Trabajo de Fin de Grado

Curso 2024-2025

Programación probabilística: lenguajes y métodos de inferencia

Probabilistic programming: languages and inference methods

Autor: **Rebeca Maestro López**

Director: **Dr. Miguel Palomino Tarjuelo**

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

NOTA ASIGNADA POR EL TRIBUNAL: 8.5

Índice

1. Resumen

Este trabajo ofrece una visión integral de los modelos probabilísticos y su implementación práctica en Pyro, un marco de programación probabilística basado en PyTorch. Comienza con una introducción a conceptos fundamentales como la inferencia bayesiana, inferencia variacional y métodos de Montecarlo, abordando también el método del descenso del gradiente estocástico. Luego, se presentan las herramientas clave de Pyro, incluyendo primitivas de modelaje, guías para la inferencia variacional, clases como `SVI` y `PyroOptim`, y estrategias de optimización. Finalmente, se exploran aplicaciones en dos áreas fundamentales: los modelos generativos, con un enfoque en los autocodificadores variacionales, y las redes neuronales bayesianas. En ambos casos, se discuten diferencias clave con los métodos clásicos, su implementación en Pyro y los resultados obtenidos a través de experimentos prácticos. Este trabajo enfatiza la flexibilidad y precisión que Pyro aporta a la modelización probabilística en aprendizaje automático.

Palabras clave: Modelos probabilísticos, Inferencia bayesiana, Inferencia variacional, Montecarlo, Pyro, Redes neuronales bayesianas, Autocodificadores variacionales, Gradiente estocástico, Modelos generativos, Optimización probabilística

2. Abstract

This document provides a comprehensive overview of probabilistic models and their practical implementation in Pyro, a probabilistic programming framework built on PyTorch. It begins by introducing fundamental concepts such as Bayesian inference, variational inference, Monte Carlo methods and stochastic gradient descent. Next, it explores Pyro's key tools, including modeling primitives, guides for variational inference, classes like `SVI` and `PyroOptim`, and optimization strategies. Finally, applications in two fundamental areas are explored: generative models, focusing on variational autoencoders, and Bayesian neural networks. For both cases, key differences from classical methods, their implementation in Pyro, and experimental results are discussed. This work highlights the flexibility and precision Pyro brings to probabilistic modeling in machine learning.

Key words: Probabilistic models, Bayesian inference, Variational inference, Monte Carlo, Pyro, Bayesian neural networks, Variational autoencoders, Stochastic gradient, Generative models, Probabilistic optimization

3. Introducción

La programación probabilística es una rama de la inteligencia artificial que permite modelar y resolver problemas que involucran incertidumbre utilizando conceptos probabilísticos y herramientas de programación.

Los antecendentes y bases de esta son los modelos probabilísticos junto con la inferencia estadística y la evaluación que se realiza en el aprendizaje automático. Por ello la programación probabilística se encuentra entre los campos del aprendizaje automático, la estadística y los lenguajes de programación. Utilizando la semántica formal, los compiladores y otras herramientas de programación consigue construir evaluadores de inferencia eficientes para modelos y aplicaciones del aprendizaje automático utilizando la teoría y los algoritmos de inferencia de la estadística.

En este Trabajo de Fin de Grado (TFG), se aborda el estudio y la aplicación de la programación probabilística en el contexto de la inferencia bayesiana. La capacidad de modelar la incertidumbre de manera explícita y razonar sobre ella es fundamental en muchas aplicaciones del mundo real y la programación probabilística ofrece un marco formal para este propósito.

El objetivo principal de este TFG es dar una visión general de los métodos de inferencia que se aplican en la programación probabilística. Para lograr este objetivo, se explorarán a través de ejemplos los diversos enfoques y técnicas de programación probabilística con el fin de desarrollar soluciones efectivas y eficientes para el ejemplo en cuestión.

En la siguiente sección, se proporcionará una visión general de los conceptos fundamentales de la programación probabilística.

4. Conceptos previos

Asumiendo que se tiene cierta base en probabilidad recordamos algunos conceptos previos para una mejor comprensión de la materia en cuestión.

Modelos probabilísticos

En primer lugar, los modelos probabilísticos son aquellos que nos permiten relacionar distintos aspectos de un problema a través de distribuciones de probabilidad conjunta sobre unas variables aleatorias. En general consta de observaciones X que se tienen sobre las variables Y (el problema) y los parámetros θ para las distribuciones.

El modelo más sencillo donde encontramos estos elementos es una regresión lineal bayesiana. Por ejemplo, si buscamos comprender la relación entre las calificaciones de un examen parcial y las de un examen final, podríamos explicar nuestros datos con la función $Y = wX+\epsilon_\theta$. En este caso Y es la calificación final y X es la calificación del examen parcial. El término ϵ_θ lo hemos introducido para reflejar el ruido y está parametrizado por una distribución que depende de θ. De esta forma se refleja la incertidumbre en nuestra estimación.

Distribución a priori

En segundo lugar, el concepto de distribución *a priori* se utiliza enormemente para representar la distribución o creencia que se tenía antes de evaluar las observaciones y tener nueva información. Se requieren en general varias distribuciones de probabilidad condicionada (que ayudan a comprender las relaciones que hay entre las variables) para describir un modelo.

Además, no todas las variables son medibles o constituyen el objetivo del problema y por ello se diferencia entre las variables latentes sobre las que se tiene una distribución a priori y las variables observables que describen los datos. Siguiendo con el ejemplo de las calificaciones, estas serían las variables observables, como también lo pueden ser las horas de estudio o la asistencia a clase. Mientras que el conjunto de variables latentes puede estar formado por el estrés, la salud mental o la motivación intrínseca por esos exámenes.

De esta forma las funciones de densidad conjunta que describen el problema suelen tener la siguiente estructura

$$p_\theta(x, z) = p_\theta(x|z)p_\theta(z)$$

donde $p_\theta(z)$ es la distribución a priori sobre las variables latentes y $p_\theta(x|z)$ es la probabilidad condicionada o verosimilitud (*likelihood*).

Inferencia bayesiana

Por último introducimos brevemente el concepto de inferencia bayesiana que seguiremos desarrollando más adelante. Esta es una forma de inferencia estadística basada en el teorema

de Bayes, que combina distribuciones de probabilidad con los datos observados para obtener otras a posteriori. Recordamos la fórmula de Bayes para mostrar la relación entre los sucesos A y B (que será el dato observado):

$$p(A|B) = \frac{p(B|A) \cdot p(A)}{p(B)}$$

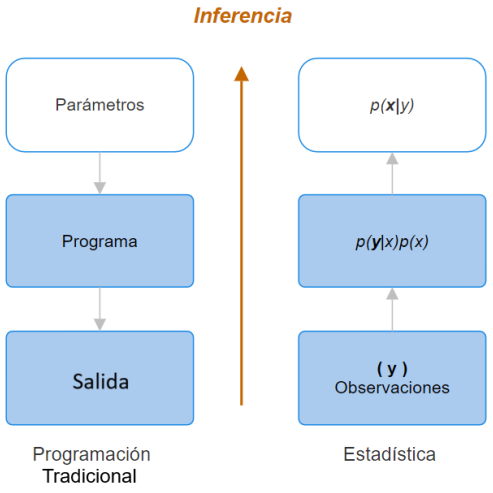
La fórmula esta formada por tres términos:

- La probabilidad $p(B|A)$, es la distribución de probabilidad del suceso B habiéndose dado el suceso A.
- La distribución de probabilidad del suceso A, $p(A)$, independiente de cualquier observación.
- La distribución de probabilidad marginal, $p(B)$, es la distribución de probabilidad del dato observado B independiente de cualquier otro suceso.

5. Métodos de inferencia

Siguiendo con los conceptos introducidos previamente, vemos como la programación probabilística trata de realizar inferencia bayesiana utilizando las herramientas de la informática. Estas son los lenguajes de programación, para la denotación de modelos; y los algoritmos de inferencia estadística, para calcular la distribución condicionada de las entradas del programa que podrían haber dado lugar a la salida observada.

En la siguiente imagen buscamos ilustrar esto último mostrando cómo un programa probabilístico no sigue el procedimiento habitual y se 'invierte' con la inferencia.



En un programa tradicional, el flujo comienza con la entrada, aplica una serie de instrucciones o reglas específicas y produce una salida determinista y concreta. La estructura se basa en pasos definidos y conocidos que transforman los datos iniciales en un resultado final.

Por el contrario, en la inferencia estadística y la programación probabilística, el flujo se invierte. Comienza con un conjunto de observaciones y un modelo que describe la relación probabilística entre las variables, pero los valores iniciales de algunas variables son desconocidos. El objetivo es, entonces, inferir las posibles entradas o parámetros que podrían haber producido los resultados observados. En otras palabras, en lugar de avanzar de causa a efecto, como en un programa tradicional, se trabaja de efecto a causa, ajustando el modelo para hacer plausible las observaciones.

Volviendo a la estadística, una vez hemos especificado el modelo podemos aplicar la inferencia bayesiana que nos muestra cuánto se adapta nuestro modelo a los datos observados

$$p_\theta(x) = \int p_\theta(x, z)\, dz$$

De aquí se deriva la distribución posterior

$$p_\theta(z|x) = \frac{p_\theta(x,z)}{\int p_\theta(x,z)\,dz}$$

a partir de la cual también se consigue una distribución posterior predictiva

$$p_\theta(x'|x) = \int p_\theta(x'|z)p_\theta(z|x)\,dz$$

Además, es deseable aprender los parámetros θ que maximizan la probabilidad marginal de nuestros modelos a partir de los datos observados x:

$$\theta_{max} = \arg\max_\theta p_\theta(x) = \arg\max_\theta \int p_\theta(x,z)\,dz$$

Calcular estas distribuciones puede ser realmente costoso y a menudo imposible, por lo que se han desarrollado algoritmos de inferencia que las aproximan y que veremos a continuación. Para realizar estos algoritmos generalmente las distribuciones que se incluyen en los lenguajes de programación probabilísticos para modelar son aquellas que se pueden muestrear eficientemente, son diferenciables y de las que se puede calcular eficientemente la densidad puntual.

5.1. Inferencia Variacional

Este método ofrece un esquema unificado para encontrar θ_{max} y calcular una aproximación manejable $q(z)$ a la verdadera distribución posterior desconocida $p_{\theta_{max}}(z|x)$ al convertir las integrales intratables en la optimización de un funcional de p y q.

La inferencia variacional aproxima la verdadera distribución posterior buscando en un espacio de distribuciones plausibles para encontrar la que sea más similar a la verdadera distribución posterior de acuerdo a alguna medida o divergencia. Se llama distribución variacional a la distribución o conjunto de distribuciones que se usan para aproximar la buscada. Se suelen detonar por q, q_ϕ o $q_\phi(z|x)$, donde ϕ son los parámetros variacionales. Por ejemplo, en el caso de un espacio de distribuciones gaussianas los parámetros variacionales serían la media y la desviación típica $\phi = \{\mu, \sigma\}$.

La medida más usada es la divergencia de Kullback-Leibler $\mathrm{KL}(q(z)||p_\theta(z|x))$ y optimizando la aproximación se obtiene la solución

$$q^*(z) = \arg\min_{q(z)\in\mathcal{Q}} \{KL(q(z)\|p(z|x))\},$$

donde $\mathcal{Q}$ es la familia de distribuciones variacionales. La divergencia de Kullback-Leibler (KL) entre las dos distribuciones está definida como:

$$KL(q||p) = \int q(z)\log\frac{q(z)}{p(z|x)}\,dz = \mathbb{E}\left[\log\frac{q(z)}{p(z|x)}\right]$$

No obstante, calcular esta expresión requiere conocer de antemano la distribución posterior, lo que representa un desafío significativo. Aplicando el teorema de Bayes, la KL-divergencia puede reformularse como la diferencia entre una constante intratable (independiente de q_ϕ) y un término tratable, conocido como el límite inferior de la evidencia (ELBO), que se definirá a continuación. Por lo tanto, maximizar este término tratable resulta equivalente a minimizar la KL-divergencia original.

Desarrollando la expresión anterior por el lado derecho se obtiene el ELBO:

$$\begin{aligned} KL(q(z)||p(z|x)) &= \mathbb{E}_q[\log q(z)] - \mathbb{E}_q[\log p(z|x)] \\ &= \mathbb{E}_q[\log q(z)] - \mathbb{E}_q[\log p(z,x)] + \mathbb{E}_q[\log p(x)] \\ &= \mathbb{E}_q[\log p(x)] - \{\mathbb{E}_q[\log p(z,x)] - \mathbb{E}_q[\log q(z)]\} \end{aligned}$$

Luego,

$$KL(q(z)||p(z|x)) = \log p(x) - \text{ELBO}(q)$$

donde $\text{ELBO}(q)$ depende de $p(z,x)$ pero no de $p(x)$. También se puede demostrar que:

$$\text{ELBO} = \mathbb{E}_{q(z|x)}[\log p(x|z)] - \text{KL}(q(z)||p(z))$$

El ELBO es una función objetivo que se descompone en dos términos fundamentales:

- El primer término corresponde a la verosimilitud (o log-verosimilitud), que mide como de bien la distribución variacional $q(z)$ explica los datos observados x. Este término evalúa la capacidad del modelo para ajustar los datos, lo cual es crucial para que el modelo sea útil en la práctica.
- El segundo término es la divergencia KL negativa entre la distribución variacional $q(z)$ y la distribución de pesos previa $p(z)$. Este término introduce una penalización que regula cuánto se desvía la distribución aproximada $q(z)$ de la distribución previa $p(z)$. En otras palabras, asegura que el modelo no se ajuste excesivamente a los datos observados (es decir, evita el sobreajuste) al imponer que $q(z)$ respete las propiedades del prior $p(z)$.

Esta descomposición ilustra cómo el ELBO actúa como un balance entre la verosimilitud (ajustarse bien a los datos) y el valor del prior (regularizar el modelo). La verosimilitud incentiva a que un modelo sea lo más explicativo posible respecto a los datos. Sin embargo, si solo maximizamos este término, el modelo puede sobreajustarse. Por otro lado, la penalización KL asegura que el modelo permanezca en una región razonable en el espacio de distribuciones, de acuerdo con nuestras creencias previas (el prior). Este balance es lo que permite que el ELBO funcione como una aproximación robusta para optimizar modelos probabilísticos complejos.

Para maximizar el ELBO y, en consecuencia, minimizar la divergencia KL, se utilizan diversos métodos de optimización que veremos a continuación.

5.2. Montecarlo

Los métodos de Montecarlo emplean números (pseudo-)aleatorios, que serán muestras que se distribuyen aproximadamente de acuerdo con la distribución posterior. Después, las muestras se promedian para estimar tanto la distribución posterior como las expectativas posteriores. Una de las fortalezas fundamentales de los métodos de Montecarlo es que la tasa de mejora en función del número de muestras N, ilustrada por el teorema central del límite (normalidad asintótica), no depende de la dimensión del parámetro.

Uno de los métodos de Montecarlo más utilizados es el basado en cadenas de Markov (MCMC). Las cadenas de Markov son secuencias de variables en las que cada valor generado depende solo del valor anterior. Construyendo una cadena de Markov que tiene la distribución deseada como su distribución de equilibrio, se puede obtener una muestra de dicha distribución registrando los estados de la cadena. Cuantos más pasos se incluyan, más se acercará la distribución de la muestra a la distribución real deseada.

Los métodos MCMC son especialmente útiles para muestrear distribuciones complejas, donde calcular una distribución de probabilidad exacta es intratable. Entre las técnicas MCMC más comunes se encuentran el algoritmo de Metropolis-Hastings y el muestreo de Gibbs, que permiten explorar el espacio de parámetros eficientemente al generar una secuencia de muestras que, al cabo de muchas iteraciones, sigue la distribución objetivo. El algoritmo Hamiltonian Monte Carlo (HMC) es otro ejemplo de los algoritmos de Metropolis-Hasting para distribuciones continuas. Está diseñado con un esquema ingenioso para generar una nueva propuesta, con el objetivo de asegurar que se rechacen la menor cantidad posible de muestras y que exista la menor correlación posible entre las mismas. Además, el tiempo de *burn-in* (calentamiento) del HMC es extremadamente corto en comparación con el algoritmo estándar de Metropolis-Hasting.

La mayoría de los paquetes de software para estadística bayesiana implementan el muestreador No-U-Turn (abreviado como NUTS), que es una mejora del algoritmo clásico HMC, permitiendo ajustar automáticamente los hiperparámetros del algoritmo en lugar de configurarlos manualmente.

Gracias a su flexibilidad, los métodos MCMC se emplean en una amplia variedad de campos, desde la inferencia bayesiana hasta el aprendizaje de modelos en alta dimensión, permitiendo estimar propiedades de distribuciones complejas a partir de simulaciones iterativas.

5.3. Descenso del gradiente estocástico

En este apartado, presentamos una introducción al método de descenso del gradiente estocástico (SGD, del inglés *Stochastic Gradient Descent*), un enfoque fundamental en el aprendizaje automático. Este método es ampliamente utilizado en problemas de optimización donde los conjuntos de datos son masivos y calcular el gradiente exacto de la función objetivo resulta computacionalmente costoso.

A continuación, describimos en detalle este método y las condiciones necesarias para ga-

rantizar su convergencia.

Supongamos que tenemos una función objetivo diferenciable $F : \mathbb{R}^n \to \mathbb{R}$, que deseamos minimizar. Formalmente, buscamos resolver el problema:

$$\min_{w \in \mathbb{R}^n} F(w).$$

En lugar de calcular el gradiente real de F, lo cual puede ser computacionalmente prohibitivo en escenarios con grandes volúmenes de datos, empleamos un estimador $g(w; \eta)$. Este estimador aproxima el gradiente real $\nabla F(w)$ utilizando un subconjunto de datos y una variable aleatoria η, que introduce un componente de ruido en el cálculo. Este enfoque es especialmente útil en aprendizaje automático, donde los datos suelen ser extremadamente numerosos.

El método SGD actualiza iterativamente los parámetros $w \in \mathbb{R}^n$ para aproximarse a un mínimo de $F(w)$. En cada iteración k, los parámetros se actualizan según la regla:

$$w_{k+1} = w_k - \alpha_k g(w_k; \eta_k),$$

donde:

- α_k es la tasa de aprendizaje, un escalar positivo que controla el tamaño del paso en la dirección del gradiente aproximado.
- $g(w_k; \eta_k)$ es el estimador del gradiente en la iteración k, calculado a partir de un subconjunto de datos seleccionado aleatoriamente, introduciendo la componente estocástica η_k.

Este procedimiento permite actualizar los parámetros de manera eficiente, incluso en conjuntos de datos masivos, ya que evita calcular el gradiente real en cada iteración.

Un caso típico ocurre en redes neuronales, donde la función de pérdida $L(w)$ puede expresarse como una suma de contribuciones individuales de cada registro del conjunto de datos. Siendo n el número de registros o neuronas. Por ejemplo:

- Para la norma euclidiana al cuadrado:

$$L(w) = \|N(X|w) - Y\|^2 = \sum_{i=1}^{n} l_i(w), \quad l_i(w) = \big(N(X|w)_i - Y_i\big)^2.$$

En este caso, calcular el gradiente real (o completo) $\nabla L(w)$ es computacionalmente costoso porque n suele ser enorme. En cambio, calcular el gradiente parcial $\nabla l_i(w)$ para un único registro i, o un subconjunto pequeño de ellos, resulta mucho más eficiente.

Seleccionando aleatoriamente un índice i o un subconjunto pequeño de datos, podemos garantizar que $\mathbb{E}[g(w)] = \nabla L(w)$, donde $g(w)$ es un estimador insesgado del gradiente. Esto proporciona la base para el uso de SGD.

Sin embargo, este método presenta limitaciones: al usar solo un registro (o un tamaño de muestra muy pequeño), la varianza del estimador $g(w)$ puede ser considerablemente alta, lo que afecta negativamente la convergencia.

Para mitigar este problema, se utiliza el método de descenso del gradiente mini-batch o semi-estocástico. En este caso, se selecciona un subconjunto aleatorio de datos (denominado *lote* o *batch*) en cada iteración. Si el conjunto de datos completo contiene n registros y dividimos los datos en m lotes de tamaño $|A|$, entonces el gradiente estimado se calcula como:

$$g(w) = \frac{n}{|A|}\nabla\Big(\sum_{j\in A_k} l_j(w)\Big),$$

donde A_k es el k-ésimo lote de datos, y $|A|$ representa el tamaño del lote, que se mantiene constante.

La actualización de los parámetros w se realiza entonces como:

$$w_{k+1} = w_k - \alpha\frac{n}{|A|}\nabla\Big(\sum_{j\in A_k} l_j(w)\Big)(w_k).$$

Generalmente los lotes se extraen del conjunto de datos sin reemplazo, es decir, no es posible que se reutilice el lote ni los datos que contenía. El algoritmo recorre todos los lotes del conjunto de entrenamiento hasta que ha utilizado todos los datos, momento en el cual vuelve a comenzar a muestrear y dividir en lotes del conjunto de datos completo. Un recorrido completo por todo el conjunto de datos de entrenamiento se denomina época (*epoch*). Un lote puede ser tan pequeño como un solo ejemplo o tan grande como el conjunto de datos completo.

El uso de lotes reduce la varianza del estimador y permite aprovechar eficientemente las capacidades de paralelización de hardware moderno, como las GPUs.

Por otro lado, es necesario mencionar el algoritmo Adam, basado en este método, propuesto por Diederick P.Kingma y Jimmy Lei Ba en 2015, que tiene una mejor y más rapida convergencia que otros métodos existentes basados en el descenso del gradiente estocástico. Este algoritmo, que es el optimizador usado en los siguientes ejemplos, se caracteriza porque su valor de entrenamiento no es constante y en cada iteración el lote (que se define en el descenso del gradiente semi-estocástico) es escogido aleatoriamente. Esta actualización permite una mejor convergencia cuando se está cerca del mínimo local [ver detalles en (Kingma, P., y ba, 2015)].

En el contexto de la inferencia variacional, esta técnica también es crucial ya que permite optimizar funciones de probabilidad complejas en espacios de alta dimensión. En este caso, $g(w;\eta)$ no solo estimaría el gradiente de una función de pérdida, sino también de una función de evidencia que permite ajustar distribuciones aproximadas para modelar incertidumbres en parámetros de modelos complejos.

6. Introducción a Pyro

Pyro va a ser el lenguaje de programación probabilística escogido para desarrollar los ejemplos y probar los distintos metodos de inferencia. Pyro es un lenguaje de programación basado en Python y en PyTorch, cuyo principal motor de inferencia es la inferencia variacional estocástica. Este consigue convertir cálculos abstractos probabilísticos en problemas concretos de optimización que se resuelven con el método estocástico de gradiente descendente en PyTorch, consiguiendo así aplicar los métodos probabilísticos a modelos que eran previamente intratables y a grandes conjuntos de datos.

6.1. Primitivas de modelaje

Los modelos probabilísticos en Pyro son especificados como funciones de Python mediante unas funciones primitivas especiales. Al ser abstractas y flexibles, los comportamientos pueden variar mucho según el uso que les demos. Esto permite al usuario enfocarse en la lógica del modelo en lugar de en los detalles de bajo nivel de implementación. Las distintas piezas que conforman el modelo (`model()`) son:

- Variables aletorias latentes z, que se modelan con la primitiva `pyro.sample` donde el primer argumento en cada llamada es el nombre de la variable.
- Variables observadas x, que se modelan con la primitiva `pyro.sample` con el argumento clave `obs` donde se le pasan los datos que son las observaciones.
- Parámetros de aprendizaje θ, con `pyro.param`.
- *Plates* con `pyro.plate` para modelar los datos de manera independiente o como manejadores de contexto.

6.2. Guía

La idea básica detrás de la guía (o *guide*) es introducir una distribución parametrizada, donde los parámetros de esta distribución se conocen como parámetros variacionales. Esta distribución se denomina distribución variacional en la literatura, pero en Pyro se le llama simplemente guía, para simplificar. La guía sirve como una aproximación a la distribución posterior $p(z|x)$, lo que significa que buscamos encontrar en un espacio de distribuciones restringido (definido por los parámetros variacionales) la distribución que se aproxime lo mejor posible a la verdadera distribución posterior del modelo.

Al igual que el modelo, la guía se codifica como una función estocástica (`guide()`) que contiene declaraciones de `pyro.sample` y `pyro.param`. Sin embargo, la guía no incluye datos observados, ya que debe ser una distribución propiamente normalizada (sin condicionamientos explícitos en los datos observados). En Pyro, el modelo (`model()`) y la guía (`guide()`) deben tener la misma estructura de llamada, es decir, ambas funciones deben recibir los mismos argumentos, lo que permite mantener una estructura consistente en ambos.

En el marco de la inferencia variacional, el **modelo** y la **guía** tienen roles complementarios:

El modelo, denotado por $p(x, z)$, describe la historia generativa de los datos observados (x) y las variables latentes (z). Especifica cómo los datos x podrían haber sido generados a partir de las variables latentes z, utilizando distribuciones conjuntas. En términos de probabilidad, el modelo define:

$$p(x, z) = p(z)p(x \mid z),$$

donde $p(z)$ es la distribución a priori (*prior*) sobre las variables latentes, y $p(x \mid z)$ es el modelo de observación que describe cómo los datos observados x son generados condicionalmente a partir de z.

La guía, denotada por $q_\phi(z)$, aproxima la distribución posterior $p(z \mid x)$, que es generalmente intratable en la mayoría de los modelos generativos complejos. Su objetivo principal es proporcionar una distribución $q_\phi(z)$ que sea:

- Fácil de muestrear y calcular.
- Lo más cercana posible a la verdadera distribución posterior $p(z \mid x)$.

En términos prácticos, la guía es un modelo variacional que diseñamos específicamente para realizar inferencia sobre z.

En este sentido, el modelo cuenta una "historia generativa" de cómo los datos podrían haberse producido, mientras que la guía es una herramienta práctica para realizar inferencia sobre los valores de las variables latentes en esa historia. Dado que la guía es una aproximación de la posterior, debe proporcionar una densidad de probabilidad conjunta válida sobre todas las variables latentes del modelo.

6.3. Inferencia Variacional en Pyro: clase SVI

En Pyro, la clase `SVI` (*Stochastic Variational Inference*) encapsula las herramientas necesarias para llevar a cabo inferencia variacional. Para usar `SVI`, el usuario debe proporcionar tres elementos fundamentales:

- **El modelo**: define la estructura probabilística de los datos.
- **La guía**: una distribución variacional que aproxima la posterior.
- **El optimizador**: que actualiza los parámetros de la guía para minimizar la pérdida.

Con estos tres elementos, podemos crear una instancia de `SVI` que realice optimización utilizando el objetivo ELBO. Para ello, el código sería:

```
import pyro
from pyro.infer import SVI, Trace_ELBO

svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
```

Aquí, `Trace_ELBO` especifica que el objetivo de la optimización será maximizar ELBO, lo cual se traduce en minimizar la pérdida entre la guía y la posterior real.

Métodos de SVI: `step()` y `evaluate_loss()`

La clase `SVI` proporciona dos métodos principales:

- `step()`: realiza un paso de gradiente y devuelve una estimación de la pérdida (es decir, el negativo de ELBO). Los argumentos que se pasen a `step()` se envían directamente a `model()` y a `guide()`, permitiendo que ambos usen los mismos datos de entrada.
- `evaluate_loss()`: calcula una estimación de la pérdida sin actualizar los parámetros (sin paso de gradiente). Al igual que `step()`, los argumentos de entrada se transmiten a `model()` y `guide()`.

Uso del Argumento `num_particles`

Para el caso donde la pérdida es ELBO, ambos métodos `step()` y `evaluate_loss()` aceptan un argumento opcional llamado `num_particles`. Este parámetro define el número de muestras que se utilizan para calcular la pérdida (en el caso de `evaluate_loss`) o para calcular la pérdida y el gradiente (en el caso de `step()`). Este ajuste permite un mayor control sobre la precisión de la estimación de la pérdida en función del número de muestras, lo cual es útil para mejorar la estabilidad y precisión del proceso de inferencia variacional.

6.4. Optimización en Pyro

En Pyro, el modelo y la guía son funciones estocásticas que pueden tener cualquier estructura, siempre que cumplan con dos condiciones:

- La guía no debe contener declaraciones `pyro.sample` con el argumento `obs`, ya que no se permite incluir datos observados en la guía.
- Tanto `model` como `guide` deben tener la misma estructura de llamada, es decir, deben recibir los mismos argumentos.

La flexibilidad de Pyro permite que las ejecuciones de `model()` y `guide()` se comporten de maneras muy diferentes. Por ejemplo, puede suceder que algunas variables latentes y parámetros aparezcan solo en ciertas ejecuciones. Además, durante la inferencia, es posible que algunos parámetros se generen dinámicamente. Esto implica que el espacio de optimización, que está determinado por los valores del modelo y la guía, puede crecer y cambiar en tiempo real.

Para manejar este nivel de dinamismo, Pyro necesita crear un optimizador para cada nuevo parámetro en el momento exacto en que aparece durante el aprendizaje. Por suerte, PyTorch cuenta con una biblioteca de optimización ligera (`torch.optim`) que se adapta perfectamente a estas necesidades.

La Clase `PyroOptim`

Todo el manejo de optimización dinámica en Pyro se controla mediante la clase `PyroOptim`. Esta clase es esencialmente una interfaz de Pyro que envuelve a los optimizadores de PyTorch. `PyroOptim` recibe dos argumentos:

- `optim_constructor`: el constructor del optimizador de PyTorch que se desea utilizar (por ejemplo, `torch.optim.Adam`).
- `optim_args`: una especificación de los argumentos que necesita el optimizador (como la tasa de aprendizaje).

Durante la optimización, cada vez que aparece un nuevo parámetro, `optim_constructor` crea una nueva instancia de un optimizador elegido, configurado con los argumentos proporcionados en `optim_args`. Esto hace posible que Pyro gestione optimizadores de forma flexible, incluso cuando los parámetros cambian dinámicamente durante la inferencia.

6.5. Ejemplo

Con el ejemplo a continuación mostramos el uso de los principales elementos de Pyro. Supongamos que tenemos que estimar la probabilidad de que salga un '6' en un dado trucado. Simulamos los datos con 50 lanzamientos y guardamos la cantidad de veces que sale el 6.

```
data_t = torch.tensor([3, 4, 6 ... 4, 2])
data = torch.tensor([1 if x == 6 else 0 for x in data_t)
```

Para el modelo sabemos que hay una probabilidad desconocidad de que salga un 6 y que cada lanzamiento se modela con una distribución *Bernoulli* condicionada por z.

```
def model(data):
    # Probabilidad latente de obtener un "6" (z ~ Uniform(0, 1))
    z = pyro.sample("z", dist.Uniform(0, 1))

    for i in range(len(data)):
        pyro.sample(f"obs_{i}", dist.Bernoulli(z), obs=data[i])
```

Para la guía, existen varias distribuciones que podemos elegir para aproximar la posterior. Una opción es utilizar la función delta, una distribución *puntual* que siempre devuelve el

mismo valor. En este caso, estamos aproximando la posterior con un único valor fijo, el cual es aprendido durante la optimización.

Aunque este enfoque simplifica los cálculos, ya que no introduce variabilidad en z, es útil porque captura la probabilidad más plausible para z. Además, funciona porque la guía sólo necesita encontrar un valor óptimo para z que maximice la probabilidad de los datos observados bajo el modelo generativo. Sin embargo, tiene la limitación de no capturar la incertidumbre en z, lo cual puede ser problemático si los datos son insuficientes o contienen ruido.

La opción más común y que veremos en este ejemplo es utilizar una distribución Beta. Este enfoque permite capturar la incertidumbre, lo que significa que el modelo no solo aprende un valor medio para z, sino también la dispersión de la distribución posterior. Además, la distribución Beta ofrece una mayor flexibilidad, ya que puede representar distribuciones complejas y reflejar la confianza en los datos observados.

```
# Definimos la guía (q(z)): una aproximación a la posterior p(z | x)
def guide(data):
    # Usamos dos parámetros aprendidos para definir una distribución Beta
    alpha_q = pyro.param("alpha_q", torch.tensor(2.0), constraint=positive)
    beta_q = pyro.param("beta_q", torch.tensor(2.0), constraint=positive)
    pyro.sample("z", dist.Beta(alpha_q, beta_q))
```

Finalmente usamos la clase **SVI** para minimizar la divergencia entre el modelo generativo y la guía.

```
optimizer = Adam({"lr": 1e-3})
svi = SVI(model, guide, optimizer, loss=Trace_ELBO())

# Entrenamos la guía
num_steps = 2000
for step in range(num_steps):
    loss = svi.step(data)
    if step % 500 == 0:
        print(f"Step {step} - Loss: {loss:.4f}")
```

Los valores aprendidos que Pyro devuelve son los parámetros $alpha_q$ y $beta_q$, con ellos calculamos la media y la varianza de la distribución Beta aproximada. Así los resultados obtenidos al aplicar el ejemplo son:

- **Estimación final de** z (media de la distribución Beta): 0,2724. Esto sugiere que, en promedio, se estima que el dado trucado tiene un 27,24 % de probabilidad de favorecer el número 6.
- **Varianza asociada a la estimación de** z**:** 0,0306. Este valor refleja la incertidumbre en la estimación de z, indicando que, aunque la probabilidad estimada es 0,2724, hay cierto grado de variabilidad o falta de certeza en los datos observados.

7. Aplicaciones

Pyro se utiliza en una amplia gama de aplicaciones, incluyendo modelado bayesiano (como regresión, flujos normalizadores y generación de modelos probabilísticos profundos), series temporales (pronósticos jerárquicos y modelos de espacio de estados), procesos gaussianos (optimización bayesiana y aprendizaje de kernels profundos), epidemiología (modelos para predecir la evolución del SARS-CoV-2 u otros virus) y algoritmos avanzados de inferencia (como MCMC e inferencia variacional estocástica) entre otros. Además, Pyro facilita el desarrollo de modelos personalizados gracias a su diseño modular y sus herramientas avanzadas para la programación probabilística.

En este trabajo vamos a ver con más profundidad dos de estas aplicaciones. La primera es uno de los modelos generativos más utilizados y la segunda es un analisis de las mejoras e inconvenientes de la introducción de distribuciones en las redes neuronales.

7.1. Modelos generativos: autocodificadores variacionales

7.1.1. Introducción teórica

Un autocodificador variacional (VAE, por sus siglas en inglés *variational autoencoder*) es un modelo generativo que aprende una representación comprimida de los datos en un espacio latente pero, a diferencia de los autocodificadores clásicos, esta representación es probabilística. En lugar de comprimir los datos a un conjunto de valores fijos, el VAE codifica los datos como distribuciones (normalmente gaussianas), permitiendo la generación de nuevos datos al tomar muestras de estas distribuciones.

Diferencia clave con un autocodificador clásico:

Un autocodificador clásico convierte los datos a una representación fija y luego intenta reconstruirlos; el VAE, en cambio, introduce una estructura probabilística que facilita la generación de datos nuevos al tomar muestras desde el espacio latente.

Usos en Aprendizaje Automático:

Los VAE se usan ampliamente en:

- Generación de imágenes: crear imágenes nuevas con características similares a las del conjunto de entrenamiento.
- Reducción de dimensionalidad: aprender representaciones más compactas y robustas.
- Creación de datos sintéticos: generar datos variados que mantienen propiedades del conjunto de datos original.

7.1.2. Implementación en Pyro

Como ejemplo para la implementación del VAE, utilizaremos el conjunto de datos **Fashion-MNIST**. Este dataset es una alternativa al conjunto clásico MNIST y contiene imágenes en escala de grises de tamaño 28×28, donde cada imagen representa una prenda de vestir. Las clases del conjunto de datos incluyen categorías como camisetas, pantalones, zapatillas, entre otros, y está compuesto por:

- **Conjunto de entrenamiento**: 60.000 imágenes.
- **Conjunto de prueba**: 10.000 imágenes.

El objetivo del VAE será aprender una representación latente comprimida de estas imágenes, que permita tanto la reconstrucción de las entradas como la generación de nuevas imágenes de prendas de vestir.

El VAE se compone de dos redes neuronales principales, ambas redes definidas como subclases de `torch.nn.Module`:

- **Codificador**: proyecta las imágenes de entrada aplanadas de tamaño 784 (28×28) al espacio latente, y genera dos vectores de tamaño 20: uno para la media (z_{mean}) y otro para la varianza (z_{var}) de la distribución gaussiana latente. Su arquitectura es la siguiente:
  - La función de activación es `ReLU`.
  - Capa de entrada: Recibe vectores de tamaño 784.
  - Capa oculta (`fc1`): Proyecta el vector de entrada al espacio intermedio de tamaño 400.
  - Capas de salida (`fc21` y `fc22`):
    - `fc21`: Produce un vector de tamaño 20 que representa la media (μ).
    - `fc22`: Produce otro vector de tamaño 20 que representa el logaritmo de la varianza ($\log \sigma^2$), que se pasa por una operación exponencial para obtener σ^2.
- **Decodificador**: reconstruye las imágenes originales a partir de las muestras tomadas del espacio latente (z). Su arquitectura es:
  - La función de activación es `ReLU`.
  - Capa de entrada: Recibe vectores de tamaño 20, correspondientes a las muestras del espacio latente.
  - Capa oculta (`fc3`): Proyecta el vector latente al espacio intermedio de tamaño 400, con la activación `ReLU` que introduce no linealidad.
  - Capa de salida (`fc4`): Transforma el espacio intermedio en un vector de tamaño 784, que representa las imágenes reconstruidas. Una función de activación sigmoide (`torch.sigmoid`) se aplica para obtener valores en el rango $[0, 1]$, adecuados para representar intensidades de píxeles.

La dimensión latente (20) define el tamaño del espacio en el que se representan las variables latentes (z). Este valor se considera adecuado para la base de datos utilizada, ya que permite capturar suficiente información para reconstruir las imágenes. Sin embargo, valores más grandes podrían capturar más detalles, aunque conllevan un mayor riesgo de sobreajuste y un incremento en el coste computacional.

En este modelo concreto asumimos que los datos observados (*data*) siguen una distribución normal cuya media es una muestra decodificada y generada a partir de las variables latentes (z) y la varianza tiene un valor fijo de 0,1. Esta varianza resulta adecuada en la práctica ya que con valores más altos las imágenes generadas salían muy distorsionadas.

- Para z asumimos que sigue una distribución a priori normal estándar $N(0, I)$.
- Las imágenes *img* son generadas a partir del decodificador, condicionadas en z. Esta distribución también es normal.

```
def model(self, data):
    with pyro.plate("data", data.size(0)):
        z = pyro.sample("latent", Normal(0, 1).expand([data.size(0), 20]))
        img = self.decoder.forward(z)
        pyro.sample("obs", Normal(img, 0.1), obs=data.reshape(-1, 784))
```

La guía utiliza el codificador para calcular la media y la varianza de z, y define una distribución normal para muestrear z.

```
def guide(self, data):
    with pyro.plate("data", data.size(0)):
        z_mean, z_var = self.encoder.forward(data)
        pyro.sample("latent", Normal(z_mean, z_var.sqrt()).to_event(1))
```

7.1.3. Entrenamiento

Para entrenar el VAE utilizamos la clase `SVI`, donde se pueden configurar distintas optimizaciones, en concreto usaremos la optimización con el algoritmo Adam (sección 5.3).

```
def initialize_optimizer(self, lr=1e-3):
    optimizer = Adam({"lr": lr})
    elbo = Trace_ELBO()
    return SVI(self.model, self.guide, optimizer, loss=elbo)
```

Durante el entrenamiento:

- La guía genera aproximaciones para las variables latentes z.

- El modelo evalúa la probabilidad de los datos observados y Pyro optimiza automáticamente el ELBO.

Durante cada época, el modelo procesa lotes de datos. Para cada lote, se calcula el ELBO y se ajustan los parámetros del codificador y decodificador.

```
def train(self, epoch):
    for batch_idx, (x, _) in enumerate(self.train_loader):
        loss = self.compute_loss_and_gradient(x)
        train_loss += loss
```

7.1.4. Análisis y Resultados

Una vez entrenado el modelo, se evalúa su rendimiento en dos tareas principales: la reconstrucción de imágenes de entrada y la generación de nuevas imágenes a partir de muestras del espacio latente. A continuación, se presentan y analizan los resultados obtenidos.

Reconstrucción de Imágenes

El modelo VAE reconstruye las imágenes de prueba al pasar los datos originales a través del codificador y el decodificador. En la Figura 1 se muestran ejemplos de reconstrucciones. La primera fila corresponde a las imágenes originales, mientras que la segunda fila muestra las imágenes reconstruidas por el modelo.

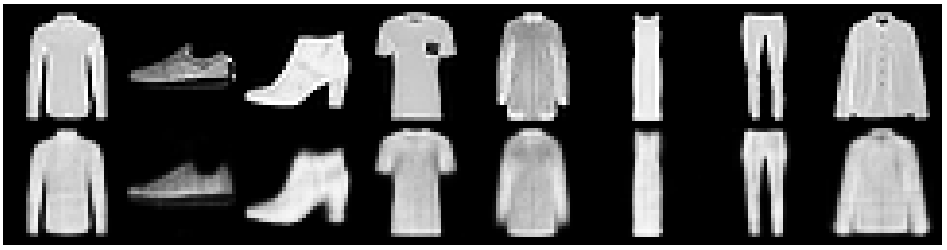


Figura 1: Reconstrucción de imágenes de FashionMNIST. Las imágenes originales (arriba) se comparan con las reconstrucciones (abajo).

El modelo logra captar las características principales de las prendas, aunque algunas reconstrucciones presentan cierta pérdida de detalle. Esto es una consecuencia natural del uso de un espacio latente comprimido, que prioriza la representación de características globales sobre los detalles finos.

Generación de Nuevas Imágenes

Además de la reconstrucción, el decodificador del VAE se utiliza para generar imágenes completamente nuevas. Estas imágenes se producen al muestrear vectores latentes de una distribución normal estándar $N(0, I)$ y pasarlos a través del decodificador.

La Figura 2 muestra ejemplos de imágenes generadas. Cada una corresponde a una muestra aleatoria del espacio latente, sin ningún dato de entrada.

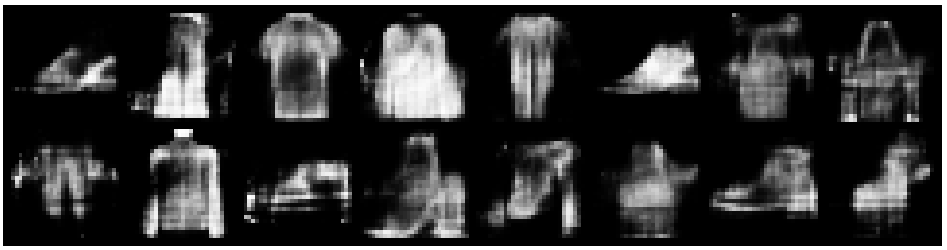


Figura 2: Imágenes generadas por el modelo VAE.

Las imágenes generadas son coherentes con las categorías del conjunto de datos original (*p. ej.*, camisetas, zapatos), demostrando que el modelo ha aprendido una representación significativa del espacio de datos.

Discusión de Resultados

- **Reconstrucción**: el modelo es capaz de reconstruir imágenes con un nivel razonable de detalle, lo que indica que ha aprendido a comprimir y descomprimir las imágenes de FashionMNIST.
- **Generación**: las imágenes generadas reflejan patrones claros y son visualmente similares a las imágenes reales del conjunto de datos. Esto muestra que el modelo ha captado la estructura estadística de los datos.
- **Limitaciones**: aunque las reconstrucciones y las imágenes generadas son efectivas, en algunos casos las imágenes pueden aparecer borrosas o carecer de detalles. Esto se debe principalmente a la simplicidad del modelo y al tamaño relativamente pequeño del espacio latente.

Análisis

Los resultados obtenidos muestran que el VAE implementado en Pyro es capaz de aprender representaciones útiles de los datos, tanto para la tarea de reconstrucción como para la generación de nuevas imágenes. Este enfoque probabilístico ofrece una manera poderosa y flexible de modelar datos complejos.

7.2. Redes neuronales bayesianas

7.2.1. Introducción teórica

Las redes neuronales bayesianas son redes neuronales que incorporan incertidumbre en sus predicciones al modelar los pesos de la red como distribuciones probabilísticas en lugar de valores fijos. Esto permite que la red no solo haga una predicción, sino que también indique el nivel de confianza o incertidumbre de la misma, lo cual es especialmente útil cuando los datos son escasos o ruidosos.

Diferencia clave con una red neuronal clásica:

En una red clásica, los pesos son valores fijos optimizados durante el entrenamiento. En una red bayesiana, los pesos son distribuciones que se ajustan para reflejar la incertidumbre, proporcionando intervalos de confianza en las predicciones.

Aplicaciones en Aprendizaje Automático:

Las redes neuronales bayesianas son útiles en:

- Clasificación en condiciones de incertidumbre: por ejemplo, en diagnósticos médicos, donde es vital conocer la confianza del modelo en sus predicciones.
- Cuantificación de riesgos: en modelos financieros o de seguridad, donde la incertidumbre de la predicción ayuda a evaluar riesgos asociados a cada resultado.

7.2.2. Implementación en Pyro

Para realizar un análisis de las redes neuronales bayesianas vamos a partir de un conjunto de datos generados por una función, sobre el cual se entrenarán los distintos modelos. Este conjunto simulará observaciones reales que contienen ruido pero que siguen la forma de una función concreta como vemos en la Figura 3. El objetivo de los modelos será predecir nuevos datos que concuerden con la función real.

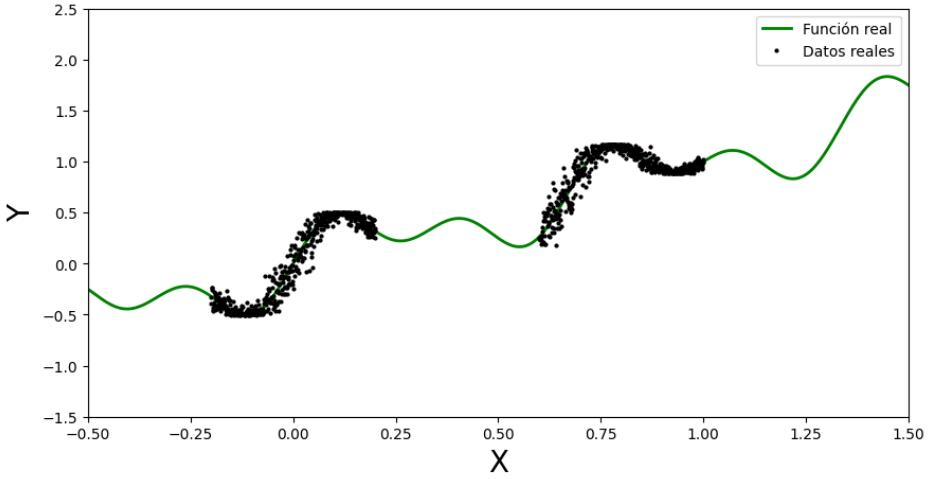


Figura 3: Datos ilustrados

Red neuronal estándar

El primer modelo es una red neuronal estándar implementada con `torch.nn.Module`. En este caso sencillo la función de activación es `nn.Tanh()` y tiene 3 capas internas con 30, 15 y 10 neuronas respectivamente.

Para el entrenamiento de esta, la función de pérdida que utilizaremos será el error cuadrático medio y la optimización con el algoritmo Adam (sección 5.3) para ajustar los pesos del modelo con una tasa de aprendizaje del $0,001$.

```
model = NN()
loss_f = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

Red neuronal bayesiana

El otro modelo es una red neuronal bayesiana que sigue una lógica similar en Pyro. La función de activación es `nn.Tanh()` y tiene 3 capas internas con la misma distribución de neuronas, donde los pesos y los sesgos son distribuciones normales de media 0 y desviación típica 5. Hemos escogido distribuciones normales por su versatilidad pero realmente se pueden inicializar con distribuciones arbitrarias ya que el objetivo del modelo no es mantener estas distribuciones iniciales sino, dados unos pesos w de una capa, aprender la distribución posterior $p(w|datos)$. En el código vemos cómo inicializar una capa.

```
class BayesianNN(PyroModule):
 def __init__(self):
    super().__init__()
    self.fc1 = PyroModule[nn.Linear](1, 30)
    self.fc1.weight = PyroSample(dist.Normal(0., 5.).expand([30, 1]))
```

```
        self.fc1.weight.to_event(2)
        self.fc1.bias = PyroSample(dist.Normal(0., 5.).expand([30]).to_event(1))

    def forward(self, x, y=None):
        x = x.reshape(-1, 1)
        x = self.tanh(self.fc1(x))
        x = self.tanh(self.fc2(x))
        x = self.tanh(self.fc3(x))
        mu = self.fc4(x).squeeze()
        sigma = pyro.sample("sigma", dist.Gamma(.5, 1))
        with pyro.plate("data", x.shape[0]):
            obs = pyro.sample("obs", dist.Normal(mu, sigma * sigma), obs=y)
        return mu
```

Por otro lado nuestro modelo se ajusta a los datos observados (y) cuando asumimos que son generados a partir de una distribución a priori condicionada por la red neuronal bayesiana. Esta es una distribución normal en la que la media se actualiza con la salida de la red y la función de activación; y la varianza es una distribución *Gamma(0.5,1)* que busca simular e inferir el ruido. Se podrían haber utilizado otras distribuciones positivas para representar la varianza como la *Exponencial* o la *Beta* pero no son tan flexibles.

Para la guía haremos uso de una herramienta muy útil de Pyro que son las autoguías. Esta realiza la inferencia variacional mediante diferenciación automática (Kucukelbir, Tran, Ranganath, Gelman, y Blei, 2016), con lo que Pyro puede generar automáticamente una guía para un modelo dado. En este caso, utilizaremos la autoguía *AutoDiagonalNormal*, que emplea una distribución gaussiana multivariada con una matriz de covarianza diagonal para aproximar la distribución posterior.

```
    model = BayesianNN()
    guide = AutoDiagonalNormal(model)
    adam = pyro.optim.Adam({"lr": 1e-3})
    svi = SVI(model, guide, adam, loss=Trace_ELBO())
```

7.2.3. Entrenamiento

Para el entrenamiento de la red neuronal seguimos el procedimiento estándar. Para las 5,000 épocas que se han establecido se predice la salida con las observaciones de entrada, se calcula la función de pérdida, después los gradientes y se ajustan los pesos de la red neuronal.

```
n_epochs = 5000
for epoch in range(n_epochs):
    model.train()
    optimizer.zero_grad()
    y_pred = model(x_train)
```

```
    loss = loss_f(y_pred, y_train)
    loss.backward()
    optimizer.step()
```

En el caso de la red neuronal bayesiana, la entrenamos con el mismo número de épocas 5,000 y con dos métodos distintos de inferencia vistos anteriormente.

En primer lugar, entrenamos la red con inferencia variacional (sección 5.1). Para calcular el gradiente del ELBO en cada época simplemente llamamos al método `step` de SVI. El argumento `data` que pasamos a SVI.step será entregado tanto al modelo como a la guía.

```
x_train = torch.from_numpy(x_obs).float()
y_train = torch.from_numpy(y_obs).float()
for epoch in range(n_epochs):
    loss = svi.step(x_train, y_train)
```

En el segundo caso, usamos el método de Montecarlo MCMC (sección 5.2) con el núcleo de NUTS (Hoffman y Gelman, 2011) para inferenciar muestras de la distribución posterior. De esta forma la red neuronal bayesiana se entrena indirectamente, se actualizan los pesos y los sesgos promediando las muestras.

```
model = BayesianNN()
nuts_kernel = NUTS(model, jit_compile=True)
mcmc = MCMC(nuts_kernel, num_samples=50)
mcmc.run(x_train, y_train)
```

7.2.4. Análisis y Resultados

Para evaluar los modelos visualizamos las muestras generadas en la predicción. Esta ha sido realizada con la misma cantidad de datos para todos los modelos y sus variantes.

Red neuronal clásica

En el caso de la red neuronal clásica se establece el modelo en modo evaluación y se predicen 1,000 muestras.

```
    # Predicción
    model.eval()
    x_test = torch.linspace(-0.5, 1.5, 1000).reshape(-1, 1)
    y_test_pred = model(x_test).detach().numpy()
```

Esto nos da los resultados de la figura 4, donde podemos observar que la red neuronal se ha sobreajustado a los datos ya que solo coincide en las observaciones proporcionadas.

La posible ventaja de este modelo está en el coste computacional que, con características similares a nivel de capas, consigue predecir 3 veces más rapido que la red neuronal bayesiana (con la inferencia variacional).

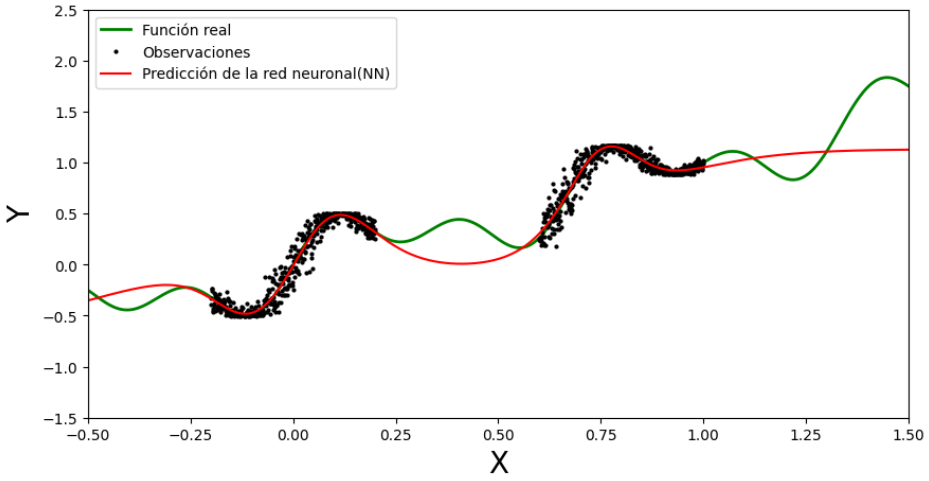


Figura 4: Comparativa con red neuronal

Red neuronal bayesiana

Para evaluar este modelo hemos usado la interfaz *Predictive* de Pyro. Hemos testeado la misma cantidad de datos y esta nos saca las predicciones obtenidas.

```
predictive = Predictive(model, guide=guide, num_samples=500)
x_test = torch.linspace(-0.5, 1.5, 1000)
preds = predictive(x_test)
```

En los resultados de la red neuronal bayesiana observamos que, aunque no termina de ajustarse a la función real, consigue dar una región de confianza (sombreada en amarillo) que cubre la mayoría de las observaciones. Además la media también refleja las curvas de la función real.

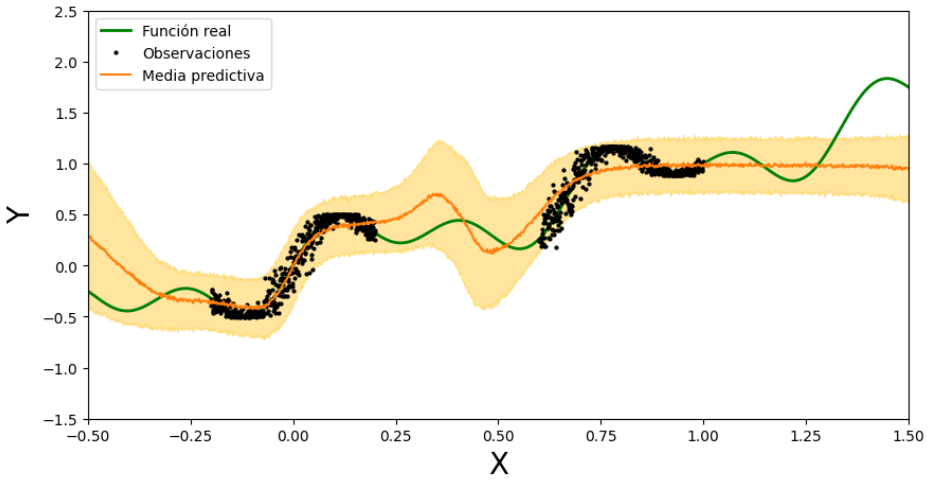


Figura 5: Comparativa con red neuronal bayesiana

También hemos comparado la red neuronal inicializándola con otros parámetros como la desviación típica o variando el número de épocas en el entrenamiento. En la figura 6 podemos ver la importancia de tener una buena distribución a priori y cómo el exceso de entrenamiento tampoco da buenos resultados. En este caso la distribución a priori de los pesos tenía una desviación típica de 4 y un entrenamiento con 15,000 épocas. Por ello observamos un mayor sobreajuste a los datos y peor predicción.

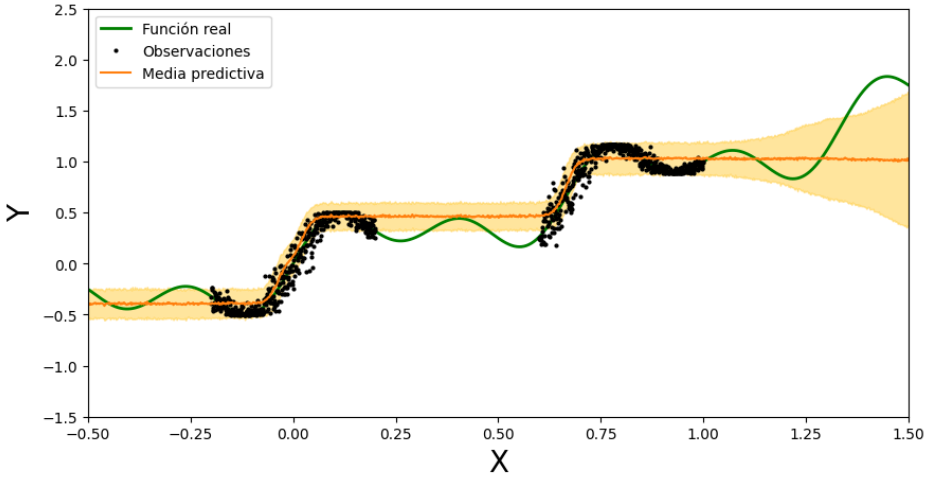


Figura 6: Comparativa de bnn normal con desviación 4

Por último vemos los resultados del método MCMC. Podemos observar la principal ventaja de los métodos de inferencia basados en MCMC que es la precisión en la predicción de la posterior. MCMC genera muestras de la verdadera distribución posterior del modelo, lo que significa que, si se ajusta correctamente, se puede obtener de manera confiable una buena representación de la posterior independientemente de su distribución.

En la figura 7 se observa como el ajuste a las observaciones es muy bueno sin descuidarse la forma de la media en los tramos donde no hay datos. Aunque no se ajuste completamente, no es lineal y la región de confianza en el tramo de la derecha es la que mejor ha cubierto la función en comparación con los otros modelos.

El mayor inconveniente de estos métodos es su coste computacional. Este se ha visto reflejado en la práctica ya que el tiempo empleado en generar estas predicciones ha sido casi 10 veces mayor que en los modelos que usan inferencia variacional.

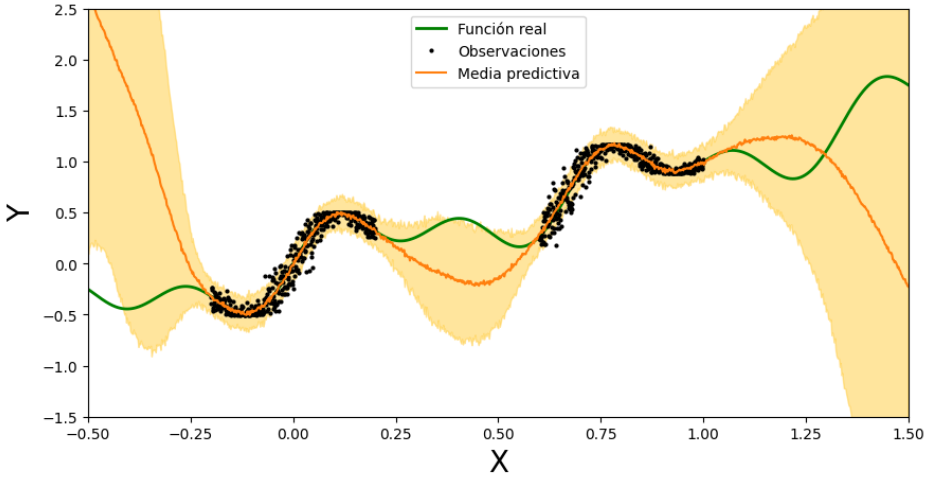


Figura 7: Comparativa con MCMC

8. Conclusiones

A lo largo de este trabajo hemos demostrado que la programación probabilística constituye una herramienta esencial para la modelización y el aprendizaje automático, ofreciendo un enfoque flexible y robusto para abordar problemas complejos. Su utilidad se amplifica con herramientas modernas como Pyro, que simplifican y facilitan tanto la implementación como el uso práctico de modelos probabilísticos avanzados.

Hemos analizado las diferencias clave entre los distintos métodos de inferencia como la inferencia variacional y los métodos basados en Monte Carlo (MCMC). Estas técnicas presentan fortalezas específicas que las hacen idóneas para distintos contextos, desde el manejo de grandes volúmenes de datos hasta la precisión en problemas con alta complejidad posterior. No obstante, hemos comprendido que no existe una técnica claramente superior; la elección del método depende específicamente del problema planteado y de sus limitaciones computacionales.

Aunque este trabajo ha priorizado una perspectiva global del ámbito de la programación probabilística, no se puede ignorar la importancia de las líneas de investigación actuales. Entre estas destacan la optimización de la inferencia en tiempo real para modelos dinámicos, el desarrollo de algoritmos de Monte Carlo con mayor eficiencia como los métodos Hamiltonianos adaptativos y la creación de modelos generativos profundos que permiten un aprendizaje más interpretativo y estructurado. Además, se están diseñando lenguajes especializados para programación probabilística, como NumPyro y Turing, que buscan maximizar la eficiencia y escalabilidad de los modelos.

En el caso de Pyro, hemos destacado cómo los modelos probabilísticos como los autocodificadores variacionales, las redes neuronales bayesianas y los modelos jerárquicos pueden implementarse para abordar problemas de inferencia compleja. Asimismo, la capacidad de integrar algoritmos de inferencia con estructuras de datos optimizadas, como tensores en PyTorch, permite procesar grandes conjuntos de datos de manera eficiente. Aplicaciones recientes incluyen avances en biología, como la modelización de redes genéticas dinámicas; en finanzas, con la predicción robusta de series temporales bajo incertidumbre; y en el procesamiento del lenguaje natural, con modelos generativos jerárquicos aplicados a la traducción automática y el análisis semántico contextual.

En definitiva, la programación probabilística no solo transforma la forma en que entendemos los datos y modelamos sistemas complejos, sino que también invita a explorar y contribuir a un área que promete seguir creciendo en los próximos años.

9. Anexo

El conjunto completo del código y los ejemplos se encuentra disponible en https://github.com/rebecmae73/TFG-Info

Referencias

Auzina, I. A., Bereska, L., Timans, A., y Nalisnick, E. (2023). Bayesian Neural Networks notebook with Pyro. (https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/DL2/Bayesian_Neural_Networks/dl2_bnn_tut1_students_with_answers.html#Bayesian-Neural-Network-with-Gaussian-Prior-and-Likelihood)

Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer.

Contributor, P. (2024). Variational Autoencoder Example. (https://pyro.ai/examples/vae.html)

Davidson-Pilon, C. (2015). *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference* . Addison-Wesley Professional.

Hoffman, M. D., y Gelman, A. (2011). The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. (https://arxiv.org/abs/1111.4246)

Kingma, P., D., y ba, J. L. (2015). Adam: A method for stochastic optimization. En *3rd international conference for learning representations.*

Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., y Blei, D. M. (2016). Automatic Differentiation Variational Inference. (https://arxiv.org/abs/1603.00788)

Martin, O. A., Kumar, R., y Lao, J. (2021). *Bayesian Modeling and Computation in Python*. Boca Raton.

Prince, S. J. (2023). *Understanding Deep Learning*. The MIT Press. (http://udlbook.com)