building that mesh is nontrivial for complex shapes, and the solution quality depends on how fine it is. PINNs skip this entirely. The solution is a continuous function, so you can plug in any point wherever you want. As Roggeveen and Brenner [14] note, this makes PINNs naturally grid-free, which helps for problems with moving boundaries or irregular geometry.

3. **Scales to high dimensions**: Classical methods hit the curse of dimensionality: cost grows as N^d for a grid in d dimensions, which quickly becomes unmanageable. Neural nets do not discretise the domain; the solution is one function of all inputs, and the number of parameters grows far more slowly. Han, Jentzen, and E [15] solved nonlinear PDEs in up to 100 dimensions, including Black–Scholes for baskets of assets, with accuracy and speed that classical methods cannot match. For finance with many underlyings or risk factors, this is a decisive advantage.

3.3.7 Drawbacks

There are two very important drawbacks:

1. **Convergence is a nightmare**: The loss has several competing terms (PDE residual, boundary conditions, data). The optimizer can favor one and ignore another. For example, while training a PINN for the Dupire equation, it would often fit the boundary values well but let the PDE residual drift, producing a solution that looks reasonable but violates the physics. Wang, Teng, and Perdikaris [13] identify the cause: gradients from different loss terms can differ by orders of magnitude, so the optimizer effectively ignores the smaller ones. Fixing this requires careful hyperparameter tuning.

2. **No built-in uncertainty**: A PINN gives you a single number for each input, with no indication of how confident it is or how large the error might be. For inverse problems like local volatility calibration, where the data is noisy and the problem is ill-posed, knowing the reliability of the solution matters as much as the solution itself. Flores et al. [16] note that PINNs are not naturally equipped for this; adding uncertainty quantification requires extra machinery (e.g. Bayesian neural networks).

Chapter 4

Comparing Methods

4.1 Comparison Framework

Both methods were evaluated on the same synthetic local volatility calibration problem: recover $\sigma(K, T)$ from a grid of call prices generated by a known ground truth surface (Figure 4.1). Three metrics were used:

- **Accuracy (MSE)**: Mean squared error between the predicted local volatility surface and the ground truth. Lower is better.
- **Smoothness (Grad^2)**: Average squared gradient magnitude of the predicted surface. The true local volatility surface is smooth, so a well-calibrated method should recover a smooth surface. Lower is better.
- **Time (s)**: Total wall-clock training time. Practically relevant for any real-time calibration use case.

Each method was tested at three noise levels (0%, 1%, 5%) applied to the call price inputs, to assess robustness to market microstructure noise.

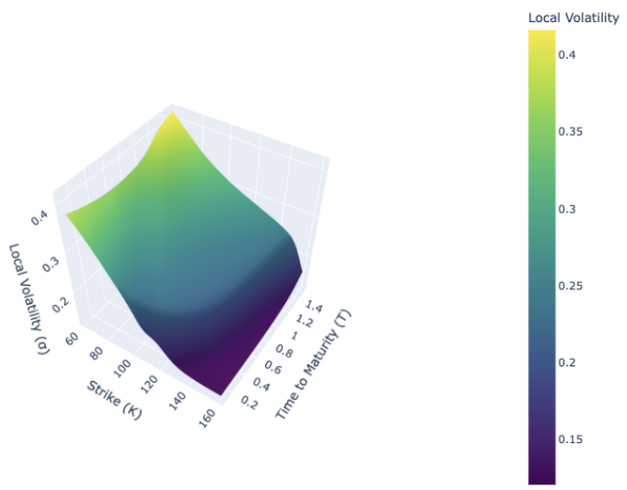

Figure 4.1: Ground truth local volatility surface $\sigma(K, T)$ used as the calibration target across all experiments.

4.2 Results

The results in Figure 4.2 tell a clear story. The Neural Adjoint dominates on accuracy and smoothness across every noise level, achieving MSEs two to three orders of magnitude lower than the Inverse PINN. It also produces substantially smoother surfaces, suggesting it respects the underlying structure of the local volatility function rather than just fitting to the observed call prices. The tradeoff is severe: the Neural Adjoint takes roughly 950 seconds to train versus around 75 seconds for the Inverse PINN, making it about 13 times slower.

Notably, the Inverse PINN's accuracy does improve meaningfully at higher noise (5%), closing some of the gap. This may reflect the fact that the PDE residual loss acts as a regulariser, preventing the network from overfitting to noisy observations. Still, the Neural Adjoint maintains a significant advantage at every level.

	Method	Noise Level	Time (s)	Accuracy (MSE)	Smoothness (Grad^2)
0	Neural Adjoint	0%	950.4	0.000002	0.000861
1	Inverse PINN	0%	75.2	0.000562	0.010250
2	Neural Adjoint	1%	965.5	0.000001	0.000768
3	Inverse PINN	1%	75.7	0.000533	0.010794
4	Neural Adjoint	5%	950.3	0.000101	0.002582
5	Inverse PINN	5%	72.2	0.000997	0.017570

Figure 4.2: Comparison of Neural Adjoint and Inverse PINN across noise levels. Metrics are accuracy (MSE), smoothness (squared gradient magnitude), and training time.

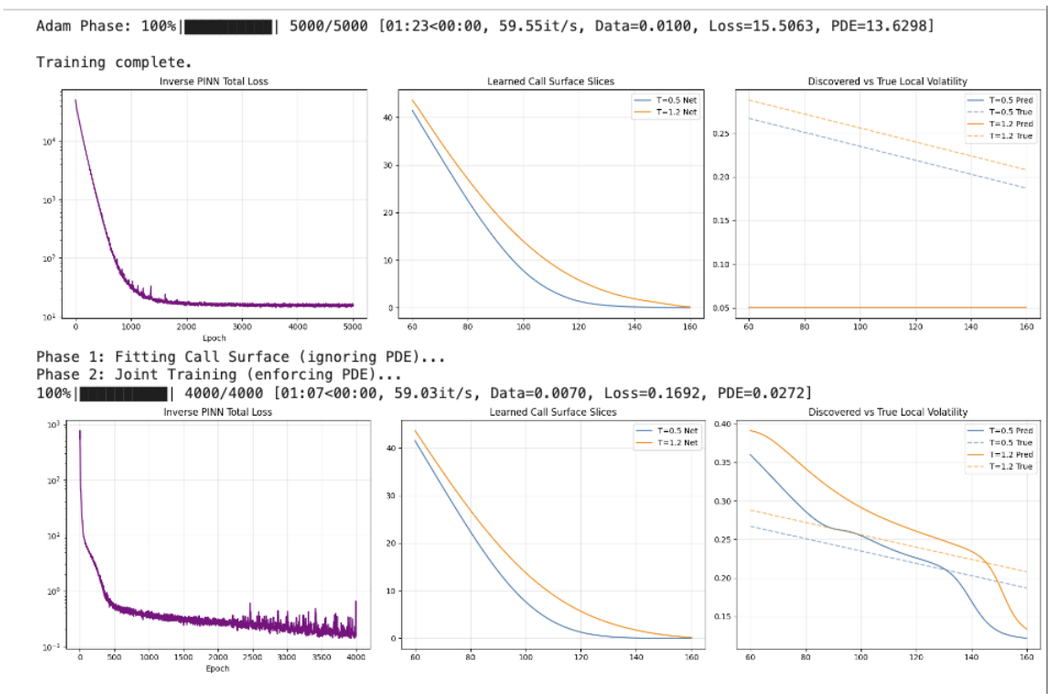

Figure 4.3: Training dynamics for the Neural Adjoint (top row) and Inverse PINN (bottom row). Left: total loss over epochs. Centre: learned call price surface slices at $T = 0.5$ and $T = 1.2$. Right: predicted vs. true local volatility at the same maturities.

Figure 4.3 shows the training dynamics for each method. The Neural Adjoint (top row) converges to a loss that is orders of magnitude lower and recovers the local volatility surface with nearly perfect accuracy. The call surface slices align closely with the truth, and the right panel shows the predicted and true local volatility curves overlapping almost exactly.

The Inverse PINN (bottom row) converges faster and more stably, but the recovered local volatility is noticeably noisier, with the predicted curves deviating from the truth especially at the edges of the strike domain. This is consistent with its weaker smoothness score in the table: the PDE constraint helps, but it is not sufficient to fully regularize the solution in this problem.

4.3 Conclusions

Overall, I think that the Neural Adjoint method out preforms the PINN for inverse problems substantially. This makes sense considering the Neural Adjoint Method is a specialization to solve inverse problems, whereas to the PINN is solving to lower a loss function related to the value we wish to find. However, the PINN was *substantially* easier and quicker to develop. Not only was training time orders of magnitude faster, but development time was too. The Neural Adjoint method required using forward and backward equations. Whereas when using the PINN, it was just implimenting the PDE residuals, which are given by the governing functions.

This project took significantly longer than I anticipated, but I am really happy with how everything turned out. I learned a lot about numerical methods, neural networks, stochastic PDEs, physics informed machine learning, and optimization.

Bibliography

[1] Black, F., & Scholes, M. (1973). The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3), 637–654.

[2] Crépey, S. (2003). Calibration of the local volatility in a generalized Black-Scholes model using Tikhonov regularization. *SIAM Journal on Mathematical Analysis*, 34(5), 1183–1206.

[3] Dupire, B. (1994). Pricing with a smile. *Risk*, 7(1), 18–20.

[4] Gatheral, J. (2006). *The volatility surface: A practitioner's guide*. Wiley Finance, Hoboken, NJ.

[5] Giles, M. B. (2006). Smoking adjoints: Fast Monte Carlo Greeks. *Risk*, 19(1), 93–97.

[6] Golub, G. H., Hansen, P. C., & O'Leary, D. P. (1999). Tikhonov regularization and total least squares. *SIAM Journal on Matrix Analysis and Applications*, 21(1), 185–194.

[7] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359–366.

[8] Lagnado, R., & Osher, S. (1997). A technique for calibrating derivative security pricing models: Numerical solution of an inverse problem. *Journal of Computational Finance*, 1(1), 13–25.

[9] Rahaman, N., Baratin, A., Arpit, D., Draxler, F., Lin, M., Hamprecht, F. A., Bengio, Y., & Courville, A. (2019). On the spectral bias of neural networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 97, 5301–5310. PMLR.

[10] Raissi, M., Perdikaris, P., & Karniadakis, G. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear PDEs. *Journal of Computational Physics*, 378, 686–707.

[11] Shreve, S. E. (2004). *Stochastic calculus for finance II: Continuous-time models*. Springer, New York.

[12] Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2018). Deep image prior. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 9446–9454.

[13] Wang, S., Teng, Y., & Perdikaris, P. (2020). Understanding and mitigating gradient pathologies in physics-informed neural networks. *arXiv preprint arXiv:2001.04536*.

[14] Roggeveen, J. V., & Brenner, M. P. (2025). Meshless solutions of PDE inverse problems on irregular geometries. *arXiv preprint arXiv:2510.25752*.

[15] Han, J., Jentzen, A., & E, W. (2018). Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34), 8505–8510.

[16] Flores, P., Graf, O., Protopapas, P., & Pichara, K. (2025). Improved uncertainty quantification in physics-informed neural networks using error bounds and solution bundles. *arXiv preprint arXiv:2505.06459*.

[17] Ren, S., Padilla, W. J., & Malof, J. (2020). Benchmarking deep inverse models over time, and the neural-adjoint method. In *Advances in Neural Information Processing Systems (NeurIPS)*, 33.

[18] Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). Neural ordinary differential equations. In *Advances in Neural Information Processing Systems (NeurIPS)*, 31. *arXiv preprint arXiv:1806.07366*.

[19] Machine Learning & Simulation. (2021). Adjoint state method for an ODE — Adjoint sensitivity analysis [Video]. *YouTube*. `https://www.youtube.com/watch?v=k6s2G5MZv-I`

[20] Brunton, S. (2024). Neural ODEs (NODEs) [Physics informed machine learning] [Video]. *YouTube*. `https://www.youtube.com/watch?v=nJphsM4ob0k`

[21] Brunton, S (2024) Physics Informed Neural Networks (PINNS) [Physics Informed Machine Learning] [Video]. *YouTube*. `https://www.youtube.com/watch?v=-zrY7P2dVC4`