Foundations of Neural Networks

Notes from the specialization offered by John Hopkins University

Jose Espino

March 17, 2026

Contents

1 Introduction to Neural Networks

Neural networks, which are contained in the broader field of deep learning, sit within the rich history of artificial intelligence. AI itself as a concept emerged a lot before the appearance of modern computers—with famous mathematician Alan Turing introducing the concept of an *abstract computational machine capable of reading and writing symbols with unlimited memory* all the way back in 1935.

Historically, AI systems excelled at tasks that were easy to formalise mathematically, such as games like checkers and chess. This is because such tasks involve finite state spaces and explicit rules, which allowed traditional algorithms to perform well. Tasks like vision and natural language understanding proved challenging for AI; these are tasks that are intuitive for humans but cannot easily be described through generalisable rules. This motivated a shift towards learning from examples, similar to how humans acquire those skills over time.

Before machine learning—systems that learn patterns from data—became mainstream, the technique the majority of AI systems relied on was *kwoledge–based learning*. Knowledge–based learning consists of manually constructing rules for possible inputs the system might have to process. This approach proved difficult to scale, since rules that describe the world become too numerous, inconsistent, or ambiguous as more and more possible cases are considered. ML emerged as a response to these limitations, relying on the intrinsic patterns in data as opposed to manually–crafted rules. Early ML systems relied heavily on *feature engineering*, since humans had to design the right input for representation. For instance, early medical imaging ML models often performed poorly unless a medical professional summarized the important features first, because raw pixel data was difficult for these models to interpret.

For many tasks, it is difficult to know which features are optimal and whether they will generalise well; thus, the idea of letting the algorithm learn the best representation of the data itself appeared. *Representation learning* is the process of allowing algorithms to transform raw inputs into meangingful internal structures. An example is the autoencoder, which consists of an encoder that compresses data into a latent representation and a decoder that reconstructs the original data. The network is then trained to minimize the reconstruction error; if the autoencoder succeeds, it means the model discovered a useful internal representation of the input. This approach outperformed feature engineering and paved the way for deep learning.

Deep learning extended representation learning by stacking multiple layers of simple transformations atop one another. Each layer would then extract a progressively more abstract representation of the data. As hardware and algorithms got better, networks grew deeper and more sophisticated. Often, more layers added resulted in the model being able to capture more complex hierarchical structures. The reason why deep learning is currently thriving is the wide availability of data coupled with the dramatical increase in computing power. Today, deep learning surpasses humans in many domains, such as computer vision, natural language processing, speech recognition, strategy games, and others.

Machine learning, as a field, focuses on computational algorithms that estimate complex functions from data to explain or predict the world, typically without emphasising confidence intervals as much as classical statistics. An ML program learns from experience (E) with respect to a task (T) and performance metric (P) if performance on T, as measured by P, improves with E. Some common tasks are:

- Classification: assign discrete labels to items
- Regression: predict continuous values
- Anomaly Detection: flag rare or abnormal values
- Clustering: groups similar data points together
- among others...

Metrics tend to be task specific; for example, the performance of a model in classification may be measured through accuracy, precision, recall, F1 score, or ROC curve. On the other hand, if the task at hand were regression, performance could be measured through mean absolute error, mean square error, coefficient of determination, or others.

Experience is the data used for learning—the training set. The model iteratively adjusts its parameters using data to improve its preedictions. In *supervised learning*, the model receives input-label pairs during its training and learns to match data points to labels. In *unsupervised learning*, on the other hand, the model receives inputs only and aims to discover structure or distribution within those data points.

Another key concept is that of the **capacity** of a model; this refers to its capacity to fit a wide range of functions and often correlates with complexity. Linear models have low capacity, since they can only fit lines or planes, but models with higher exponential degrees can fit more complex patterns. A model's goal is to achieve proper generalisation, that is, to be able to produce accurate predictions on any type of data input to it. When a model fits the training data too closely, capturing noise and outliers, it becomes *overfitted*. Overfitting can usually be spotted by a large gap between a low training error and high validation error; that is to say, the model predicts data it has been trained on very well, but fails to keep that high accuracy on unseen data. When the model is unable to capture the underlying structure in data, it is considered *underfitted*. Underfitting can be identified when both training and validation errors are high—which means the model just sucks at any prediction.

Data is often split into three roles:

1. Training set: Portion of the data that the model learns from; the model parameters are changed to fit it.
2. Validation set: A separate slice of the data used to tune choices (i.e. hyperparameters) and pick between models. The model doesn't learn from it; you just check the model's performance on it to guide decisions.
3. Test set: Set of data the model is not trained on and that did not influence parameter selection either. It provides an unbiased view of the model's real-world performance on unseen data.

More data can help reduce overfitting, since makes it harder for the model to memorise noise, but this is not always feasible and will not fix poor or underrepresented data.

Hyperparameters are the configuration choices that control the training process and/or architecture of a given model. Some examples of hyperparameters are the learning rate, number of layers, regularization strength (L1/L2), number of trees in random forests, amongst others. These can be tuned through several methods, such as:

- Grid Search: It is a brute force way to find the optimal hyperparameters by trying every combination possible in a small, predefined grid. It can be slow and expensive if there are many settings, wide ranges, or settings that lie in a continuous space.
- Random Search: Instead of trying every possible combination of hyperparameters, sample combinations are randomly sampled from reasonable ranges. This is more efficient because usually one can discover good settings faster with the same budget, especially when only a few hyperparameters really matter.
- Bayesian Optimization: A smart, guided search in which a simple model of hyperparameters is built and then based on the validation scores obtained from the trials you have run. Then, the next set of hyperparameters is proposed where improvement is expected. This works best when the search space is kept reasonable and the validation metric is consistent.
- Gradient-based Optimization: If certain parameters are differentiable, we can nudge them directly by moving them in the direction that makes error decrease.
- Early Stopping: A rule of stopping to train the model once it stops improving or starts worsening on the validation set. This aims to prevent overfitting from adapting too much to the training data.
- Cross-validation with Tuning: This method involves splitting the data into k equal folds, and then, for each hyperparameter setting, you train on k-1 folds and validate on the remaining fold. You repeat k times so that every fold is used for validation at least once, and then average the validation results to judge that hyperparameter setting.

Bias is the error obtained from overly simplistic assumptions; it leads to systematic mistakes—underfitting. **Variance**, on the other hand, is the error that comes from too much sensitivity to data fluctation, especially sensitivity to noise. For example, there might be a lot of variance if one uses a very high-degree polynomial model that wiggles through noise. As you might expect, high variance leads to overfitting. Reducing one tends to increase the other, so you should aim for that *sweet spot* of lowest combined error.

An important concept in machine learning is that of **loss functions**. These functions are the ones that measure the discrepancy between predictions and targets, which the optimizer will then try to minimise. The choice of a loss function must align with the task, since the type of output will change how discrepancy is measured. Some common cost functions for regression problems are:

- Mean Squared Error: This metric is the average of the squared differences between predictions and true values. The reason why squaring is used is because it means big mistakes are punished a lot. This metric is particularly good for data without many outliers. For a single sample, the formula is:

$$\text{MSE}(y, \hat{y}) = (y - \hat{y})^2$$

  For an entire dataset, the formula is:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- Mean Absolute Error: The average of the absolute differences. It treats all errors linearly, so it is more robust to outliers than MSE. For a single sample, the formula is:

$$\text{MAE}(y, \hat{y}) = |y - \hat{y}|$$

For an entire dataset, the formula is:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

- Huber Loss: It is supposed to be a hybrid of MSE and MAE in which, for small errors it behaves like MSE by being smooth and encouraging small adjustments, whereas for large errors it behaves like MAE to limit the impact of outliers. A parameter δ decides what counts as small and what counts as large. For a single sample, the formula is:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & \text{if } |y - \hat{y}| \leq \delta \\ \delta\left(|y - \hat{y}| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases}$$

For an entire dataset, the formula is:

$$\text{Huber}_\delta = \frac{1}{N}\sum_{i=1}^{N} L_\delta(y_i, \hat{y}_i)$$

When it comes to classification, some common loss functions are:

- Binary Cross-entropy: it is used when there are two classes where class 1 has a predicted probability of p. It punishes being confident and wrong. For a single sample, the formula is:

$$\text{BCE}(y, p) = -\left[y\log(p) + (1 - y)\log(1 - p)\right]$$

For an entire dataset, the formula is:

$$\text{BCE} = \frac{1}{N}\sum_{i=1}^{N}\left(-y_i\log(p_i) - (1 - y_i)\log(1 - p_i)\right)$$

- Categorical Cross-entropy: it is used when there are more than two classes with a probability distribution of $p = (p_1, \cdots, p_k)$ over K classes and one-hot true label y. It punishes low probability being assigned to the true class. For a single sample, the formula is:

$$\text{CCE}(\mathbf{y}, \mathbf{p}) = -\sum_{k=1}^{K} y_k\log(p_k)$$

For an entire dataset, the formula is:

$$\text{CCE} = \frac{1}{N}\sum_{i=1}^{N}\left(-\sum_{k=1}^{K} y_{ik}\log(p_{ik})\right)$$

- Hinge Loss: It is used for binary classification with labels $y \in \{-1, +1\}$ and a raw score s (not a probability), for which you want s to be positive for $+1$ and negative for -1, with a margin of at least 1. The idea is that there will be no loss if the example is correct and confidently separated. Otherwise, there is a linear penalty. For a single sample, the formula is:

$$\text{Hinge}(y, s) = \max\Big(0,\ 1 - y\, s\Big)$$

$$\text{Hinge} = \frac{1}{N} \sum_{i=1}^{N} \max\Big(0,\ 1 - y_i s_i\Big)$$

When it comes to optimisation, there are also several methods that allow us to train the model. The most prominent ones are:

- Stochastic Optimization Setting: Loss surfaces are noisy due to stochastic mini-batches and data noise, and they are often non-convex. This type of optimization requires careful learning rate control.
- SGD with Momentum: Uses the exponential moving average of gradients to smooth down steps. It slows movement in unhelpful directions and allows larger effective steps in helpful ones.
- RMSProp: Uses exponential moving average of squared gradients to adapt per-parameter learning rates. It reduces step size more where gradients are large or volatile while enlarging it along more horizontal directions.
- Adam: Adaptive Moment Estimation is a combination of momentum (EMA of gradients) and RMSProp (EMA of squared gradients). It uses bias correction because moving averages start at zero and it is controlled by hyperparameters (e.g. the exponential decay rates of the EMAs). It has been shown to outperform other methods across multiple different settings.

Putting everything together, a sample workflow for using a neural network to tackle a specific problem would be:

1. Define the task and performance metric aligned to your goals.
2. Collect and prepare data; then, split t into train, validation, and test.
3. Pick a baseline model and a suitable loss function for it.
4. Optimise with a reasonable method (e.g. Adam), monitor training vs. validation curves.
5. Tune hyperparameters.
6. Control overfitting through techniques like regularisation (L1/L2, dropout), early stopping, and more data whenever possible.
7. Evaluate fairly.
8. Select the model that balances bias-variance and meets the performance metric on validation/test data.
9. Document settings, seeds, and results for reproducibility.

1.1 Deep Feedforward Networks

A deep feedforward neural network, also known as a multilayer perceptron (MLP) is basically a network that maps inputs to outputs through a stack of layers with no feedback loops—so information flows strictly forward. They are the core building block of deep learning and powerful nonlinear function approximators.

$$
\begin{aligned}
&\textbf{Notation:} \quad \mathbf{x} \in \mathbb{R}^{d_0}, \quad \mathbf{h}^{(0)} = \mathbf{x}, \\
&\text{for layers } \ell = 1, \ldots, L: \\
&\qquad \mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \quad \text{with } W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}},\ \mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell}, \\
&\qquad \mathbf{h}^{(\ell)} = \phi^{(\ell)}\Big(\mathbf{z}^{(\ell)}\Big), \\
&\text{output: } \hat{\mathbf{y}} = \mathbf{h}^{(L)}.
\end{aligned}
$$

A multilayer perceptron consists of three core parts:

1. Input Layer: this layer contains no parameters and is where the raw data is received by the model and passed onto the first hidden layer.

2. Hidden Layers: these are the core of the network. Each layer performs a two-step transformation: first, a linear transformation with parameters that are learnt by the network during training, and then an activation function (also known as a non-lineratity) usually after the linear transformation.

3. Output Layer: applies a final linear map to the data being processed. Sometimes, it is followed by a task-specific activation, such as sigmoid activation for binary classification or the softmax function for multi-class classification.

Activation functions are the aforementioned functions applied on each hidden layer after the linear layers. They are needed because without them, stacked linear layers collapse to one linear transform, which means that non-linear relationships cannot be modelled. Some common choices of non-linearities are:

- Sigmoid: maps to $(0, 1)$; it is bounded but can cause vanishing gradients.
- tanh: it maps to $(-1, 1)$; it is centered around zero but can also vanish.
- ReLU: it outputs $\max(0, z)$; it mitigates vanishing and works well in practice.

The theorem of **universal approximation** states that a multilayer perceptron with at least one hidden layer and a suitable nonlinearity can approximate any continuous function on a compact set to arbitrary accuracy. It is important to note that existence doesn't equal learnability—the theorem does not say that we can efficiently find the weights by training. The theorem implies that a very wide single hidden layer can approximate anything; in practice, deeper networks often approximate complex functions more efficiently.

Backpropagation is the algorithm that computes how much each weight throughout the network contributed to the error. This help us then nudge each weight to reduce future error. It has three main stages:

1. Forward Pass: The input flows through the network, producing a prediction and a loss (error).

$$\textbf{Forward pass:} \quad \mathbf{h}^{(0)} = \mathbf{x}, \qquad \text{for } \ell = 1, \dots, L:$$
$$\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \qquad \mathbf{h}^{(\ell)} = \phi^{(\ell)}\Big(\mathbf{z}^{(\ell)}\Big).$$
$$\text{Output: } \hat{\mathbf{y}} = \mathbf{h}^{(L)}.$$

2. Backward Pass: Using the chain rule, gradients are passed backwards layer by layer, effectively computing the partial derivatives of the loss with respect to every parameter.

$$\textbf{Backward pass:} \quad \boldsymbol{\delta}^{(\ell)} := \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}.$$

   At the output layer:

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(L)}} \odot \phi^{(L)\prime}\Big(\mathbf{z}^{(L)}\Big).$$

   For hidden layers (recursively):

$$\boldsymbol{\delta}^{(\ell)} = \Big(W^{(\ell+1)}\Big)^{\top}\boldsymbol{\delta}^{(\ell+1)} \odot \phi^{(\ell)\prime}\Big(\mathbf{z}^{(\ell)}\Big), \qquad \ell = L-1, \dots, 1.$$

   Gradients with respect to parameters:

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}\Big(\mathbf{h}^{(\ell-1)}\Big)^{\top}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}.$$

   For a softmax output with categorical cross-entropy loss:

$$\hat{\mathbf{y}} = \text{softmax}\Big(\mathbf{z}^{(L)}\Big), \quad \mathcal{L} = -\sum_{k=1}^{K} y_k \log \hat{y}_k \quad \Rightarrow \quad \boldsymbol{\delta}^{(L)} = \hat{\mathbf{y}} - \mathbf{y}.$$

3. Update: An optimizer uses the calculated gradients to update the parameters.

$$\textbf{Update:} \quad W^{(\ell)} \leftarrow W^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(\ell)}}, \qquad b^{(\ell)} \leftarrow b^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(\ell)}}.$$

The key part of backpropagation is that, instead of recomputing derivatives from scratch for every weight, it reuses intermediate results through the computational graph. This makes the calculation of gradients tractable even when there are millions of parameters. When training deep networks, especially with functions like sigmoid or tanh, the gradients that flow back to early layers can become extremely small. Those early layers then stop learning because the weights barely change—a phenomenon known as vanishing gradients. This happens because in backpropagation, gradients get multiplied by each other's derivative. If those derivatives are often smaller than one, the product across many layers shrinks exponentially. Saturating activations like sigmoid or tanh have derivatives near zero, which attenuates this issue. Mitigations to this issue are using non-saturating activations like ReLU, normalization, or beter initialization.

Here a recipe for training a neural network:

1. **Choose architecture:** Decide the number of hidden layers, units per layer, and the activation function.
   - Start with 2–4 hidden layers.

- Common widths: $64, 128, 256$.
- Activation: ReLU or GELU are strong defaults.

2. **Initialize weights:** Use an initialization that matches your activation.
   - **He/Kaiming** for ReLU-family activations.
   - **Xavier/Glorot** for tanh or sigmoid networks.
3. **Pick loss & output (task-specific):**
   - **Regression:** Linear output; loss = MSE/MAE/Huber.
   - **Binary classification:** Sigmoid output; loss = binary cross-entropy.
   - **Multi-class classification:** Softmax output; loss = categorical cross-entropy.
4. **Optimizer:** Start with Adam and modest regularization.
   - Adam learning rate $\approx 10^{-3}$.
   - Add weight decay (L2) e.g. 10^{-4}.
5. **Stabilize training (as needed):**
   - Add Batch Normalization to stabilize activations and speed convergence.
   - Add Dropout if you observe overfitting.
   - Consider a learning-rate schedule (e.g., cosine decay, step decay, or warmup).
6. **Train with early stopping:**
   - Monitor a validation metric (e.g., loss, accuracy, F1).
   - Stop when the metric plateaus or degrades for several epochs (patience).
   - Keep the best-performing checkpoint.
7. **Tune hyperparameters:** Use efficient search strategies.
   - Random or Bayesian search over: learning rate, width/depth, dropout rate, weight decay, batch size.
   - Optionally use k-fold cross-validation when data is limited.
8. **Final evaluation:**
   - Retrain the model on train + validation with the best hyperparameters.
   - Evaluate *once* on the held-out test set to report final performance.

1.2 Regularisation in Deep Learning

Deep learning models are **high-capacity models**: they are flexible enough to represent highly complex functions. While this makes them powerful, it also creates a significant risk of **overfitting**—the model learning noise or pecularities in the training data rather than the true underlying patterns in it, rendering the model unable to properly generalise. Regularisation is what helps the model achieve good performance on unseen data. A model should be complex enough to capture meaningful structure but not so flexible that it memorises the training data; regularisation is what helps achieve this balance.

A common way to regularise a model is to add a penalty term to the loss function, something known as **penalty-based regularisation**. Let the original training loss be

$$J_0(\mathbf{w}),$$

where $\mathbf{w}$ denotes the model parameters. The regularized objective is

$$J(\mathbf{w}) = J_0(\mathbf{w}) + \lambda\,\Omega(\mathbf{w}),$$

where:

- $\Omega(\mathbf{w})$ is the regularization term,
- $\lambda > 0$ controls the strength of regularization. If it is too small, the regularisation effect is weak and overfitting may persist; however, if it is too large, the model might just straight up underfit. Thus, it is a hyperparameter that must be tuned.

Another common approach to regularisation is **L2 Regularisation**, also known as weight decay. L2 regularisation penalises the squared Euclidean norm of the weights:

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2 = \sum_{i=1}^{d} w_i^2.$$

Hence the objective becomes

$$J(\mathbf{w}) = J_0(\mathbf{w}) + \lambda \sum_{i=1}^{d} w_i^2.$$

The effect of weight decay is that weights shrink towards zero, without forcing them to be exactly zero. The gradient of the penalty term is then:

$$\nabla_{\mathbf{w}} \left(\lambda \|\mathbf{w}\|_2^2\right) = 2\lambda\mathbf{w}.$$

Which makes the gradient descent updates take the form of:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\left(\nabla J_0(\mathbf{w}_t) + 2\lambda\mathbf{w}_t\right),$$

where η is the learning rate. Equivalently, it can be expressed as:

$$\mathbf{w}_{t+1} = (1 - 2\eta\lambda)\mathbf{w}_t - \eta\nabla J_0(\mathbf{w}_t),$$

which shows explicitly that the weights are decayed toward zero.

Another approach is **L1 Regularisation**, which penalises the absolute values of the weights:

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_{i=1}^{d} |w_i|.$$

Thus,

$$J(\mathbf{w}) = J_0(\mathbf{w}) + \lambda \sum_{i=1}^{d} |w_i|.$$

The effect of L1 regularisation is sparsity; many weights can become exactly zero, which might be useful for feature selection or simpler models. Since $|w_i|$ is not differentiable at zero, a subgradient is used:

$$\frac{\partial}{\partial w_i}|w_i| = \begin{cases} +1, & w_i > 0, \\ -1, & w_i < 0, \\ \text{any value in } [-1, 1], & w_i = 0. \end{cases}$$

So, the update is approximately:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left(\nabla J_0(\mathbf{w}_t) + \lambda \operatorname{sign}(\mathbf{w}_t) \right),$$

where $\operatorname{sign}(\mathbf{w})$ is applied componentwise.

Regularisation also has a **Bayesian interpretation**. Suppose the likelihood of the data is $p(\mathcal{D} \mid \mathbf{w})$, and we place a prior $p(\mathbf{w})$ on the parameters. Then the maximum a posteriori (MAP) estimate is

$$\mathbf{w}_{\text{MAP}} = \arg\max_{\mathbf{w}} p(\mathbf{w} \mid \mathcal{D}).$$

Using Bayes' rule,

$$p(\mathbf{w} \mid \mathcal{D}) \propto p(\mathcal{D} \mid \mathbf{w})\, p(\mathbf{w}).$$

Taking negative logarithms gives

$$-\log p(\mathbf{w} \mid \mathcal{D}) = -\log p(\mathcal{D} \mid \mathbf{w}) - \log p(\mathbf{w}) + C,$$

where C is a constant independent of $\mathbf{w}$. Thus, minimizing the negative log-posterior is equivalent to minimizing

$$J(\mathbf{w}) = -\log p(\mathcal{D} \mid \mathbf{w}) - \log p(\mathbf{w}).$$

L2 can be seen as a Gaussian prior. If

$$p(\mathbf{w}) \propto \exp\left(-\frac{1}{2\sigma^2} \|\mathbf{w}\|_2^2 \right),$$

then

$$-\log p(\mathbf{w}) \propto \|\mathbf{w}\|_2^2,$$

which yields L2 regularization.

L1, on the other hand, can be expressed as a Laplace prior. If

$$p(\mathbf{w}) \propto \exp\left(-\frac{1}{b} \|\mathbf{w}\|_1 \right),$$

then

$$-\log p(\mathbf{w}) \propto \|\mathbf{w}\|_1,$$

which yields L1 regularization.

Instead of adding a penalty term, regularisation can also be formulated as a constrained optimisation problem. The L2 constraint is:

$$\min_{\mathbf{w}} J_0(\mathbf{w}) \quad \text{subject to} \quad \|\mathbf{w}\|_2^2 \leq k.$$

Geometrically, this restricts the solution to lie inside an L2 ball (a sphere in low dimensions). In the case of L1:

$$\min_{\mathbf{w}} J_0(\mathbf{w}) \quad \text{subject to} \quad \|\mathbf{w}\|_1 \leq k.$$

Which geometrically restricts the solution to an L1 ball, which appears as a diamond in two dimensions. A constrained problem can then be converted into an unconstrained one using a Lagrange multiplier:

$$\mathcal{L}(\mathbf{w}, \alpha) = J_0(\mathbf{w}) + \alpha\Big(\Omega(\mathbf{w}) - k\Big),$$

where $\alpha \geq 0$. For example, with an L2 constraint:

$$\mathcal{L}(\mathbf{w}, \alpha) = J_0(\mathbf{w}) + \alpha\left(\|\mathbf{w}\|_2^2 - k\right).$$

For hard constraints, one can use projected gradient descent:

$$\widetilde{\mathbf{w}}_{t+1} = \mathbf{w}_t - \eta \nabla J_0(\mathbf{w}_t),$$

followed by projection onto the feasible set:

$$\mathbf{w}_{t+1} = \Pi_{\mathcal{C}}(\widetilde{\mathbf{w}}_{t+1}),$$

where $\Pi_{\mathcal{C}}$ is the projection onto the constraint set $\mathcal{C}$. Thus the algorithm alternates:

$$\text{update} \rightarrow \text{project} \rightarrow \text{update} \rightarrow \text{project}.$$

Deep learning often operates in regimes where the number of parameters exceeds the number of training samples. Such problems are often **underconstrained**. Thus, there may be infinitely many parameter vectors $\mathbf{w}$ that fit the training data well. If

$$N_{\text{params}} > N_{\text{samples}},$$

then the optimisation problem may admit many minimisers:

$$\{\mathbf{w} : J_0(\mathbf{w}) \text{ is minimal}\}.$$

Regularisation helps choose among these solutions by preferring ones with desirable properties, such as small norm or sparsity:

$$\mathbf{w}^\star = \arg\min_{\mathbf{w}}\Big(J_0(\mathbf{w}) + \lambda\Omega(\mathbf{w})\Big).$$

Another important regularisation strategy is **data augmentation**. Deep models are data-hungry. If the available dataset is limited, the model may overfit. Data augmentation artificially expands the training set by applying label-preserving transformations. If T is a transformation applied to input x, then augmented data is

$$x' = T(x).$$

The learning objective can be interpreted as minimising expected loss over transformations:

$$\min_{\mathbf{w}} \mathbb{E}_{(x,y)\sim\mathcal{D}} \mathbb{E}_{T\sim\mathcal{T}}\Big[\ell(f_{\mathbf{w}}(T(x)), y)\Big].$$

This encourages invariance or robustness to such transformations. Some common transformations are rotation, scaling, cropping, flipping, translation, noise injection, brightness/contrast changes, among others. Data augmentation creates a more challenging training scenario and prevents the model from relying on superficial cues. This often improves robustness and generalisation. That being said, not all augmentations are appropriate for all datasets—poorly chosen ones can lead to unrealistic samples, label corruption, or unnecessary computational overhead. A more advanced augmentation method is **mixup**, where two examples are combined:

$$\widetilde{x} = \lambda x_i + (1-\lambda)x_j,$$

$$\widetilde{y} = \lambda y_i + (1-\lambda)y_j,$$

where $\lambda \in [0,1]$. This encourages smoother decision boundaries.

Noise is any unwanted perturbation or alteration in the data. It is almost always present in real-world data and can reduce accuracy, increase variance, and make predictions generally unstable. However, carefully injecting noise during training can improve robustness, since it encourages the model to learn more stable representations. A common method is to perturb the input:

$$\widetilde{x} = x + \varepsilon,$$

where

$$\varepsilon \sim \mathcal{N}(0, \sigma^2 I)$$

for Gaussian noise, or

$$\varepsilon_i \sim \text{Uniform}(-a, a)$$

for uniform noise. The training objective then becomes

$$\min_{\mathbf{w}} \mathbb{E}_{(x,y)\sim\mathcal{D}} \mathbb{E}_{\varepsilon}\Big[\ell(f_{\mathbf{w}}(x+\varepsilon), y)\Big].$$

Noise can also be applied to weights:

$$\widetilde{\mathbf{w}} = \mathbf{w} + \varepsilon_w,$$

or to hidden activations:

$$\widetilde{h} = h + \varepsilon_h.$$

Noise injection in general can be viewed as a regulariser because it makes training more difficult in a controlled way, thereby discouraging fragile solutions.

Multitask Learning refers to a single model being trained to solve several related tasks simultaneously. Suppose there are T tasks, with task-specific losses $\mathcal{L}_1, \ldots, \mathcal{L}_T$. A standard objective is

$$\mathcal{L}_{\text{MTL}}(\mathbf{w}) = \sum_{t=1}^{T} \alpha_t \mathcal{L}_t(\mathbf{w}),$$

where α_t determines the importance of task t. Typically, early layers are shared:

$$h = g_{\mathbf{w}_{\text{shared}}}(x),$$

and each task has its own head:

$$\widehat{y}^{(t)} = f^{(t)}_{\mathbf{w}_t}(h).$$

This process is regularising because it forces the model to perform multiple related tasks, which prevents it from over-specialising to one singular task. This is often done with, for instance, parsing + tagging + sentiment analysis, or object classification + localisation. It is important to ensure the tasks are sufficiently related; otherwise, one might get negative transfer in which learning one task harms another.

Lastly, **dropout** is another very popular regularisation method. During training, each hidden unit is randomly kept with probability p. If h is a hidden activation vector and m is a Bernoulli mask,

$$m_i \sim \text{Bernoulli}(p),$$

then the dropped-out activation is

$$\widetilde{h} = m \odot h,$$

where $\odot$ denotes elementwise multiplication. In inverted dropout, one uses

$$\widetilde{h} = \frac{m \odot h}{p},$$

so that the expected activation remains unchanged during training. The effect of this is discouraging the co-adaptation of neurons to improve generalisation.

1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are among the most influential innovations in deep learning. They are designed primarily for data with spatial or structured relationships, especially images, videos, audio signals, time series, or volumetric data such as CT or MRI scans. CNNs are especially powerful because they exploit **spatial hierarchies** and **local patterns**. In contrast to standard fully connected neural networks, CNNs are built to recognize local structures like edges, corners, textures, shapes, and increasingly complex objects in deeper layers.

At the heart of a CNN lies the **convolutional operator**, which applies a small filter (or kernel) across an input to produce a **feature map**. By stacking many such layers, the network learns hierarchical features layer by layer. CNNs power many modern AI systems like image classification, facial recognition, object detection, video analytics, and many, many others.

Traditional feed-forward fully-connected neural networks find it difficult to deal with images and other structured data. Firstly, because of the number of parameters; a fully-connected layer on a large image would require an enormous number of weights that would render computation extremely expensive. In addition to that, regular feed-forward networks lack a built-in spatial structure. This prevents such networks from exploit the strong relation between nearby pixels. Lastly, traditional feed-forward networks handle translation rather poorly—if an object moves in the image, a standard dense network does not naturally reuse what it learnt. Convolutional neural networks address these issues through local connectivity, parameter sharing, sparser architecture, and hierarchical feature extraction.

title=The Convolution Operator

A convolution is a mathematical operation that combines two functions to create a third that describes how the form of one of them is modified by the other. For two continuous functions f and g, the convolution is defined by

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau.$$

For discrete signals, the convolution then becomes a sum:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]\, g[n-m].$$

It is worth noting that in most deep learning libraries, the operation implemented in CNNs is actually **cross-correlation** rather than strict mathematical convolution, because the kernel is usually not flipped:

$$(f \star g)[n] = \sum_{m} f[n+m]\, g[m].$$

Nevertheless, by convention this is still called a *convolution* in deep learning.

For an image X and a kernel K, the discrete 2D convolution/cross-correlation is

$$Y(i,j) = \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} K(u,v)\, X(i+u, j+v),$$

where:

- X is the input image or feature map,
- K is the filter/kernel,
- Y is the output feature map.

The kernel slides across the input and computes a weighted sum of local regions. The output highlights the presence of patterns that match the kernel. Real CNNs usually operate on multi-channel inputs. For example, an RGB image has three input channels. If:

- $X \in \mathbb{R}^{H \times W \times C_{\text{in}}}$,

- $K \in \mathbb{R}^{k_h \times k_w \times C_{\text{in}}}$,

then one output channel is computed as:

$$Y(i,j) = \sum_{c=1}^{C_{\text{in}}} \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} K(u,v,c)\, X(i+u, j+v, c).$$

If we want C_{out} output channels, then we use C_{out} different kernels, each producing one feature map. Thus the full output is

$$Y \in \mathbb{R}^{H_{\text{out}} \times W_{\text{out}} \times C_{\text{out}}}.$$

The **stride** of a filter determines how far the kernel moves at each step. A stride of 1, for instance, means the kernel will move one pixel at a time. A stride of 2 would then make the kernel skip every other location. Larger stride reduces spatial resolution. **Padding** adds extra values, usually zeroes, around the input to help control the output size and preserve boundary information. For one spatial dimension, if:

- input size n,
- kernel size k,
- padding p,
- stride s,
- dilation d,

then the output size is

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - d(k-1) - 1}{s} + 1 \right\rfloor.$$

For 2D inputs, the formula must be applied separately to height and width:

$$H_{\text{out}} = \left\lfloor \frac{H + 2p_h - d_h(k_h - 1) - 1}{s_h} + 1 \right\rfloor,$$

$$W_{\text{out}} = \left\lfloor \frac{W + 2p_w - d_w(k_w - 1) - 1}{s_w} + 1 \right\rfloor.$$

A **dilated convolution** is a type of convolution that inserts gaps between kernel elements to enlarge the receptive field without increasing the number of parameters. If the kernel size is k and dilation is d, then the effective kernel size is

$$k_{\text{eff}} = k + (k-1)(d-1).$$

This allows the model to capture wider context efficiently.

Convolutions work well because of several key features. The first one is that of parameter sharing: the same kernel is reused across all spatial locations, drastically reducing the number of parameters. For a fully connected layer from an input of size HWC_{in} to C_{out} outputs, the number of weights is

$$(HWC_{\text{in}})\, C_{\text{out}}.$$

For a convolutional layer with kernel size $k_h \times k_w$, the number of weights is

$$k_h k_w C_{\text{in}} C_{\text{out}}.$$

This is often vastly smaller. Another important feature of CNNs is sparse connectivity: each output value depends only on a local neighbourhood, not on the entire input. This reflects the fact that local structure matters a lot in structured data like images. In addition to this sparse connectivity, CNNs also have translation equivariance, meaning that if the input shifts, the feature map will shift accordingly:

$$T_\Delta(X) * K = T_\Delta(X * K),$$

Lastly, there is hierarchical feature learning. CNNs learn features in stages:

- early layers: edges, gradients, simple textures,
- middle layers: motifs, parts, repeated structures,
- deeper layers: object parts and semantic concepts.

This mirrors how humans might recognize an object:

$$\text{edges} \to \text{shapes} \to \text{patterns} \to \text{objects}.$$

A convolutional layer applies trainable kernels and then usually applies a nonlinearity (e.g. ReLU). If X is the input and K_m is the m-th kernel, then the m-th feature map is

$$Z_m = X * K_m + b_m,$$

followed by activation

$$A_m = \phi(Z_m),$$

where ϕ may be ReLU:

$$\phi(z) = \max(0, z).$$

The filters are learned by gradient descent through backpropagation. Different filters may learn to detect corners, horizontal/vertical edges, textures, colour transitions, repeated patterns, or other specific patterns in images.

An important technique when it comes to CNNs is **pooling**, a downsampling operation that reduces the spatial size of feature maps while retaining important information. It reduces dimensionality, which leads to less overfitting and thus higher robustness to small shifts and distortions. A common type of pooling is **max pooling**, in which if a strong feature appears anywhere in the region, it is preserved. For a local region R, max pooling outputs

$$y = \max_{x \in R} x.$$

For a 2×2 pooling region with stride 2, each 2×2 block is replaced by its maximum.
Average pooling, on the other hand, outputs the mean of the region, producing a smoother representation than max pooling:

$$y = \frac{1}{|R|} \sum_{x \in R} x.$$

Pooling obeys a size formula similar to convolution. For one spatial dimension,

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1.$$

Typically pooling uses:

- kernel 2×2,
- stride 2,
- no padding.

Pooling operations do not learn parameters; they are fixed. Since they discard information, proper pooling is a balance between preserving detail and achieving abstraction and efficiency.

In Bayesian terms, a **prior** encodes assumptions about what kinds of functions are plausible before seeing data. The architectures of CNNs encode such strong assumptions about images. The two key ones are:

1. Stationarity: The same kind of local feature, such as an edge, may appear anywhere in the input.
2. Locality: Nearby pixels are more strongly related than distant ones, so local neighbourhoods can be processed together.

The phrase *infinitely strong prior* refers to the idea that the architecture severely restricts the class of functions the network can represent. Thus, rather than learning any arbitrary mapping, a CNN assumes that the function must be compatible with

- local receptive fields,
- shared filters,
- repeated spatial structure.

This assumption can prove benefitial insofar that it narrows the hypothesis space and also reduces overfitting. However, in cases where the data does not satisfy these assumptions (e.g. tabular data), the prior might be too restrictive and negatively impact model performance. Pooling itself also imposes a strong architectural assumption—higher layers should progressively summarise and abstract lower-level features.

The standard convolutional layer has several variants:

- Strided Convolution: A convolution with stride $s > 1$ moves the filter by more than one step in order to reduce output resolution and computational cost. It can replace pooling while still performing learnt feature extraction.
- Dilated Convolution: It inserts gaps between kernel elements to expand the receptive field without increasing parameter count. If dilation is d, then the operation becomes

$$Y(i,j) = \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} K(u,v)\, X(i + d\,u,\ j + d\,v).$$

- Transposed Convolution: It increases spatial resolution, thus it is commonly used for upsampling. It can often be seen in the architecture of autoencoders, generative models, and image segmentation architectures. For one spatial dimension, a common output-size formula is

$$n_{\text{out}} = (n-1)s - 2p + k + \text{output\_padding}.$$

- Depthwise Convolution: This type of convolution convolves each input channel separately with its own filter—an approach much cheaper than a full convolution. If there are C_{in} channels, the parameter count is

$$k_h k_w C_{\text{in}}.$$

- Pointwise Convolution: It is a 1×1 convolution used to mix channels:

$$Y(i,j,c_{\text{out}}) = \sum_{c=1}^{C_{\text{in}}} W(c,c_{\text{out}})\, X(i,j,c).$$

  Its parameter count is

$$C_{\text{in}} C_{\text{out}}.$$

- Depthwise Separable Convolution: A combination of a depthwise convolution followed by a pointwise 1×1 convolution. Total parameters:

$$k_h k_w C_{\text{in}} + C_{\text{in}} C_{\text{out}}.$$

  Compare this with standard convolution:

$$k_h k_w C_{\text{in}} C_{\text{out}}.$$

  So it can yield significant efficiency gains over a regular convolution.

-

Besides image classification, CNNs can also produce structured outputs—outputs whose elements have spatial/sequential relationships. For example, it can produce segmentation maps, generated images, or image-to-image translations.

When it comes to segmentation, the model is required to assign a label to every pixel in an input image. If there are C classes and the model outputs logits $z_{i,j,c}$, then the per-pixel softmax probability is

$$p_{i,j,c} = \frac{\exp(z_{i,j,c})}{\sum_{c'=1}^{C} \exp(z_{i,j,c'})}.$$

Then, if $y_{i,j}$ is the true class at pixel (i,j), the segmentation loss can be

$$\mathcal{L}_{\text{CE}} = -\sum_{i,j} \sum_{c=1}^{C} \mathbf{1}\{y_{i,j} = c\} \, \log p_{i,j,c}.$$

CNNs can also be used to generate sequences from images, such as for image captioning or reading text from an image. For such cases, the CNN takes the role of an encoder,

producing a representation h, which is then passed to a sequence model. The sequence model predicts tokens $y_1, \ldots, y_T$ with conditional probability

$$p(y_1, \ldots, y_T \mid x) = \prod_{t=1}^{T} p(y_t \mid y_{<t}, h).$$

Convolutional layers can be computationally expensive, especially for large inputs or many channels; efficient implementations are essential. Suppose:

- input size $H \times W \times C_{\text{in}}$,
- kernel size $k_h \times k_w$,
- output channels C_{out}.

Then the rough operation count for a brute-force convolution is

$$\mathcal{O}\Big(HW\, C_{\text{in}}\, C_{\text{out}}\, k_h k_w\Big).$$

For an $n \times n$ input and an $n \times n$ kernel in a simplified setting, the cost is often described as at least quadratic or worse in n, depending on the exact assumptions.
Convolution can be accelerated using the convolution theorem:

$$\mathcal{F}(f * g) = \mathcal{F}(f)\, \mathcal{F}(g),$$

where $\mathcal{F}$ denotes the Fourier transform. So, one can compute:

1. FFT of the input,
2. FFT of the kernel,
3. elementwise multiplication in frequency space,
4. inverse FFT.

which often reduces the cost for large kernels down to roughly

$$\mathcal{O}(n^2 \log n)$$

for $n \times n$ data. Another benefit of FFT-based convolutions is that they are highly parallelisable.

The **im2col** trick reshapes sliding input patches into columns so that convolution becomes a matrix multiplication. If:

- X_{col} contains vectorized local patches,
- W_{row} contains vectorized kernels,

then convolution can be written as

$$Y_{\text{col}} = W_{\text{row}} X_{\text{col}}.$$

The reason why this approach can be efficient is that modern hardware is optimised for matrix multiplication.

2 Advanced Neural Network Techniques

Many real-world data sources, such as text, are inherently sequential. In such settings, the order of observations and the progression through time are essential. A traditional feed-forward neural network processes each input independently and does not have a built-in mechanism for remembering previous inputs. As a result, it does not naturally model temporal dependence. A **sequence model**, on the other hand, is a model designed to capture patterns in temporal or ordered data, where the position of each element in the sequence matters. These models are used for tasks like time-series forecasting, machine translation, and speech recognition, amongst many others.
Recurrent neural networks (RNNs) are neural networks designed to process sequences by maintaining a hidden state that evolves over time. Their defining feature is **feedback connections**, which allow for information from previous timesteps to influence the current computation, giving the network a form of memory. Let:

- x_t be the input at time t,
- h_t be the hidden state at time t,
- y_t be the output at time t.

A standard RNN then updates according to

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h),$$

$$y_t = W_{hy}h_t + b_y,$$

or, if using an output activation g,

$$\hat{y}_t = g(W_{hy}h_t + b_y).$$

Here:

- W_{xh} maps input to hidden state,
- W_{hh} is the recurrent weight matrix,
- W_{hy} maps hidden state to output,
- ϕ is typically tanh or another activation.

Notice that the hidden state h_t summarizes the current input x_t together with information remembered from the past through h_{t-1}.
Although an RNN is defined recursively, it can be understood by explicitly writing out the computation at each time step.For example, for three time steps:

$$h_1 = \phi(W_{xh}x_1 + W_{hh}h_0 + b_h),$$

$$h_2 = \phi(W_{xh}x_2 + W_{hh}h_1 + b_h),$$

$$h_3 = \phi(W_{xh}x_3 + W_{hh}h_2 + b_h).$$

This is called **unfolding** or **unrolling** the RNN through time. Unfolding makes the network look like a deep feed-forward network whose layers correspond to time steps.

This is important because it allows us to visualise the temporal computation and apply gradient-based optimisation, as well as perform backpropagation through time. A crucial property of RNNs is that the same parameters are reused at every time step:

$$W_{xh}^{(1)} = W_{xh}^{(2)} = \cdots = W_{xh},$$

$$W_{hh}^{(1)} = W_{hh}^{(2)} = \cdots = W_{hh},$$

$$W_{hy}^{(1)} = W_{hy}^{(2)} = \cdots = W_{hy}.$$

This parameter sharing makes the model efficient and able to generalize across sequences of varying length.

RNNs are trained via **backpropagation through time**. Suppose a sequence has length T, and let ℓ_t denote the loss at time t. The total loss is typically

$$\mathcal{L} = \sum_{t=1}^{T} \ell_t.$$

For example, in sequence classification or prediction:

$$\ell_t = \ell(\hat{y}_t, y_t).$$

Because each hidden state depends on previous hidden states, gradients must be propagated backward through all earlier time steps. Using the chain rule, the gradient with respect to recurrent weights involves products like

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}.$$

For a vanilla RNN with activation ϕ,

$$\frac{\partial h_j}{\partial h_{j-1}} = \operatorname{diag}\Big(\phi'(a_j)\Big) W_{hh},$$

where

$$a_j = W_{xh} x_j + W_{hh} h_{j-1} + b_h.$$

Thus, the gradient can involve repeated multiplication by W_{hh} and activation derivatives, which is the source of vanishing or exploding gradients. Thus, training RNNs is rendered difficult by challenges like vanishing/exploding gradients, long-sequence memory cost, slow convergence, and high computational expense for long sequences. A common workaround for some of these issues is truncated backpropagation through time, where gradients are propagated only for a limited number of timesteps K rather than the entire history. This method clearly results in less memory and computation to the benefit of faster training.

A long-term dependency occurs when the correct output at the current time step depends on information that appeared far back in the sequence. For instance, the subject of a very long sentence is the one determining the form of the verb that appears later. Regular RNNs struggle to capture this because gradients are products of many Jacobian

terms, so they tend to shrink/grow exponentially with sequence length. If repeated factors have norm less than 1, then there might be vanishing gradients:

$$\left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \to 0 \quad \text{as } t-k \text{ increases.}$$

This makes earlier time steps have almost no influence on learning. Vanishing gradients make the network **short-sighted**: it focuses mainly on recent inputs and forgets distant context. On the other side of the coin, if repeated factors have a norm greater than 1, then the gradients will explode, causing unstable updates and divergence:

$$\left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \to \infty.$$

Exploding gradients make optimization unstable. Together, these issues limit the power of vanilla RNNs for modeling long sequences.
A standard RNN processes a sequence in one direction, typically from past to future:

$$x_1 \to x_2 \to \cdots \to x_T.$$

However, some tasks, such as speech recognition and named entity recognition, require both past and future context to interpret the current token or signal. This is where bidirectional RNNs come into play. A bidirectional RNN uses two recurrent layers:

- a forward RNN,
- a backward RNN.

Forward hidden states:

$$\overrightarrow{h_t} = \phi(W_{x\to}x_t + W_{\to\to}\overrightarrow{h_{t-1}} + b_{\to}).$$

Backward hidden states:

$$\overleftarrow{h_t} = \phi(W_{x\leftarrow}x_t + W_{\leftarrow\leftarrow}\overleftarrow{h_{t+1}} + b_{\leftarrow}).$$

The combined representation at time t is often

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}],$$

where $[\cdot;\cdot]$ denotes concatenation. These networks are able to use both past and future context, improving accuracy on offline sequence tasks and giving richer sentence representations. The issue with this model is that it requires the full sentnece in advance, which means it is not ideal for online or real-time settings. Furthermore, it doubles recurrent computation and memory.

In many applications, the input is a sequence and the output is another sequence, possibly of different length. For example, in question answering the answer will generally be a sequence of different composition and length to the question. This is what motivates the **encoder–decoder** or **sequence-to-sequence** architecture. Firstly, the encoder

processes the input sequence $x_1, \ldots, x_T$ and compresses it into a context representation c. A simple formulation is

$$h_t^{\text{enc}} = \phi(W_{xh}^{\text{enc}} x_t + W_{hh}^{\text{enc}} h_{t-1}^{\text{enc}} + b_h^{\text{enc}}),$$

with context vector

$$c = h_T^{\text{enc}}.$$

Then, the decoder generates the output sequence step by step, conditioned on the context vector:

$$h_t^{\text{dec}} = \phi(W_{yh}^{\text{dec}} y_{t-1} + W_{hh}^{\text{dec}} h_{t-1}^{\text{dec}} + W_{ch} c + b_h^{\text{dec}}),$$
$$\hat{y}_t = g(W_{ho}^{\text{dec}} h_t^{\text{dec}} + b_o).$$

Basically, the encoder summarizes the input while the decoder expands that summary into the output sequence. A weakness of the basic encoder-decoder is that compressing a long sequence into one fixed vector may lose information. This is the motivation for a mechanism named **attention**, which allows the decoder to focus on (*pay attention to*) different encoder states at each output step. At decoder time t, attention weights may be defined as

$$\alpha_{t,s} = \frac{\exp(e_{t,s})}{\sum_{r=1}^{T} \exp(e_{t,r})},$$

where $e_{t,s}$ scores how relevant encoder state h_s^{enc} is to decoder step t. The context then becomes

$$c_t = \sum_{s=1}^{T} \alpha_{t,s} h_s^{\text{enc}}.$$

Then the decoder uses c_t instead of a single fixed c. Attention helps handle variable-length inputs and outputs and has proven particularly powerful for tasks like translation, summarisation, and Q&A. Despite this, basic fixed-context models lose information on long sequences when using the mechanism, sequential decoding can be slow, and attention increases complexity.

A basic RNN has limited representational power. As with feed-forward networks, deeper architectures can learn richer hierarchical representations. More layers may help capture low-level temporal patterns in early layers, higher-level abstractions in later layers, and more complex sequential structures.
In a deep RNN, each time step has multiple recurrent layers. For layer $\ell = 1, \ldots, L$,

$$h_t^{(\ell)} = \phi\Big(W_x^{(\ell)} h_t^{(\ell-1)} + W_h^{(\ell)} h_{t-1}^{(\ell)} + b^{(\ell)}\Big),$$

where $h_t^{(0)} = x_t$. Issues that present themselves with deep RNNs, despite their richer sequence representations, are higher computational cost, increased risk of vanishing/exploding gradients, and higher sensitivity to initialisation and optimisation strategy.

Recurrent neural networks model sequential structure, where order matters. Recursive neural networks, by contrast, are designed for **hierarchical tree-structured data**. Sample uses of recursive neural networks are syntactic parse trees in natural language and hierarchical semantic structures. The core idea is that each node in the tree is represented by a vector, and child nodes are recursively combined to form the parent node

representation. If c_1 and c_2 are child vectors, then the parent vector p can be computed as

$$p = f\Big(W[c_1; c_2] + b\Big),$$

where:

- $[c_1; c_2]$ is the concatenated child representation,
- W is a learned weight matrix,
- b is a bias,
- f is an activation function such as tanh.

Sometimes one writes

$$p = f(W_1 c_1 + W_2 c_2 + b).$$

title=Difference between recurrent and recursive

- **Recurrent neural networks:** designed for sequences; order in time matters.
- **Recursive neural networks:** designed for hierarchical structures; tree topology matters.

2.1 Difference Between Recurrent and Recursive

Recursive models capture hierarchical structure well but are more computationally demanding and more complex to train than linear-sequence RNNs.

Real-world temporal data also often contains dependencies at multiple time scales. For instance, finance has daily variation and longer-term trends or speech which contains both short phonetic and longer prosodic structure. A standard RNN uses a single recurrence timescale, which may prove insufficient. In these instances is where one can apply a leaky integration. A leaky recurrent update retains part of the previous state and blends it with a candidate update:

$$h_t = \alpha h_{t-1} + (1 - \alpha)\, \widetilde{h}_t,$$

where

$$\widetilde{h}_t = \phi(W_{xh} x_t + W_{hh} h_{t-1} + b).$$

Here $\alpha \in [0, 1]$ controls memory retention. If it is small, the model will have fast adaptation but shorter memory (and the opposite is also true). Using different α values across units allows the network to represent different temporal scales. This is useful because then the network can allocate some units to short-term changes and others to long-term dependencies, which goes on to improve the modelling of multiscale temporal structure. Vanilla RNNs struggle with long-term dependencies because they repeatedly overwrite the hidden state and suffer from unstable gradient flow. Thus, **gated architectures** were introduced to control information flow more explicitly. The main idea is to use learned gates to determine what to remember, what to forget, and what to expose to the output. Another solution devised to address the gradient issue is **Long Short-Term Memory (LSTM)** architecture; LSTMs introduce a separate **cell state** c_t and a set of

gates that regulate information flow. In LSTMs, the forget gate decides how much of the previous cell state to retain:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f).$$

Here:

- $f_t \in [0,1]^d$,
- values near 0 mean "forget",
- values near 1 mean "keep".

Then, the input gate determines what new information to store:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i).$$

Candidate cell content is

$$\widetilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c).$$

The cell state is updated by combining the forget decision with the new input:

$$c_t = f_t \odot c_{t-1} + i_t \odot \widetilde{c}_t,$$

where $\odot$ denotes elementwise multiplication. This additive update helps gradients flow more stably through time. Lastly, the output gate determines the next hidden state:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o).$$

Then

$$h_t = o_t \odot \tanh(c_t).$$

The strength in the LSTM gating structure is that it allows the network to store information for long periods while selectively forgetting irrelevant information. This makes LSTMs much more effective than vanilla RNNs on long sequences.

The **Gated Recurrent Unit** (GRU) is a simpler gated architecture introduced after the LSTM. It has fewer parameters and often performs competitively. The update gate is

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z).$$

The reset gate is

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r).$$

The candidate hidden state is

$$\widetilde{h}_t = \tanh\Big(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h\Big).$$

The final hidden state is

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h}_t.$$

In GRUs, as you can see, the update gate decides ow much of the old state to keep, and the reset one determines how much past information to ignore when computing the candidate—merging the roles of the LSTM's input and forget behaviour into a simpler form. GRUs have fewer parameters than LSTMs, are often faster to train, and frequently are comparable in efectiveness.

Some key optimisation strategies for RNNs are:

- Gradient Clipping: Used to deal with exploding gradients, if $g = \nabla_\theta \mathcal{L}$, then norm clipping replaces it by

$$g_{\text{clip}} = g \cdot \min\left(1, \frac{\tau}{\|g\|_2}\right),$$

where τ is a threshold. This prevents extremely large updates and stabilises training.

- Activation choices: the choice of an activation function can affect gradient flow. For instance, tanh and sigmoid may saturate and contribute to vanishing gradients, ReLU-like units can sometimes help gradient flow, and gated architectures are usually more effective than merely changing activations.

- Adaptive optimisers: Adaptive methods like RMSProp and Adam are often helpful. In the case of RMSProp, it is as follows:

$$s_t = \rho s_{t-1} + (1-\rho) g_t^2,$$

$$\theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{s_t} + \epsilon}.$$

For Adam, it is as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t,$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t},$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

- Weight initialisation: Initialisation strongly affects RNN training dynamics. Orthogonal initialisation is a strategy that initiates weights so that

$$W_{hh}^\top W_{hh} = I.$$

This helps because orthogonal matrices preserve vector norms better than arbitrary matrices, which can help to stabilise gradients over time.

2.2 Autoencoders

Autoencoders are an important family of unsupervised or self-supervised neural network models. Their central goal is to learn useful internal representations of data by trying to reconstruct the input from a compressed or constrained latent representation.

$$x \longrightarrow z \longrightarrow \hat{x},$$

where:

- x is the input,
- z is a latent or hidden representation,
- $\hat{x}$ is the reconstruction of the input.

The main idea is that the encoder compresses the input into a lower-dimensional or otherwise constrained representation, and the decoder reconstructs the input from that representation. Autoencoders enable data compression, denoising, dimensionality reduction, feature learning, generative modelling, anomaly detection, and other similar tasks. An autoencoder consists of two main components: firstly, encoder maps the input x into a latent representation z:

$$z = f_\theta(x),$$

where f_θ is a neural network with parameters θ. A common affine-nonlinear encoder layer is

$$z = \phi(W_e x + b_e),$$

where:

- W_e is the encoder weight matrix,
- b_e is the encoder bias,
- ϕ is an activation function.

The second compoent is the decoder; it maps the latent code z back into the reconstruction $\hat{x}$:

$$\hat{x} = g_\phi(z),$$

or explicitly,

$$\hat{x} = \psi(W_d z + b_d),$$

where:

- W_d is the decoder weight matrix,
- b_d is the decoder bias,
- ψ is an output activation.

The full autoencoder is then the composition

$$\hat{x} = g_\phi(f_\theta(x)).$$

The training objective is to make $\hat{x}$ as close as possible to x. Thus, the core loss of an autoencoder measures reconstruction quality. For continuous data, the most common loss is the mean squared error (MSE):

$$\mathcal{L}_{\text{rec}}(x, \hat{x}) = \|x - \hat{x}\|_2^2.$$

Across a dataset $\{x^{(i)}\}_{i=1}^N$, one typically minimizes

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \|x^{(i)} - \hat{x}^{(i)}\|_2^2.$$

For binary or normalized image data, one may instead use binary cross-entropy:

$$\mathcal{L}_{\text{BCE}}(x, \hat{x}) = -\sum_{j=1}^{d} \left[x_j \log \hat{x}_j + (1 - x_j) \log(1 - \hat{x}_j) \right].$$

The optimization objective is always to reduce the difference between x and $\hat{x}$, forcing the latent representation z to capture meaningful structure. Autoencoders work because they are asked to reproduce their input, but they're often constrained in some way so that they cannot simply memorise the identity function trivially. These constraints might come from sparsity, a smoothness penalty, a low-dimensional bottleneck, probabilistic latent variables, or other factors that might force the model to learn a useful structure in the data.

An **undercomplete autoencoder** is a type of autoencoder that has a latent space dimension smaller than the input dimension:

$$\dim(z) < \dim(x).$$

Because the latent space is smaller than the input space, the encoder is forced to retain only the most important information needed for reconstruction. This encourages dimensionality reduction, compression, and extraction of salient features. Another type of autoencoder is the **overcomplete autoencoder**. It has a latent space dimension greater than or equal to the input dimension:

$$\dim(z) \geq \dim(x).$$

Note that, without additional constraints, an overcomplete autoencoder may learn a trivial identity mapping:

$$\hat{x} \approx x \quad \text{without learning meaningful structure.}$$

Regularisation is usually needed to prevent this, such as sparsity penalties, denoising objectives, contractive penalties, or stochastic bottlenecks.

2.2.1 Regularised Autoencoders

A regularised autoencoder adds constraints or penalties to force the model to learn more useful and robust features. The generic objective is

$$\mathcal{L} = \mathcal{L}_{\text{rec}} + \lambda \mathcal{R},$$

where:

- $\mathcal{L}_{\text{rec}}$ is the reconstruction loss,
- $\mathcal{R}$ is a regularisation term,
- λ controls the strength of the regularisation.

Important regularised autoencoders include:

- sparse autoencoders,

- denoising autoencoders,
- contractive autoencoders,
- variational autoencoders.

A sparse autoencoder encourages only a small number of hidden units to be active for any given input. Even if the latent dimension is not small, sparsity forces the representation to be selective and informative. A simple sparsity penalty uses the L1 norm of the hidden activations:

$$\mathcal{R}_{\text{sparse}} = \|z\|_1 = \sum_{j=1}^{m} |z_j|.$$

The total loss becomes

$$\mathcal{L} = \mathcal{L}_{\text{rec}} + \lambda \|z\|_1.$$

A more classical sparse autoencoder penalty matches the average activation of hidden unit j to a small target ρ. Let

$$\hat{\rho}_j = \frac{1}{N} \sum_{i=1}^{N} z_j^{(i)}.$$

Then the sparsity penalty is

$$\mathcal{R}_{\text{KL}} = \sum_{j=1}^{m} \text{KL}(\rho \,\|\, \hat{\rho}_j),$$

where

$$\text{KL}(\rho \,\|\, \hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}.$$

The full objective is

$$\mathcal{L} = \mathcal{L}_{\text{rec}} + \beta \sum_{j=1}^{m} \text{KL}(\rho \,\|\, \hat{\rho}_j).$$

Sparse autoencoders often learn more robust features, more interpretable latent factors, and better representation efficiency, which makes them useful.

A denoising autoencoder is trained to reconstruct the clean input x from a corrupted version $\tilde{x}$. By learning to remove noise, the model is encouraged to capture the underlying structure of the data rather than superficial details. Firstly, a noisy version of the input is generated via a corruption process

$$\tilde{x} \sim q_D(\tilde{x} \mid x).$$

Common corruption methods include Gaussian noise, masking noise, and so-called *salt-and-pepper* noise. The encoder proceeds to take the noisy input:

$$z = f_\theta(\tilde{x}),$$

and the decoder reconstructs the clean target:

$$\hat{x} = g_\phi(z).$$

The loss is

$$\mathcal{L}_{\text{DAE}} = \mathbb{E}_{\tilde{x} \sim q_D(\tilde{x}|x)} \left[\|x - \hat{x}\|_2^2 \right].$$

Denoising autoencoders focus on essential patterns and learn robustness to perturbations, which help them resist overfitting better than naive reconstruction models.
A **contractive autoencoder** encourages the encoder to be insensitive to small perturbations of the input. If tiny input changes produce huge latent changes, the representation is unstable. Contractive autoencoders penalise this sensitivity. Let the encoder be

$$z = f_\theta(x).$$

Its Jacobian is

$$J_f(x) = \frac{\partial f_\theta(x)}{\partial x}.$$

The contractive penalty uses the Frobenius norm of the Jacobian:

$$\mathcal{R}_{\text{contractive}} = \|J_f(x)\|_F^2 .$$

The total contractive autoencoder loss is

$$\mathcal{L}_{\text{CAE}} = \mathcal{L}_{\text{rec}} + \lambda \|J_f(x)\|_F^2 .$$

Large Jacobian norms indicate high sensitivity of the latent code to small input changes. Penalising the Jacobian encourages local stability, robustness to small perturbations, a smoother latent structure, and contraction around the data manifold. Contractive autoencoders can provde improved generalisation, less sensitivity to noise, more stable latent features, and a better manifold learning behaviour; however, Jacobian computation can be expensive and the regularisation coefficient must be tuned carefully, since too much contraction may reduce the model's representational power.
A **variational autoencoder** (VAE) combines ideas from deep learning and Bayesian latent-variable modeling. Unlike a deterministic autoencoder, a VAE does not encode an input as a single point z. Instead, it encodes the input as a **distribution** over latent variables. A VAE assumes a generative model

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

where the prior over latent variables is usually

$$p(z) = \mathcal{N}(0, I).$$

The encoder approximates the posterior $p_\theta(z \mid x)$ with a variational distribution

$$q_\phi(z \mid x).$$

A common choice is

$$q_\phi(z \mid x) = \mathcal{N}\Big(z; \mu_\phi(x), \operatorname{diag}(\sigma_\phi^2(x))\Big).$$

Thus the encoder outputs:

- a mean vector $\mu_\phi(x)$,
- a variance or log-variance vector $\sigma_\phi^2(x)$.

The VAE is trained by maximising the evidence lower bound (ELBO):

$$\mathcal{L}_{\text{VAE}}(x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] - \text{KL}\Big(q_\phi(z \mid x) \,\|\, p(z)\Big).$$

Equivalently, one often minimises the negative ELBO:

$$\mathcal{J}_{\text{VAE}}(x) = -\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] + \text{KL}\Big(q_\phi(z \mid x) \,\|\, p(z)\Big).$$

Here, the first term is a reconstruction term while the second one regularises the latent distribution towards the prior. To enable gradient-based optimization, one writes

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I).$$

This separates randomness from the parameters and allows backpropagation. VAEs are important because they are generative models that can generate new samples, provide stochastic latent representations, and are more robust than purely deterministic encoders.

2.2.2 Stochastic Encoders and Decoders

A stochastic autoencoder introduces randomness into the encoding and/or decoding process. In a deterministic autoencoder, each input always produces the same latent code and output. In a stochastic autoencoder, on the other hand, the encoder may output a distribution rather than a point while the decoder may degenerate random samples conditioned on the latent code.
A stochastic encoder defines a conditional distribution

$$q_\phi(z \mid x)$$

instead of a deterministic function $z = f_\theta(x)$. Then one samples

$$z \sim q_\phi(z \mid x).$$

This acts as a regulariser and encourages the model to learn more robust and meaningful features. A stochastic decoder defines

$$p_\theta(x \mid z),$$

so reconstruction is probabilistic rather than deterministic. This is less common than stochastic encoding in basic autoencoders but central in generative models such as VAEs.

Stochasticity can improve robustness, reduce overfitting, encourage the model to focus on stable features, and provide generative capabilities. Some examples of stochasticity in practice include dropout, noise injection, latent sampling, and Monte Carlo sampling in encoder/decoder.

2.2.3 The Manifold Hypothesis

A major conceptual motivation for autoencoders is the **manifold hypothesis**. A manifold is a continuous, locally Euclidean, non-self-intersecting surface embedded in a higher-dimensional space. High-dimensional real-world data often lies near a much lower-dimensional manifold. If $x \in \mathbb{R}^d$ but the intrinsic dimension is much smaller, then the data effectively lives on or near some manifold $\mathcal{M} \subset \mathbb{R}^d$. If natural data lies near a low-dimensional manifold, that implies that compression, denoising, and robust representations are all possible. Good latent spaces should reflect the intrinsic manifold structure. Autoencoders can then be interpreted as learning:

- an encoder that maps data points into a lower-dimensional latent representation,
- a decoder that maps points on or near the latent manifold back to the original space.

Thus:

$$x \xrightarrow{\text{encoder}} z \quad \text{and} \quad z \xrightarrow{\text{decoder}} \hat{x}.$$

The encoder discovers a compressed manifold-like representation, and the decoder reconstructs from it.
At each point on a smooth manifold, one can define a tangent space (or tangent plane in low dimensions) that locally approximates the manifold. Thus, a good representation should be sensitive to meaningful directions along the manifold while also insensitive to small perturbations orthogonal to the manifold.
Contractive autoencoders actually encourage local invariance by penalizing sensitivity away from the manifold. The Jacobian

$$J_f(x) = \frac{\partial f_\theta(x)}{\partial x}$$

captures how the encoding changes with the input. Penalizing its norm encourages the encoder to contract around the data manifold. Geometrically, this means that the encoder becomes less sensitive across irrelevant directions while preserving meaningful local structure.

2.3 Generative Models

A generative model is a model that learns the underlying probability distribution of data and uses that knowledge to generate new samples. A discriminative model learns something like

$$p(y \mid x),$$

which is useful for prediction or classification. A generative model instead learns something like

$$p(x) \quad \text{or} \quad p(x, z),$$

where x is observed data and z may be latent structure. After learning the data distribution, the model can generate new samples

$$x_{\text{new}} \sim p_{\text{model}}(x)$$

that resemble the training data distribution. Generative models:

- create new data,
- learn the underlying distribution of the training data,
- are often trained using unlabelled data,
- can be used to understand data structure and variation,
- are useful in both creative and scientific applications.

Generative models are useful for:

- **data augmentation:** generating synthetic training samples,
- **anomaly detection:** identifying data far from the learned distribution,
- **creative generation:** images, music, text, speech, video,
- **data understanding:** learning latent structure of complex data,
- **semi-supervised learning:** leveraging unlabelled data,
- **privacy-aware synthetic data:** generating substitutes for sensitive data.

Generative models can sometimes be used to produce synthetic datasets that imitate real data distributions while reducing direct exposure of sensitive records. This is useful in settings such as healthcare, though privacy guarantees are not automatic and must be considered carefully.

2.3.1 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) were introduced by Ian Goodfellow and collaborators in 2014. A GAN consists of two neural networks trained in opposition:

- a **generator** G,
- a **discriminator** D.

The generator produces synthetic data samples intended to resemble real data. The discriminator acts as a critic or binary classifier that tries to distinguish:

- real data from the dataset,
- fake data produced by the generator.

The adversarial process then consists in the generator trying to fool the discriminator, while the discriminator tries not to be fooled.
The generator takes a random latent vector z and transforms it into a synthetic sample:

$$x_{\text{fake}} = G(z),$$

where z is typically sampled from a simple prior such as:

$$z \sim p_z(z),$$

with p_z often chosen as:

- a standard Gaussian,
- or a uniform distribution.

The generator learns a mapping

$$G : \mathcal{Z} \to \mathcal{X},$$

from latent space to data space. In image generation, the generator is often implemented using a deep neural network with upsampling or transposed convolution layers.
The discriminator takes a data instance x and outputs the probability that the sample is real:

$$D(x) \in [0, 1].$$

- $D(x) \approx 1$: discriminator believes x is real,
- $D(x) \approx 0$: discriminator believes x is fake.

For images, the discriminator is often a convolutional neural network.
The standard GAN objective is a two-player min–max game:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))].$$

The first term encourages the discriminator to classify real data as real. The second one encourages the discriminator to classify generated data as fake. The generator wants generated samples to be classified as real, so it tries to minimise the discriminator's success.
GAN training alternates between two updates.

- Train the discriminator. Given:
  - a batch of real data $x \sim p_{\text{data}}$,
  - a batch of generated data $G(z)$,

  update D to improve its ability to distinguish real from fake.
- Train the generator. Update G so that the discriminator is more likely to classify $G(z)$ as real.

In practice, the original generator loss may lead to weak gradients early in training. A common alternative is the **non-saturating generator loss**:

$$\max_G \mathbb{E}_{z\sim p_z}[\log D(G(z))].$$

Equivalently, minimise

$$\mathcal{L}_G = -\mathbb{E}_{z\sim p_z}[\log D(G(z))].$$

This often stabilises training.
For a fixed generator, the optimal discriminator is

$$D^\star(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)},$$

where p_g is the distribution induced by $G(z)$. At the ideal equilibrium:

$$p_g(x) = p_{\text{data}}(x),$$

and therefore

$$D^\star(x) = \frac{1}{2}.$$

This means the discriminator can no longer distinguish real from fake and must guess with 50% confidence.
A **Nash equilibrium** is a state in a game where no player can improve their outcome by changing strategy while the other player's strategy remains fixed. In GANs, this ideal equilibrium occurs when:

- the generator produces perfectly realistic data,

- the discriminator cannot do better than random guessing.

Perfect Nash equilibrium is rarely achieved in practice. GAN training usually seeks a useful approximate balance instead.

An issue faced by GANs is mode collapse. It occurs when the generator learns to produce only a limited subset of the full data distribution. Instead of learning all modes of p_{data}, the generator finds a few outputs that reliably fool the discriminator and repeatedly generates variants of those. As a consequence, generated data may appear high quality, but lacks diversity. Common strategies to mitigate it include:

- **experience replay:** show older generated samples to the discriminator,
- **minibatch discrimination:** let the discriminator compare multiple samples jointly,
- **unrolled GANs:** give the generator a more informed view of discriminator updates,
- **alternative objectives:** such as Wasserstein GANs,
- **architectural improvements:** normalisation, regularisation, better conditioning.

One of the most common ways to evaluate GANs is the Inception Score (IS); it is based on the predicted class distribution of generated samples under a pretrained classifier:

$$\text{IS} = \exp\left(\mathbb{E}_{x\sim p_g}\left[\text{KL}\Big(p(y \mid x) \,\|\, p(y)\Big)\right]\right).$$

A good generator should produce:

- samples with confident class predictions,
- diverse samples across classes.

Another metric is FID. FID compares Gaussian approximations of real and generated feature distributions:

$$\text{FID} = \|\mu_r - \mu_g\|_2^2 + \text{Tr}\left(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\right),$$

where:

- (μ_r, Σ_r) are feature mean/covariance for real data,
- (μ_g, Σ_g) are feature mean/covariance for generated data.

2.3.2 Variational Autoencoders (VAEs)

A Variational Autoencoder (VAE) is a probabilistic generative model that learns:

- how to encode data into a latent distribution,
- how to decode latent samples back into data space.

Unlike a deterministic autoencoder, a VAE does not map an input to a single point in latent space. Instead, it maps the input to a **distribution** over latent variables. A VAE assumes a latent-variable model:

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

where:

- z is a latent variable,
- $p(z)$ is the prior, usually
$$p(z) = \mathcal{N}(0, I),$$
- $p_\theta(x \mid z)$ is the decoder distribution.

Because the true posterior $p_\theta(z \mid x)$ is intractable, the VAE introduces an approximate posterior:

$$q_\phi(z \mid x).$$

A common Gaussian form is

$$q_\phi(z \mid x) = \mathcal{N}\Big(z; \mu_\phi(x), \mathrm{diag}(\sigma^2_\phi(x))\Big).$$

Thus the encoder outputs:

- a mean vector $\mu_\phi(x)$,
- a variance or log-variance vector $\sigma^2_\phi(x)$.

The decoder reconstructs the data from a latent sample:

$$p_\theta(x \mid z).$$

For continuous outputs this may be Gaussian; for binary image pixels it may be Bernoulli. Given input x, sample $z \sim q_\phi(z \mid x)$ and decode:

$$\hat{x} \sim p_\theta(x \mid z).$$

To generate new samples, draw

$$z \sim p(z)$$

from the prior and decode it.

The VAE is trained by maximising the Evidence Lower Bound (ELBO):

$$\mathcal{L}_{\mathrm{ELBO}}(x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] - \mathrm{KL}\Big(q_\phi(z \mid x) \,\|\, p(z)\Big).$$

Equivalently, minimising the negative ELBO gives the common loss:

$$\mathcal{J}_{\mathrm{VAE}}(x) = -\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] + \mathrm{KL}\Big(q_\phi(z \mid x) \,\|\, p(z)\Big).$$

2.4 Reinforcement Learning

Reinforcement learning (RL) is a framework for learning how to make sequential decisions by interacting with an environment and receiving feedback in the form of rewards. An RL problem involves:

- an **agent**,
- an **environment**,
- **observations** or states,
- **actions**,
- **rewards**.

The goal of RL is to learn a behaviour that maximises expected cumulative reward over time. Positive rewards are like pleasure; negative rewards are like pain. The agent learns by trial and error to seek pleasure and avoid pain. Deep reinforcement learning became especially prominent after DeepMind demonstrated in 2013–2015 that a neural network-based system could learn to play many Atari games from raw pixels and later achieved major success with AlphaGo. These achievements showed that combining:

- reinforcement learning,
- deep neural networks,

can solve extremely complex sequential decision problems.
A **policy** is the rule or algorithm the agent uses to choose actions. If s denotes a state and a an action, then a deterministic policy is:

$$a = \pi(s).$$

A stochastic policy defines a probability distribution over actions:

$$\pi(a \mid s) = P(A = a \mid S = s).$$

Suppose a robot vacuum:

- moves forward with probability p,
- turns randomly with probability $1 - p$.

Then the policy has parameters such as p and rotation angle r. Searching over such parameters is called **policy search**.
If the agent receives reward R_{t+1} after acting at time t, the discounted return from time t is

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where:

- $0 \leq \gamma < 1$ is the discount factor,
- larger γ values place more importance on future rewards.

Discounting makes immediate rewards more important than distant rewards and ensures the infinite sum is well behaved.
Lete us now talk about Markov Decision Processes. A process is Markovian if the future depends only on the present state, not on the entire past. Formally, for states S_t,

$$P(S_{t+1} \mid S_t, S_{t-1}, \ldots, S_0) = P(S_{t+1} \mid S_t).$$

A Markov Decision Process is typically defined by:

$$(\mathcal{S}, \mathcal{A}, P, R, \gamma),$$

where:

- $\mathcal{S}$: set of states,
- $\mathcal{A}$: set of actions,
- $P(s' \mid s, a)$: transition probabilities,
- $R(s, a, s')$: reward function,
- γ: discount factor.

The transition probability from state s to s' given action a is

$$P(s' \mid s, a).$$

The reward for transitioning from s to s' after action a is

$$R(s, a, s').$$

The optimal state value $V^\star(s)$ is the maximum expected return starting from state s and acting optimally thereafter:

$$V^\star(s) = \max_\pi \mathbb{E}_\pi[G_t \mid S_t = s].$$

Bellman showed that $V^\star$ satisfies the recursive equation

$$V^\star(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V^\star(s')\right].$$

The optimal value of the current state equals the best expected immediate reward plus the discounted value of the next state. A practical algorithm is value iteration:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V_k(s')\right].$$

Starting from initial values such as $V_0(s) = 0$, repeated application converges under standard assumptions to $V^\star(s)$.
The optimal Q-value for a state-action pair (s, a) is the expected return obtained by taking action a in state s and then acting optimally:

$$Q^\star(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^\star(S_{t+1}, a') \mid S_t = s, A_t = a\right].$$

Bellman Optimality Equation for Q:

$$Q^\star(s,a) = \sum_{s'} P(s' \mid s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^\star(s',a') \right].$$

Q-Iteration iterative update is

$$Q_{k+1}(s,a) = \sum_{s'} P(s' \mid s,a) \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right].$$

Once $Q^\star$ is known, the optimal policy is

$$\pi^\star(s) = \arg\max_a Q^\star(s,a).$$

For complex tasks such as AlphaGo or Atari, the state space is too large for exact tabular methods. A neural network can take an observation o_t or state s_t as input and output a probability distribution over actions:

$$\pi_\theta(a \mid s).$$

Sampling actions allows a balance between:

- **exploration**: trying actions not yet known to be best,
- **exploitation**: favouring actions already believed to work well.

A common stochastic policy uses logits $u_\theta(s,a)$ and a softmax:

$$\pi_\theta(a \mid s) = \frac{\exp(u_\theta(s,a))}{\sum_{a'} \exp(u_\theta(s,a'))}.$$

Q-learning learns state-action values directly from experience using

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t) \right].$$

The agent updates its current estimate towards:

- the observed immediate reward,
- plus the best estimated future value.

If Q-values become accurate, the optimal policy is simply greedy:

$$\pi(s) = \arg\max_a Q(s,a).$$

Q-learning can learn an optimal greedy policy even if the agent explores using a different behaviour policy.

A central RL dilemma is deciding whether to:

- exploit what is currently known to work,
- or explore unknown actions that might be better.

A common strategy is the ε-greedy policy:

- with probability ε, choose a random action,
- with probability $1 - \varepsilon$, choose the greedy action.

Formally,

$$a_t = \begin{cases} \text{random action}, & \text{with probability } \varepsilon, \\ \arg\max_a Q(s_t, a), & \text{with probability } 1 - \varepsilon. \end{cases}$$

One often starts with large ε and gradually decays it over time. Another approach is to add curiosity or novelty bonuses. A conceptual exploration-adjusted score may be

$$Q_{\text{explore}}(s, a) = Q(s, a) + F(N(s, a)),$$

where:

- $N(s, a)$ counts how often action a was tried in state s,
- F gives higher bonus to less-visited pairs.

For example,

$$F(N) = \frac{k}{N + 1},$$

where k is a curiosity hyperparameter.

3 Practical Methodology and Ethics in AI

3.1 Practical Methodology

A machine learning model must be evaluated using metrics that reflect the actual goal of the problem. Choosing the wrong metric can lead to misleading conclusions about the model quality and its suitability to the given task. When choosing a metric, ask yourself questions like:

- What is the task?
- What kind of errors matter the most?
- Is the dataset balanced or otherwise?
- Is this a classification, regression, ranking, detection, or structured prediction problem?

For classification, one often uses a confusion matrix to support the analysis of the model's performance. Here, you define each quadrant of the matrix as:

- True Positive (TP): predicted positive, actually positive,
- True Negative (TN): predicted negative, actually negative,
- False Positive (FP): predicted positive, actually negative,

- False Negative (FN): predicted negative, actually positive.

Accuracy is the measure of the model's overall correctness:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

It is useful when classes are balanced and false positives are similarly costly to false negatives. If the dataset is imbalanced, however, accuracy can be misleading. For instance, if 99% of samples belong to one class, a trivial classifier predicting only that class achieves 99% accuracy but may be useless.

Another metric is that of **precision**, how many predicted positives are actually correct:

$$\text{Precision} = \frac{TP}{TP + FP}.$$

It matters the most when false positives are costly, such as with spam detection or fraud alerts.

Then, there is **recall**, the measure of how many actual positives are recovered:

$$\text{Recall} = \frac{TP}{TP + FN}.$$

It matters when false negatives are costly, such in the case of disease detection or safety-critical fault detection. The F1 Score is the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Since this metric penalises imbalance between precision and recall, a high F1 score requires both metrics to be reasonably good. More generally, one may use

$$F_\beta = (1 + \beta^2)\frac{\text{Precision} \cdot \text{Recall}}{\beta^2\text{Precision} + \text{Recall}},$$

where:

- $\beta > 1$ emphasizes recall,
- $\beta < 1$ emphasizes precision.

Specifically for the task of classification, other relevant metrics are:

- Specificity:
$$\text{Specificity} = \frac{TN}{TN + FP}.$$
- Balanced Accuracy:
$$\text{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right).$$
- ROC-AUC: The area under the receiver operating characteristics curve, which evaluates ranking quality across thresholds.

- PR-AUC: The area under the Precision-Recall curve, which might be more useful for imbalanced datasets.

When it comes to regression, other metrics should be used. Some commmonly-used ones are:

- Mean Squared Error: It strongly penalises large errors

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

- Root Mean Squared Error: It is on the same scale as the target variable

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

- Mean Absolute Error: It is more robust to outliers

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|.$$

- R^2 Score: It measures variance explained relative to a constant predictor

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}.$$

Another important concept to be familiar with is that of a **baseline**. A baseline is a simple model used for comparison, and helps establish whether the solution being developed is actually better than something simple. In general, baselines are essential for benchmarking, understanding whether complexity is justified, and diagnosing whether the data contains any useful signal.
A sample linear regression model baseline might look like:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$

For the case of logistic regression, a classification baseline might look like:

$$P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b),$$

where

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Once a baseline is established, performance can be improved by adding more data, improving the model architecture, choosing different optimisation algorithms, tuning hyperparameters, using transfer learning, applying regularisation, or other techniques.

One of the most common optimisation strategies is **gradient descent**. The standard gradient descent update is

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t),$$

where η is the learning rate.

In practice, full-batch gradient descent is rarely used for large datasets. One usually uses stochastic or mini-batch variants. Using a mini-batch $\mathcal{B}_t$,

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}_{\mathcal{B}_t}(\theta_t).$$

This is desirable because it has lower computational cost than full-batch training, more stable gradients than pure SGD, and more efficient hardware utilisation.
Momentum accelerates optimisation by accumulating a velocity term,

$$v_{t+1} = \mu v_t - \eta \nabla_\theta \mathcal{L}(\theta_t),$$

$$\theta_{t+1} = \theta_t + v_{t+1},$$

where $\mu \in [0, 1)$ is the momentum coefficient. This helps move through shallow valleys and damp oscillations.
RMSProp is an algorithm that adapts the step size per parameter using an exponential moving average of squared gradients in order to handle gradient scales more robustly:

$$s_{t+1} = \rho s_t + (1 - \rho)\Big(\nabla_\theta \mathcal{L}(\theta_t)\Big)^2,$$

$$\theta_{t+1} = \theta_t - \eta \frac{\nabla_\theta \mathcal{L}(\theta_t)}{\sqrt{s_{t+1}} + \epsilon}.$$

Adam, one of the most popular optimisation algorithms, combines momentum and adaptive scaling:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1)\nabla_\theta \mathcal{L}(\theta_t),$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)\Big(\nabla_\theta \mathcal{L}(\theta_t)\Big)^2.$$

Bias-corrected estimates are:

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}},$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}.$$

Then update:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \epsilon}.$$

The choice of optimiser largely depends on the data and training regime. For example:

- noisy gradients from small mini-batches may favor Adam or RMSProp,
- very large datasets often work well with SGD + momentum,
- sparse features may benefit from adaptive methods,
- poorly scaled features can destabilize basic SGD.

3.2 Ethics in AI

AI systems can have a big influence in several aspects of human life, such as hiring, lending, criminal justice, social media, surveillance, education, public policy, and many others. Because of that, it is important to pause and consider the ethical implications of AI technologies. Within this ethical analysis, math plays a crucial role, because it allows us to formalise concepts like fairness, privacy, transparency, accountability, risk, and robustness. Mathematical definitions help bring rigor to ethical reasoning and provide a thorough look when being complemented with social, legal, and philosophical judgement. There are four themes that frequently arise in discussions of AI ethics:

1. Fairness: ensuring equitable treatment and opportunity across individuals and groups.
2. Privacy: protecting individuals from misuse or exposure of personal data.
3. Transparency: enabling people to understand how an AI system works or makes decisions.
4. Accountability: ensuring that responsibility can be assigned when harm occurs.

These themes are often interdependent and sometimes in tension with one another. In broad terms, fairness means that an AI system should not systematically disadvantage people based on sensitive attributes like race, gender, or socioeconomic background. It often involves equal treatment, equal opportunity, or equitable outcomes across user groups. The challenge in this principle is the fact that fairness is not a single universally agreed-upon concept; different applications may require different fairness criteria. Furthermore, bias can enter at many stages of the model development process. For example, one might perform a biased data collection, use proxy variables correlated with sensitive attributes, or use a biased model architecture and training choices. Thus, even mathematicall sophisticated models can reproduce or even amplify social inequities.
In order to develop fariness-aware algorithms, it is important to be familiar with the notion of **equality of opportunity**. Let:

- X denote features,
- A denote a sensitive attribute (for example, group membership),
- $Y \in \{0, 1\}$ denote the true outcome,
- $\widehat{Y} \in \{0, 1\}$ denote the prediction.

Equality of opportunity requires equal **true positive rates** across groups:

$$P(\widehat{Y} = 1 \mid Y = 1, A = a) = P(\widehat{Y} = 1 \mid Y = 1, A = b)$$

for all sensitive groups a, b. This means that, among individuals who truly qualify for a positive outcome (e.g. receiving a job offer or being approved for a loan), the probability of receiving that positive prediction should be the same across groups.
The true positive rate for a group $A = a$ is

$$\mathrm{TPR}_a = P(\widehat{Y} = 1 \mid Y = 1, A = a).$$

Equality of opportunity then requires

$$\mathrm{TPR}_a = \mathrm{TPR}_b$$

for all relevant groups. An even stronger notion is that of **equalised odds**, which requires both equal true positive and false positive rates:

$$P(\widehat{Y} = 1 \mid Y = y, A = a) = P(\widehat{Y} = 1 \mid Y = y, A = b) \quad \text{for } y \in \{0, 1\}.$$

Equivalently:

$$P(\widehat{Y} = 1 \mid Y = 1, A = a) = P(\widehat{Y} = 1 \mid Y = 1, A = b),$$

$$P(\widehat{Y} = 1 \mid Y = 0, A = a) = P(\widehat{Y} = 1 \mid Y = 0, A = b).$$

In addition to equalised odds and equality of opportunity, another fairness notion is **demographic parity**:

$$P(\widehat{Y} = 1 \mid A = a) = P(\widehat{Y} = 1 \mid A = b).$$

This requires equal positive prediction rates across groups, regardless of the true label. It is important to note that these three are different criteria that may not be compatible within the same application. Different fairness criteria may conflict with one another. In real-world settings, one often faces trade-offs between:

- fairness and accuracy,
- fairness and calibration,
- fairness and privacy,
- fairness across multiple protected groups,
- fairness to individuals vs fairness to groups.

Thus, fairness is not simply a technical constraint; it demands a normative choice about which notion of fairness is the most appropriate given the problem's context. Fairness interventions to mitigate bias usually take place at three stages:

1. Pre-processing: This implies modifying the data before training in order to avoid the model from perpetuating the data's biases. Some sample techniques are reweighting, resampling, repairing labels, and balancing representation.

2. In-processing: Modifying the training algorithm itself to enforce fairness constraints. A generic learning objective could be written as:

$$\min_{\theta} \mathcal{L}_{\text{task}}(\theta) + \lambda \, \mathcal{R}_{\text{fair}}(\theta),$$

   where:

   - $\mathcal{L}_{\text{task}}$ is the predictive loss,
   - $\mathcal{R}_{\text{fair}}$ penalizes fairness violations,
   - λ controls the trade-off.

3. Post-processing: Modifying the model's outputs after training to satisfy fairness criteria. For example, there is threshold adjustment by group, score callibration, and rejection option classification.

AI systems often use data about individuals, such as health records, censusdata, financial behaviour, browsing history, facial images, location traces, and voice recordings. This makes privacy a key concern when it comes to discussing these systems. Generally, one should aim to minimise the risk that an individual could be identified, tracked, or harmed through the use or release or data. Privacy must be considered even against adversaries who may possess auxiliary information from other sources. **Differential privacy** is a rigurous framework for limiting the privacy risk that a person incurs when participating in a statistical database. The idea is that the output of an analysis should not change *too much* depending on whether a given individual's data is included or excluded. More formally, let K be a randomised mechanism operating on datasets. Let D and D' be **neighboring datasets**, meaning they differ in the data of exactly one individual. Then, K is ε-differentially private if for all measurable sets S,

$$P(K(D) \in S) \leq e^{\varepsilon}\, P(K(D') \in S).$$

Here, a smaller ε implies stronger privacy, because the two probabilities must remain closer. A relaxed version is (ε, δ)-differential privacy:

$$P(K(D) \in S) \leq e^{\varepsilon} P(K(D') \in S) + \delta.$$

Here:

- ε controls the multiplicative privacy loss,
- δ allows a small probability of failure.

To apply noise mechanisms, one defines the **global sensitivity** of a query f:

$$\Delta f = \max_{D,D'} \|f(D) - f(D')\|_1,$$

where the maximum is taken over neighboring datasets D, D'. This measures the largest change that one person's data can cause in the query result. A standard way to achieve differential privacy is to add Laplace noise to the true query answer. If the true query is $f(D)$, then the released value is

$$K(D) = f(D) + Z,$$

where

$$Z \sim \text{Lap}\left(0, \frac{\Delta f}{\varepsilon}\right).$$

The Laplace density is

$$p(z) = \frac{1}{2b} \exp\left(-\frac{|z|}{b}\right),$$

where

$$b = \frac{\Delta f}{\varepsilon}.$$

Here, a larger sensitivity Δf requires more noise, smaller ε requires more noise, and stronger privacy generally reduces accuracy. Another common mechanism adds Gaussian noise:

$$K(D) = f(D) + \mathcal{N}(0, \sigma^2).$$

This is often used for (ε, δ)-differential privacy.
Transparency in an AI system refers to openness and clarity about how it is built, what data it uses, what features influence its outputs, how it behaves under different conditions, and how decisions can be questioned or appealed. Transparency matters because it allows for the public to have confidence in the AI model, for it to be audited, and to detect biases. Accountability, on the other hand, refers to the ability to identify who is responsible for the design, deployment, and consequences of an AI system. It generally includes the developers, deployers, decision makers, regulators, and organisations involved in the development of the system.

Because many AI systems are difficult to interpret directly, a major research area is **Explainable AI (XAI)**. XAI aims to explain how an individual prediction is made, the role each feature had in arriving at that prediction, identifying failure modes, and enabling appeals or recourse, in order to increase trust and accountability.

An XAI technique for cases in which models prove too complex to interpret is LIME, Local Interpretable Model-Agnostic Explanations. It aims to explain a complex model by approximating it with a simpler, interpretable model around the instance of interest. Let:

- f be the original black-box model,
- g be an interpretable surrogate model,
- $\pi_x(z)$ be a proximity measure around the instance x,
- $\mathcal{L}(f, g, \pi_x)$ be a local fidelity loss,
- $\Omega(g)$ be a complexity penalty on the explanation model.

LIME solves

$$\arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g).$$

LIME:

1. perturbs the input around the point x,
2. queries the black-box model f on those perturbed samples,
3. weights samples according to closeness to x,
4. fits a simple interpretable model g,
5. uses g as the explanation.

LIME is model-agnostic and intuitive, but its explanations are local and may prove unstable depending on perturbation choices.
Another technique is that of SHAP, Shapley Additive Explanations. It uses ideas from

cooperative game theory to assign each feature a contribution value to a model's prediction. Let N be the set of all features and i a particular feature. The Shapley value for feature i is

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(|N| - |S| - 1)!}{|N|!} \Big(f_x(S \cup \{i\}) - f_x(S) \Big).$$

Here:

- S is a subset of features not containing i,
- $f_x(S)$ is the model output when only features in S are known,
- ϕ_i is the contribution of feature i.

SHAP explanations are often written as

$$f(x) \approx \phi_0 + \sum_{i=1}^{d} \phi_i,$$

where:

- ϕ_0 is a baseline prediction,
- ϕ_i is the contribution of feature i.

SHAP provides local accuracy, consistency, and principled attribution; however, exact computation can become prohibitively expensive for many features.
Partial Dependence Plots (PDP) show the average effect of one feature on the predicted outcome, averaging over the other features. Let x_s denote the feature(s) of interest and x_c the remaining features. Then the partial dependence function is

$$\widehat{f}_{x_s}(x_s) = \frac{1}{n} \sum_{i=1}^{n} f(x_s, x_c^{(i)}).$$

Here:

- $x_c^{(i)}$ denotes the other features from the i-th observation,
- f is the prediction function,
- n is the number of observations.

PDP estimates how predictions change as the feature of interest varies, while averaging out the influence of the remaining features. This is useful for global model behaviour, but can be misleading when features are strongly correlated.

AI ethics often involves unavoidable trade-offs:

- Fairness vs Accuracy: Imposing fairness constraints can alter the model decision boundary and sometimes reduce predictive accuracy. This can be represented abstractly as

  $$\min_{\theta} \mathcal{L}_{\text{task}}(\theta) \quad \text{subject to fairness constraints},$$

  or equivalently,

  $$\min_{\theta} \mathcal{L}_{\text{task}}(\theta) + \lambda \mathcal{R}_{\text{fair}}(\theta).$$

- Privacy vs Utility: Stronger privacy often requires more noise,

$$\varepsilon \downarrow \implies \text{noise} \uparrow \implies \text{utility} \downarrow .$$

- Transparency vs Performance: Highly interpretable models are often more simple, whereas more complex ones might achieve better performance.
- Accountability vs Automation: The more decisions are delegated to automated systems, the more important it becomes to define clear lines of human responsibility.

Ethical AI is not achieved merely by training a model once. It requires attention throughout the full lifecycle:

$$\text{data collection} \to \text{training} \to \text{evaluation} \to \text{deployment} \to \text{monitoring}.$$

3.3 Structured Probabilistic Modelling for Deep Learning

Structured probabilistic models provide a framework to reason about uncertainty in neural networks and in real-world systems. Every real prediction has uncertainty which can be represented explicitly and reasoned through structured probabilistic models. On top of that, they allow one to attach probabilities to each possible state of the world, update them when new information arrives, and make decisions under uncertainty. Such models are especially useful when data is noisy, incomplete, or uncertain, which is the norm in real applications.
A probabilistic model is just a mathematical framework for representing uncertainty in a system using probability theory. If X denotes observed variables and Z denotes latent or unobserved variables, then a typical probabilistic model specifies a joint distribution

$$p(X, Z).$$

This joint distribution allows us to compute the probabilities of events, infer hidden causes, make predictions, and update beliefs after observing new data. Uncertainty is represented using probability distributions. For example:

- $p(x)$: marginal distribution,
- $p(y \mid x)$: conditional distribution,
- $p(\theta)$: prior over parameters,
- $p(\theta \mid \mathcal{D})$: posterior after observing data $\mathcal{D}$.

A key feature of probabilistic reasoning is that beliefs are updated as new data arrives. This is formalized by Bayes' theorem:

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)\, p(\theta)}{p(\mathcal{D})}.$$

Here:

- $p(\theta)$ is the prior,
- $p(\mathcal{D} \mid \theta)$ is the likelihood,

- $p(\theta \mid \mathcal{D})$ is the posterior,
- $p(\mathcal{D})$ is the evidence or marginal likelihood.

The prior expresses what we believed before seeing the data; the posterior expresses what we believe after incorporating the data.
Within deep learning, probabilistic models are most prominently integrated in three techniques: Bayesian neural networks, variational autoencoders (VAEs), and Gaussian processes with deep neural networks.
A Bayesian neural network (BNN) places probability distributions over network weights rather than treating them as fixed numbers. VAEs, on the other hand, combine neural networks with latent-variable probabilistic models and are especially important in generative modeling. Lastly, the idea behind Gaussian processes with deep neural networks is that deep neural networks can be used to learn representations or kernels for Gaussian processes. A common deep-kernel-learning form is

$$f \sim \mathcal{GP}\Big(m(\cdot),\, k_\theta(\phi_\psi(x), \phi_\psi(x'))\Big),$$

where:

- $\phi_\psi(x)$ is a learned deep representation,
- k_θ is a kernel function,
- $\mathcal{GP}$ denotes a Gaussian process prior over functions.

This combines deep feature learning with principled uncertainty estimation.

Let us go deeper into Bayesian neural networks. In a standard neural network, the weights are fixed after training:

$$\mathbf{w}^\star = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}).$$

Predictions are then made using that single point estimate. In a Bayesian neural network, the weights are treated as random variables with a probability distribution:

$$\mathbf{w} \sim p(\mathbf{w}).$$

Instead of learning one fixed $\mathbf{w}$, the model learns a posterior distribution over weights:

$$p(\mathbf{w} \mid \mathcal{D}).$$

This means the model represents uncertainty about which weight values are plausible given the data. Given dataset $\mathcal{D}$, Bayes' theorem gives the posterior distribution over weights:

$$p(\mathbf{w} \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \mathbf{w})\, p(\mathbf{w})}{p(\mathcal{D})}.$$

The denominator is the evidence:

$$p(\mathcal{D}) = \int p(\mathcal{D} \mid \mathbf{w})\, p(\mathbf{w})\, d\mathbf{w}.$$

where,

- $p(\mathbf{w})$: prior belief about the weights,
- $p(\mathcal{D} \mid \mathbf{w})$: likelihood of the data under those weights,
- $p(\mathbf{w} \mid \mathcal{D})$: updated belief after observing the data.

Then, for a new input $x^\star$, the predictive distribution is obtained by integrating over all possible weights:

$$p(y^\star \mid x^\star, \mathcal{D}) = \int p(y^\star \mid x^\star, \mathbf{w})\, p(\mathbf{w} \mid \mathcal{D})\, d\mathbf{w}.$$

This means that predictions are averaged over all plausible models, weighted by their posterior probability. Bayesian neural networks provide principled uncertainty estimates that naturally incorporate prior knowledge. Because of this, they are valuable in high-stakes tasks with smaller datasets, such as healthcare or scientific modelling. That being said, posterior inference is computationally expensive, the parameter space is very high-dimensional, and exact posterior computation is usually intractable. Thus, approximate inference introduces new trade-offs. The reason why posterior calculation is hard has to do with the evidence term:

$$p(\mathcal{D}) = \int p(\mathcal{D} \mid \mathbf{w})\, p(\mathbf{w})\, d\mathbf{w}.$$

In a deep neural network, $\mathbf{w}$ may contain millions of parameters. The likelihood landscape there is highly convex and there may be many local optima and complicated curvature. Thus, direct integration over all parameter configurations is infeasible. For most nontrivial neural networks, the exact posterior cannot be computed in closed form.
Because the exact posterior is intractable, one uses approximation methods. Variational inference is one such method that approximates the true posterior $p(\mathbf{w} \mid \mathcal{D})$ by a simpler distribution $q_\phi(\mathbf{w})$. The goal is to make $q_\phi(\mathbf{w})$ close to the posterior by minimizing the Kullback–Leibler divergence:

$$\mathrm{KL}\Big(q_\phi(\mathbf{w}) \,\|\, p(\mathbf{w} \mid \mathcal{D})\Big).$$

Since the true posterior is unavailable, one instead maximizes the evidence lower bound (ELBO):

$$\mathcal{L}_{\mathrm{ELBO}}(\phi) = \mathbb{E}_{q_\phi(\mathbf{w})}\Big[\log p(\mathcal{D} \mid \mathbf{w})\Big] - \mathrm{KL}\Big(q_\phi(\mathbf{w}) \,\|\, p(\mathbf{w})\Big).$$

This method effectively transforms posterior inference into an optimization problem, which is usually more computationally tractable than direct integration.
Another method is to use Markov Chain Monte Carlo (MCMC) algorithms. MCMC methods sample from the posterior without computing it explicitly. The idea is to construct a Markov chain whose stationary distribution is the target posterior:

$$\pi(\mathbf{w}) = p(\mathbf{w} \mid \mathcal{D}).$$

After sufficient iterations, samples from the chain approximate samples from the posterior.

A Bayesian neural network's predictive distribution captures both aleatoric and epistemic uncertainty. For regression, a useful variance decomposition is:

$$\mathrm{Var}(y^\star \mid x^\star, \mathcal{D}) = \mathbb{E}_{p(\mathbf{w}|\mathcal{D})}\Big[\sigma^2(x^\star, \mathbf{w})\Big] + \mathrm{Var}_{p(\mathbf{w}|\mathcal{D})}\Big[f_{\mathbf{w}}(x^\star)\Big].$$

Here:

- the first term corresponds to aleatoric uncertainty,
- the second term corresponds to epistemic uncertainty.

This decomposition is highly useful because it tells us whether uncertainty comes from noisy data or from lack of model knowledge.