Ahmed Ibrahim

ECE 9039/9309 MACHINE LEARNING

Last Lecture

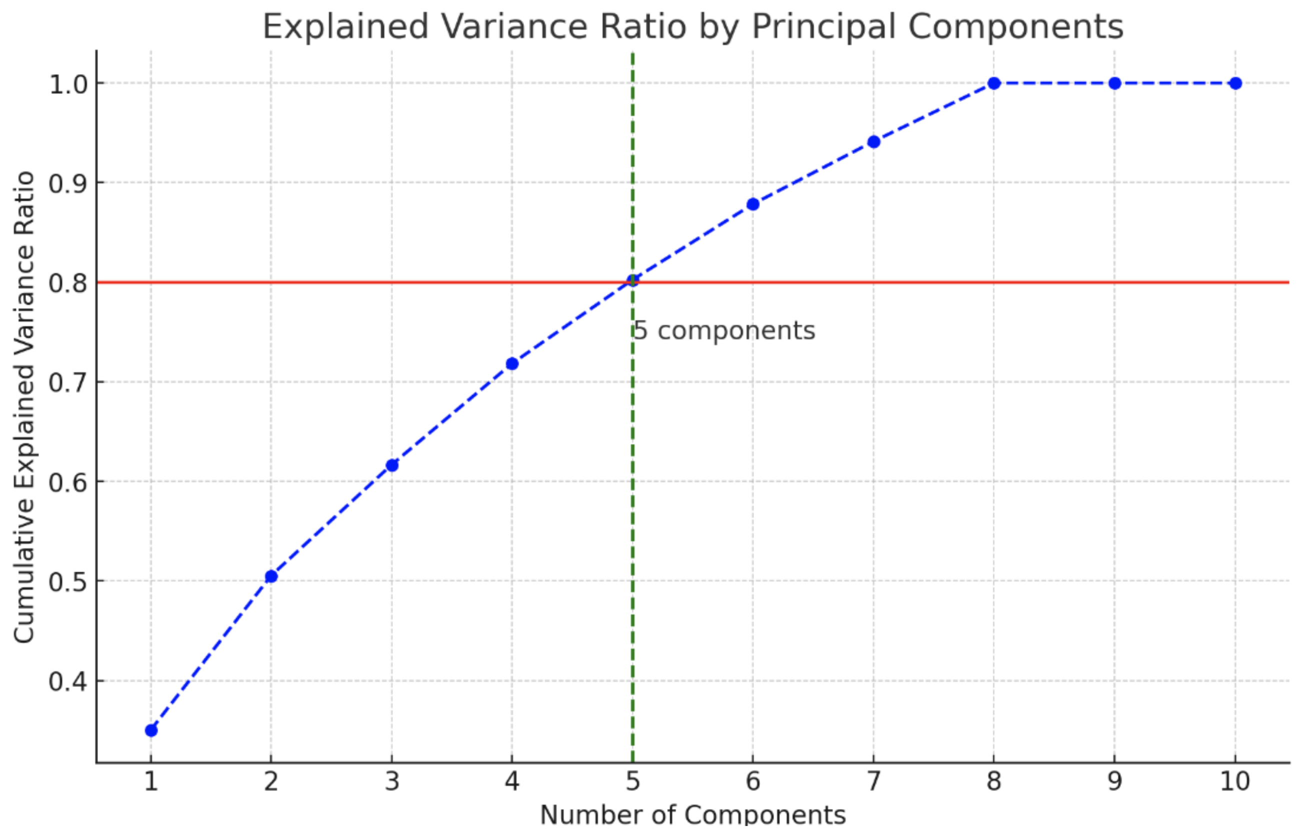
- Some Practical Exercises
- Design Neural Networks
- Popular Frameworks for ANN
- ANN Applications: Image Classification
  - Common Feature Extraction Approaches
- Deep Learning
  - Convolutional Neural Networks
- Deep Learning Challenges

Outlines

- Assignment #2 Orientation
- Autoencoders
  - Autoencoder Architecture
  - The Latent Space
  - Common Types of Autoencoders
- Hyperparameter Optimization (HPO)
- Recurrent Neural Network

Assignment #2 Orientation

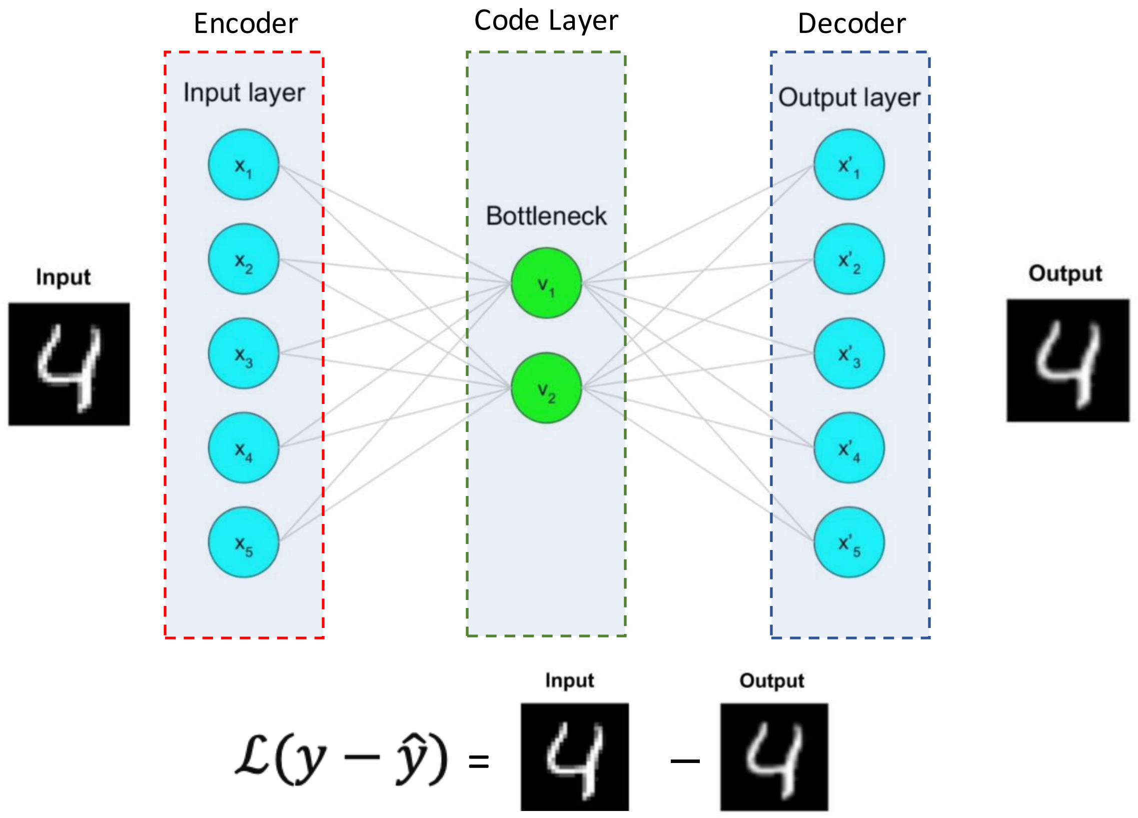
- For this example, I use a synthetic dataset generated with **sklearn.datasets.make_classification.**
- Step 1: Generate a dataset with 100 samples and 10 features.
- Step 2: apply PCA to the dataset to analyze the principal components.
- Step 3: Create a plot using *cumsum(),* with the x-axis representing the number of principal components and the y-axis representing the cumulative explained variance ratio.
- Mark the point where the cumulative variance reaches 80%.
- MSE & MAE values
- Drop 'StartupID'



Autoencoders

Autoencoders

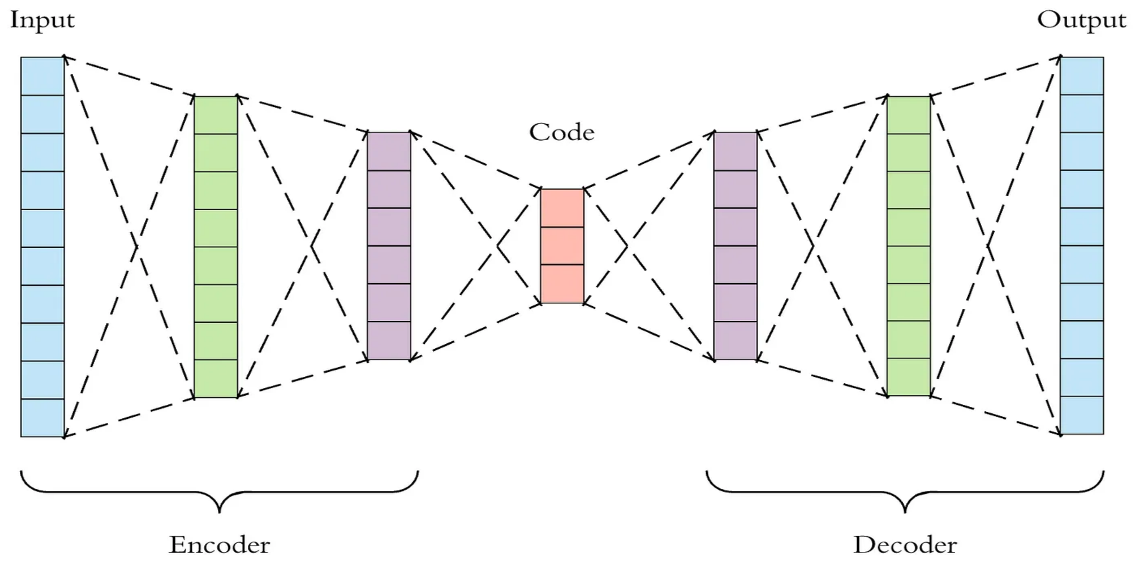
- Autoencoders are ANNs that learn to compress (encode) the input into a lower-dimensional code and then decompress (decode) the code to reconstruct the input as closely as possible.
- This process forces the autoencoder to capture the <u>most important features</u> present in the data.
- Autoencoders are useful in various applications, such as data **denoising**, **dimensionality reduction** for data visualization, and **feature extraction**.



Source: https://lilianweng.github.io/posts/2018-08-12-vae/

Autoencoder Architecture

- The **encoder layer** translates the original high-dimension input into the latent low-dimensional code.
- The input size (# of nodes) is larger than the output size.



- The # of nodes in the code layer (**code size**) is a **hyperparameter** we set before training the autoencoder.

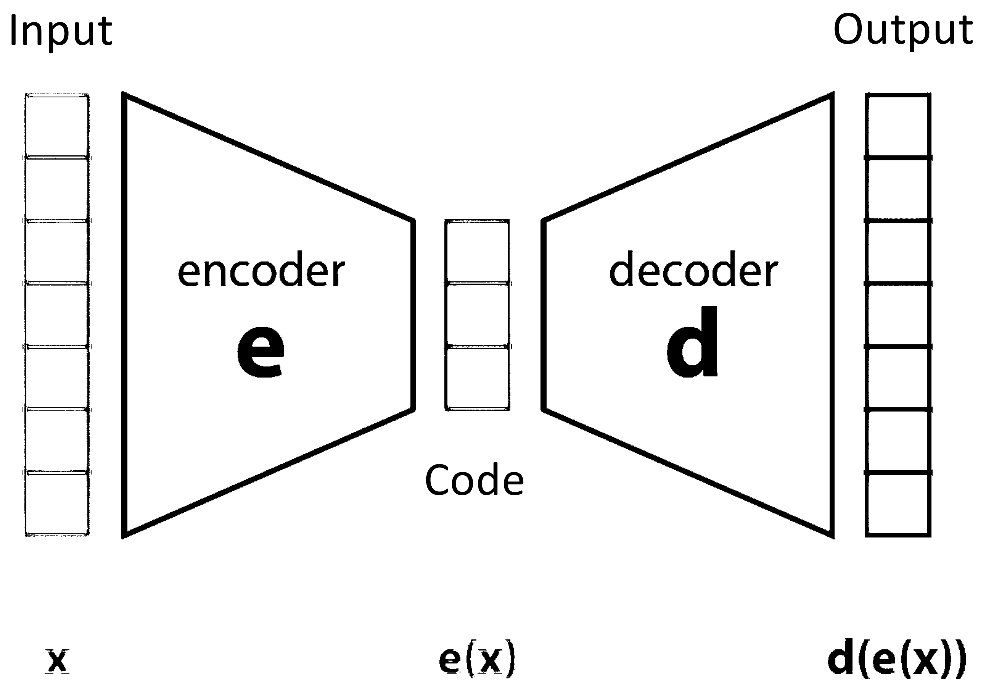
- The **decoder layer** recovers the data from the **code**, likely with larger and larger output layers.
- The input size (# of nodes) is smaller than the output size.

Source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Autoencoder Properties

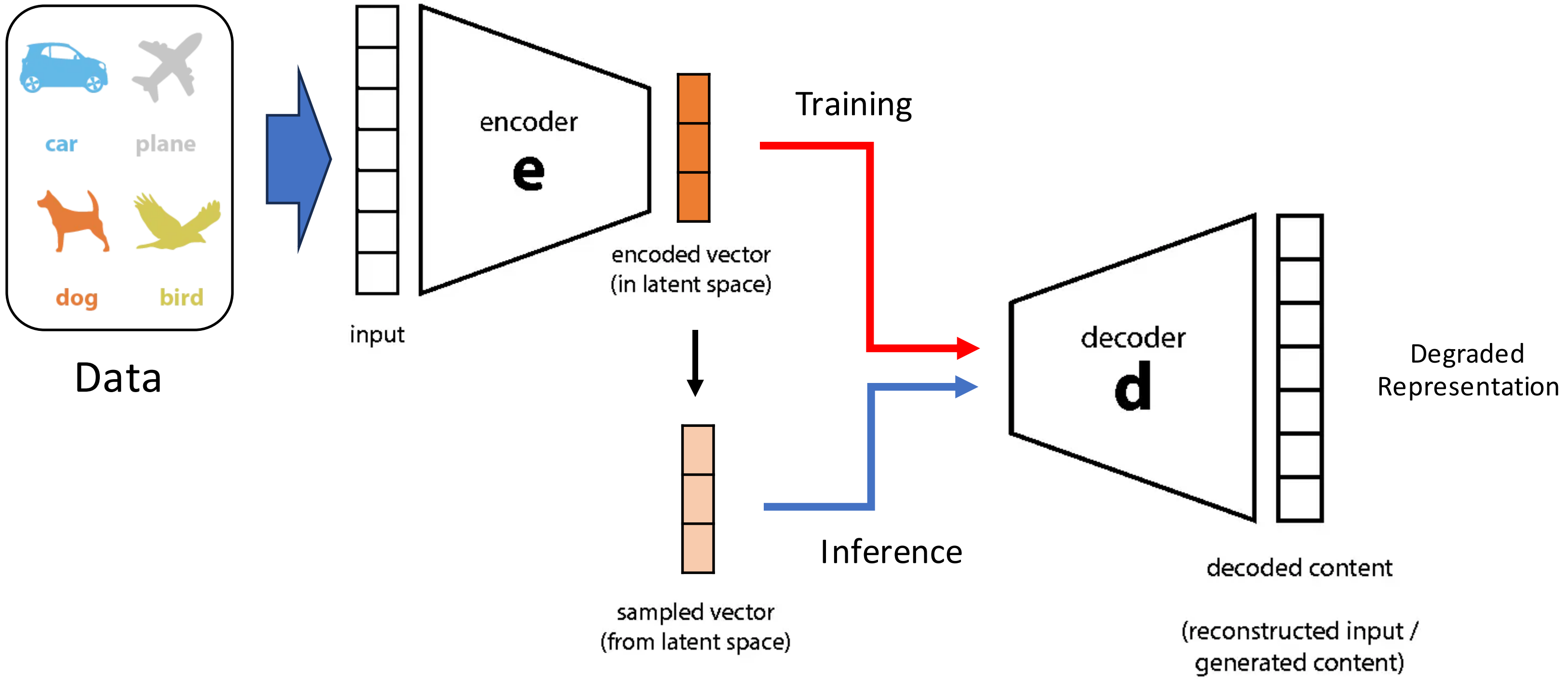
- **Lossy**—The autoencoder's output will not be the same as the input; it will be a close but degraded representation.

- $x \neq d(e(x))$ ➡ **lossy encoding** some information is lost when reducing the number of dimensions and can't be recovered later



Source: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

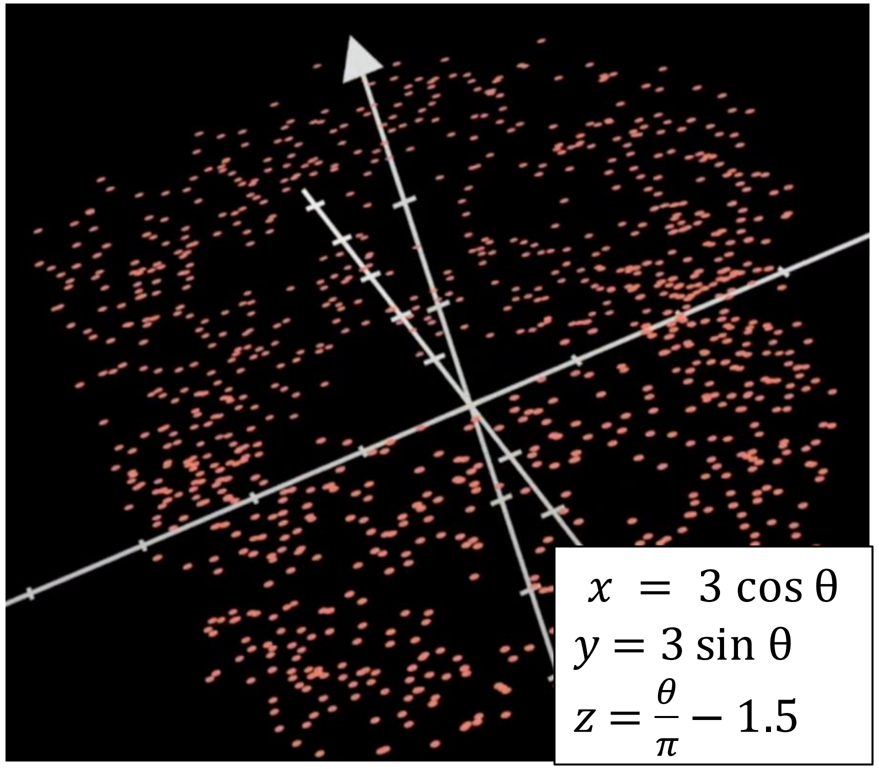
Autoencoder Properties (cont.)



- **Regenerate** – We can generate new data by decoding points randomly sampled from the **latent space**. The quality and relevance of generated data depend on the regularity of the **latent space**.

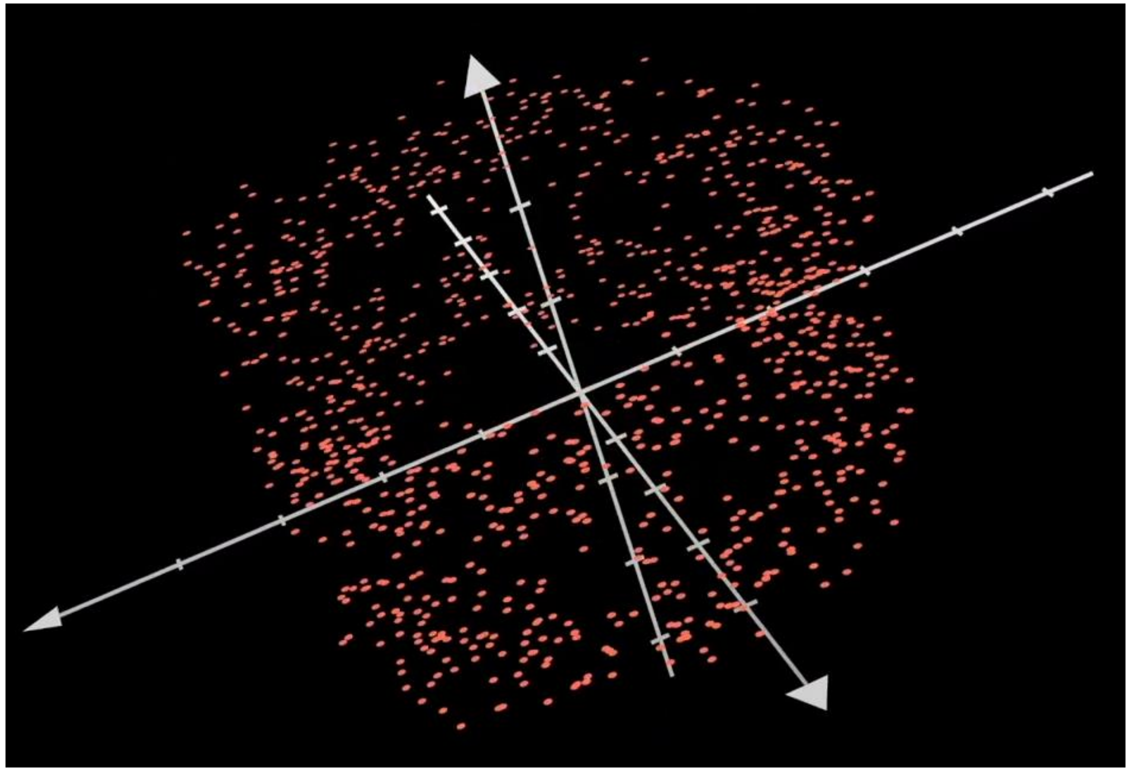
Source: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

Latent Space



$$x = 3\cos\theta$$
$$y = 3\sin\theta$$
$$z = \frac{\theta}{\pi} - 1.5$$

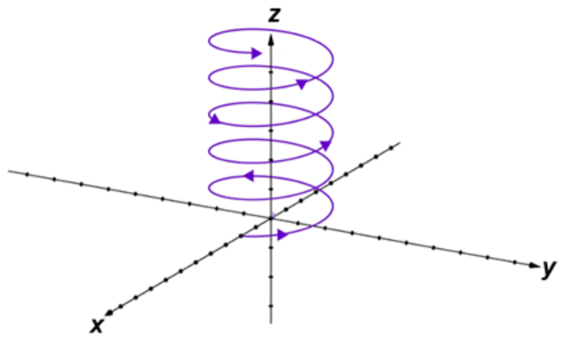
Data in the Latent Space



Data Hidden Structure

Source: https://youtu.be/3jmcHZq3A5s?t=7

The Latent Space



- The term "latent" describes the compressed representation of the input data.
- This latent representation captures the underlying patterns or features of the data that are not immediately obvious.
- The **latent space**, therefore, is the space in which this representation exists.
- It is called latent because it encodes hidden or less obvious aspects of the data.

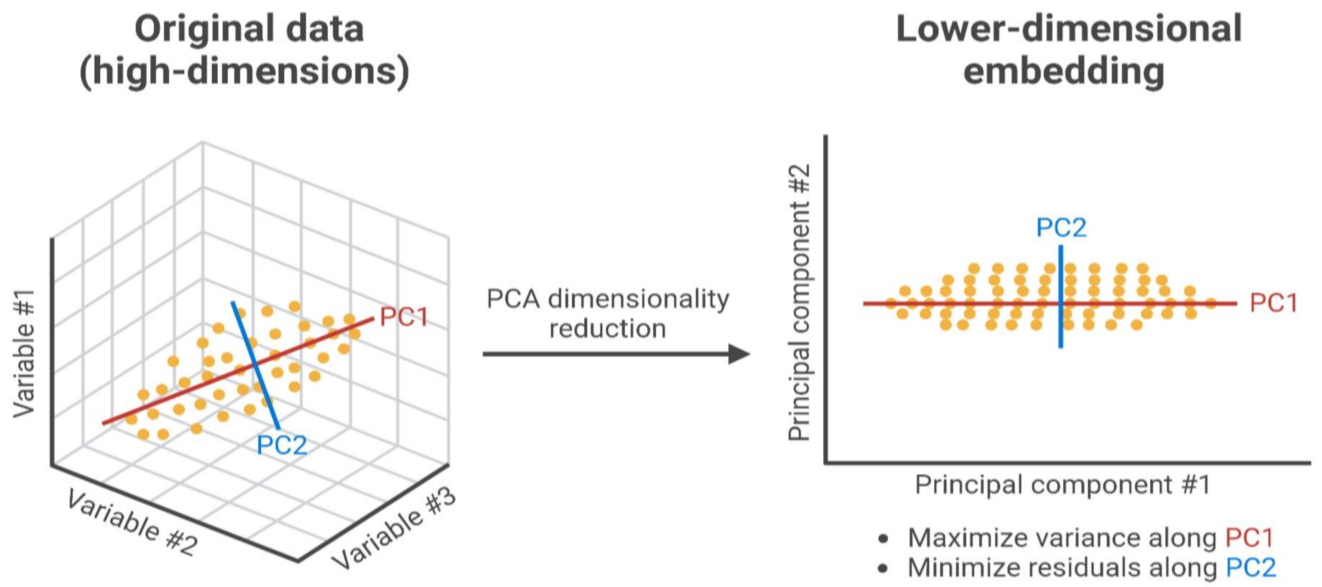
Source: https://www.math.net/spiral

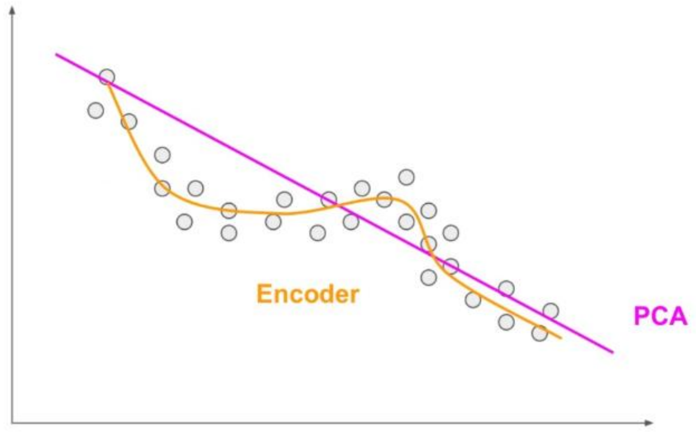
Autoencoder Hyperparameters

- There are 4 hyperparameters that we need to set before training an autoencoder:
  - **Code size** – number of nodes in the middle layer. Smaller sizes result in more compression.
  - **The number of layers** – the autoencoder can be as deep as we like.
  - **Number of nodes per layer** – The layers are stacked one after another.
  - **Loss function** – mean squared error (MSE) or binary cross entropy.

Autoencoders vs. PCA

- Both perform dimensionality reduction.
- **PCA** learns **linear relationships**.
- **Autoencoders** can learn **nonlinear relationships** (using nonlinear activation functions).
- With a linear activation function, the autoencoder is equivalent to PCA



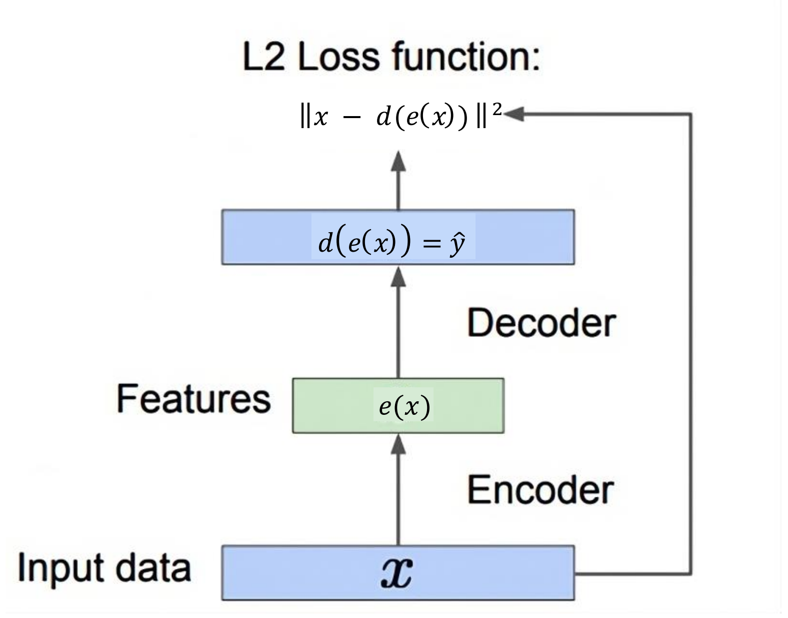


Valerio Velardo, "Autoencoders Explained Easily"

Q: Why dimensionality reduction?
A: Want features to capture meaningful factors of variation in data

How do we train Autoencoders?

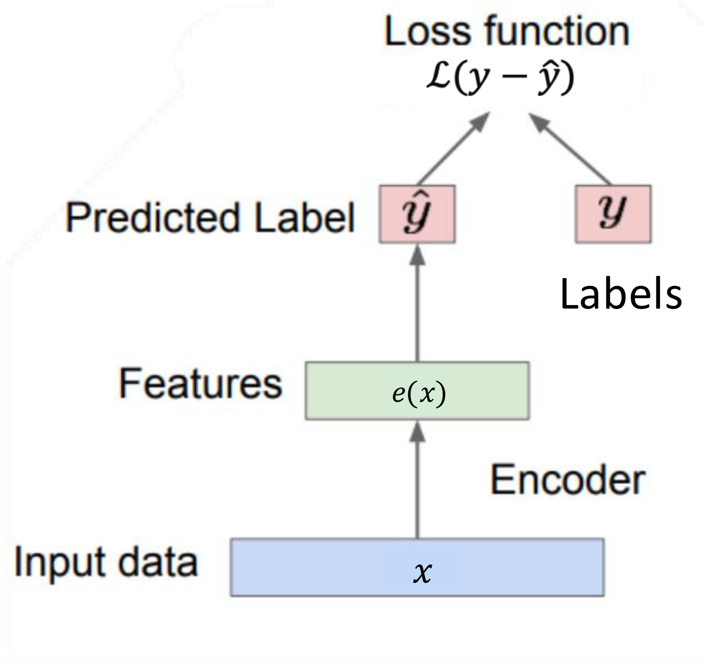
- **Data Preparation** — Data is collected and preprocessed. This might include normalizing the input values, reshaping the data into a format the autoencoder can process, and potentially removing noisy data points.
- **Architecture Design** — Decide on the autoencoder's architecture. This includes the **number of layers**, the **number of neurons** in each layer, the type of layers (**dense**, **convolutional**), and the type of autoencoder (**vanilla**, **denoising**, **variational**).
- **Choice of Loss Function** — Choose an appropriate loss function to measure the difference between the input and the output. Common choices include mean squared error (**MSE**) for **continuous** input data or binary cross-entropy for **binary input data**.
- **Optimizer Selection** — Choose an optimization algorithm to update the network's weights. This could be stochastic gradient descent (SGD), Adam, RMSprop, etc. (discussed later!)



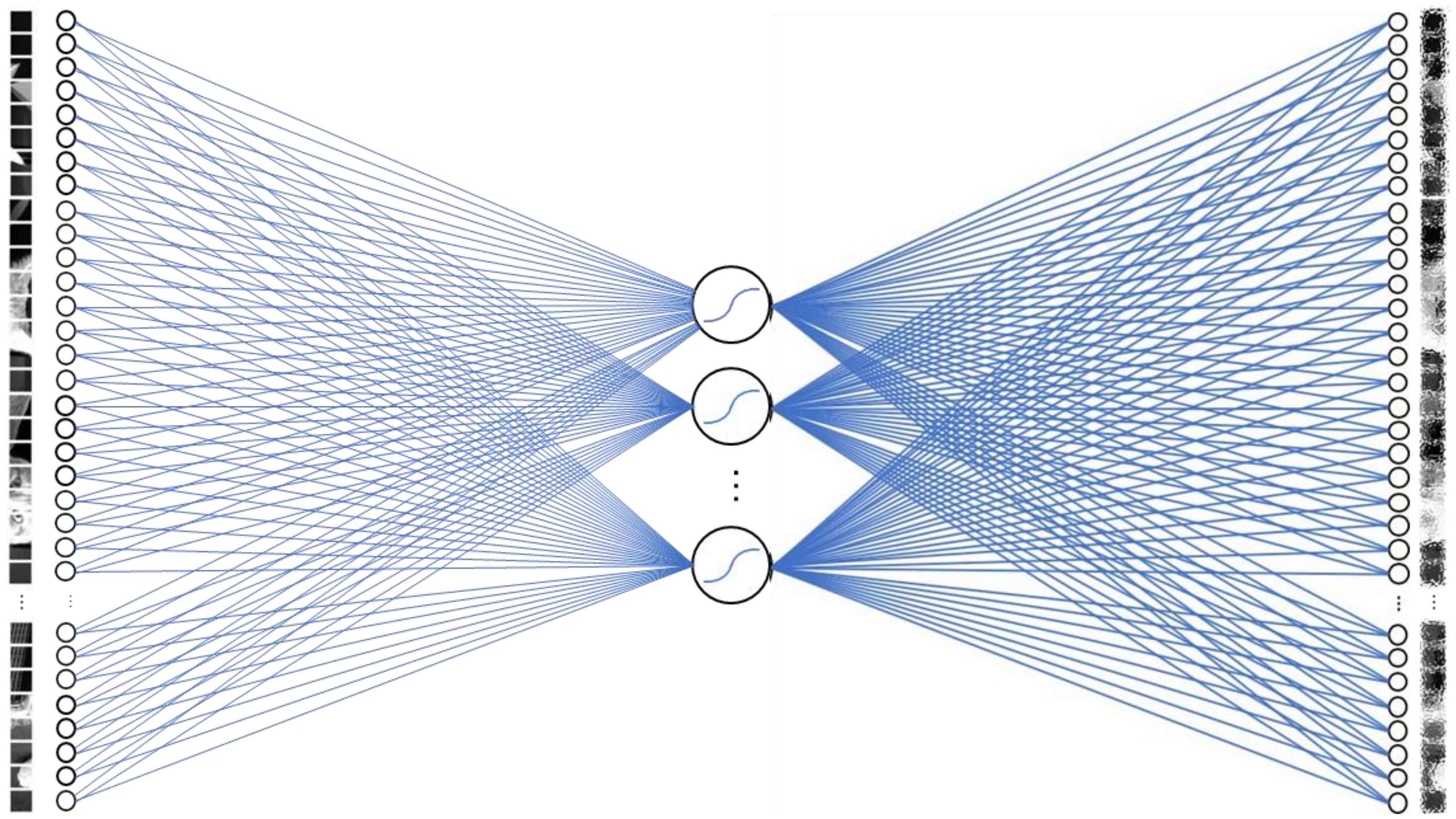
- Train such that features can be used to reconstruct original data

How do we train Autoencoders? (cont.)

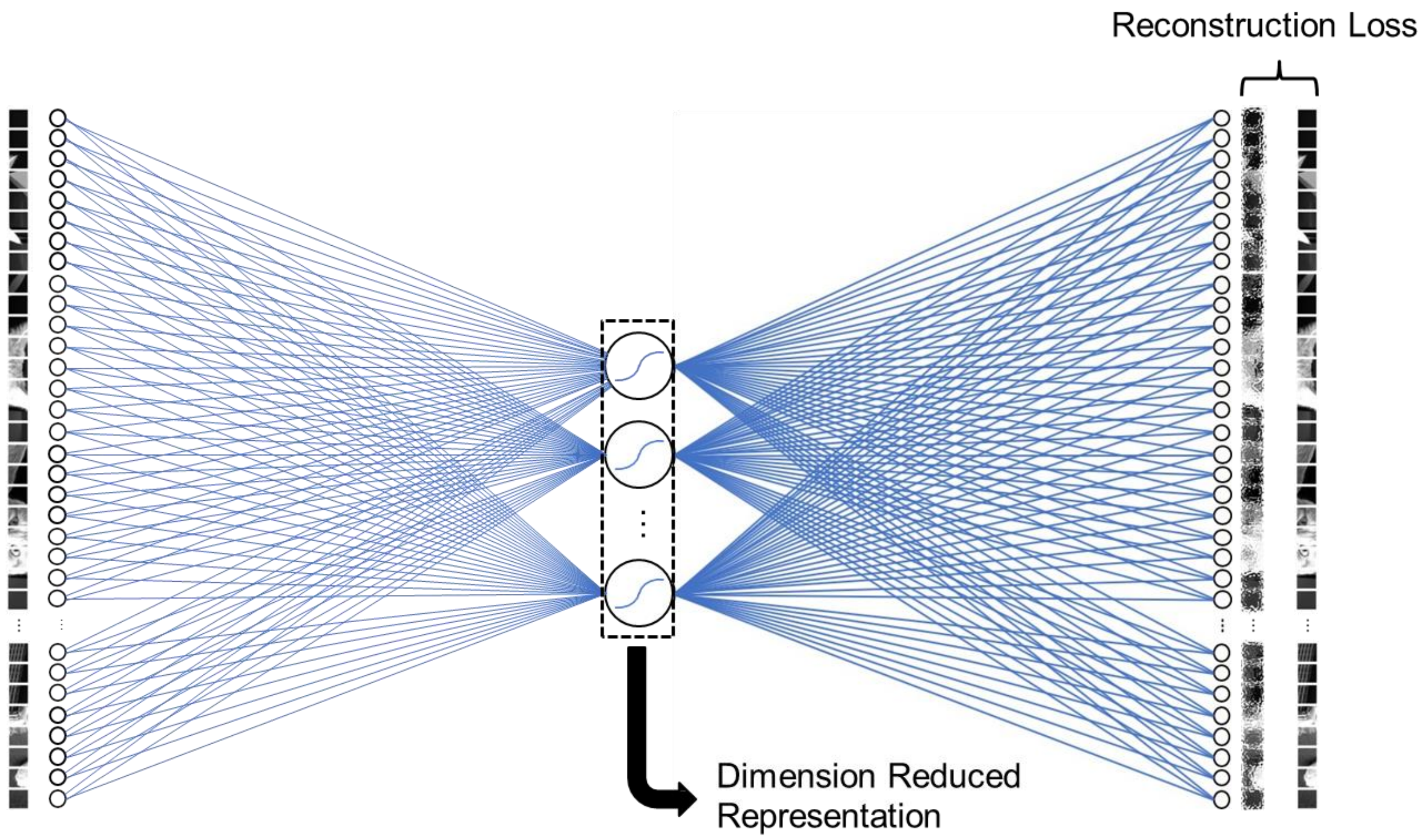
- **Encoding and Decoding Process —** Initialize the encoding and decoding process.
- **Training —** During training, the autoencoder uses backpropagation to update the weights. The goal is to minimize the reconstruction loss (the difference between the original and reconstructed inputs).
- **Validation —** Use a separate validation dataset not seen by the model during training to tune **hyperparameters** and prevent **overfitting**. This helps ensure that the autoencoder generalizes well to unseen data.
- **Evaluation —** After training, evaluate the autoencoder's performance. You might evaluate how well tasks like **clustering** or **classification** perform using the encoded data.
- **Fine-tuning**—Based on the evaluation, you may need to adjust the model architecture, retrain with different hyperparameters, or collect more data.



Autoencoder - Image Classification Example



Autoencoder - Image Classification Example



Common Types of Autoencoders

- Autoencoders come in various types, each designed for specific tasks or to handle different types of data. Here are some common types:
  - **Vanilla Autoencoder**: This is the simplest form of an autoencoder.
  - **Multilayer Autoencoder**: This type extends the vanilla autoencoder by adding more hidden layers, which can help learn more complex codes.
  - **Contractive Autoencoder**: These autoencoders add a **regularization term** to the loss function, encouraging the model to learn a robust representation of the data that is less sensitive to slight variations in input data.

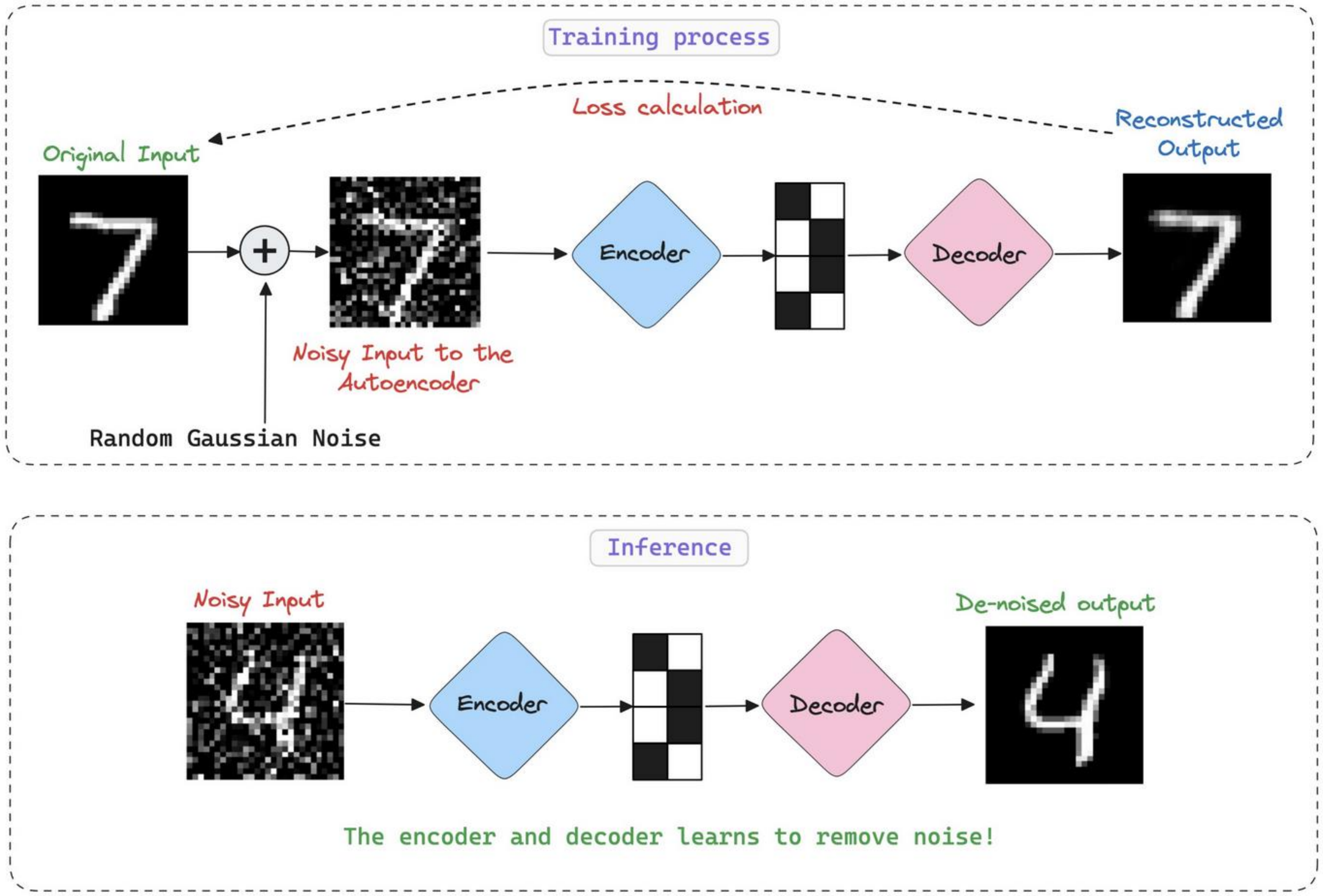
Denoising Autoencoder: This type of autoencoder is trained to remove noise from the input data. It takes a noisy input and learns to output the clean version of the data.

Convolutional Autoencoder: These are designed to handle image data efficiently. They use convolutional layers in the encoder to capture the spatial hierarchy of pixels and deconvolutional layers in the decoder to reconstruct the images.

Variational Autoencoder (VAE): VAEs are generative models that learn the parameters of a probability distribution representing the data. They are used to generate new instances similar to the input data.

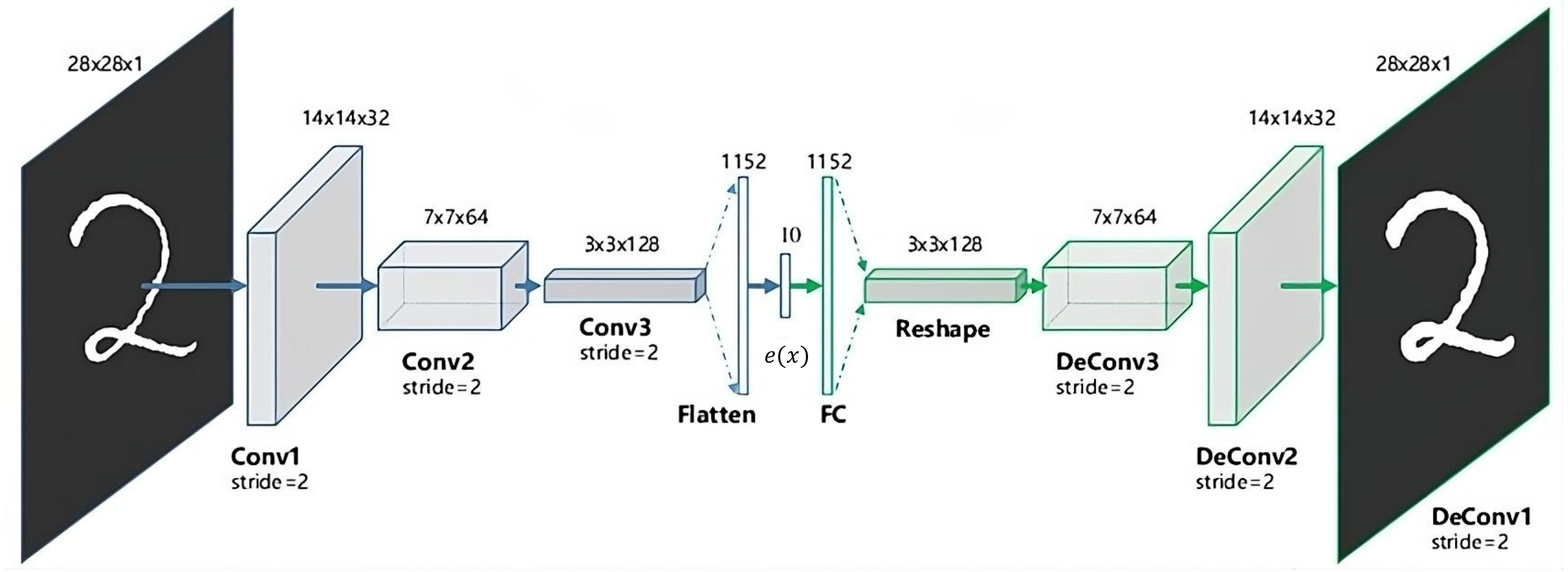
Denoising Autoencoder

Tutorial By: [Taegyun Jeontgjeon]
CEO @ SI Analytics



Source: https://twitter.com/akshay_pachaar/status/1746572162323099734

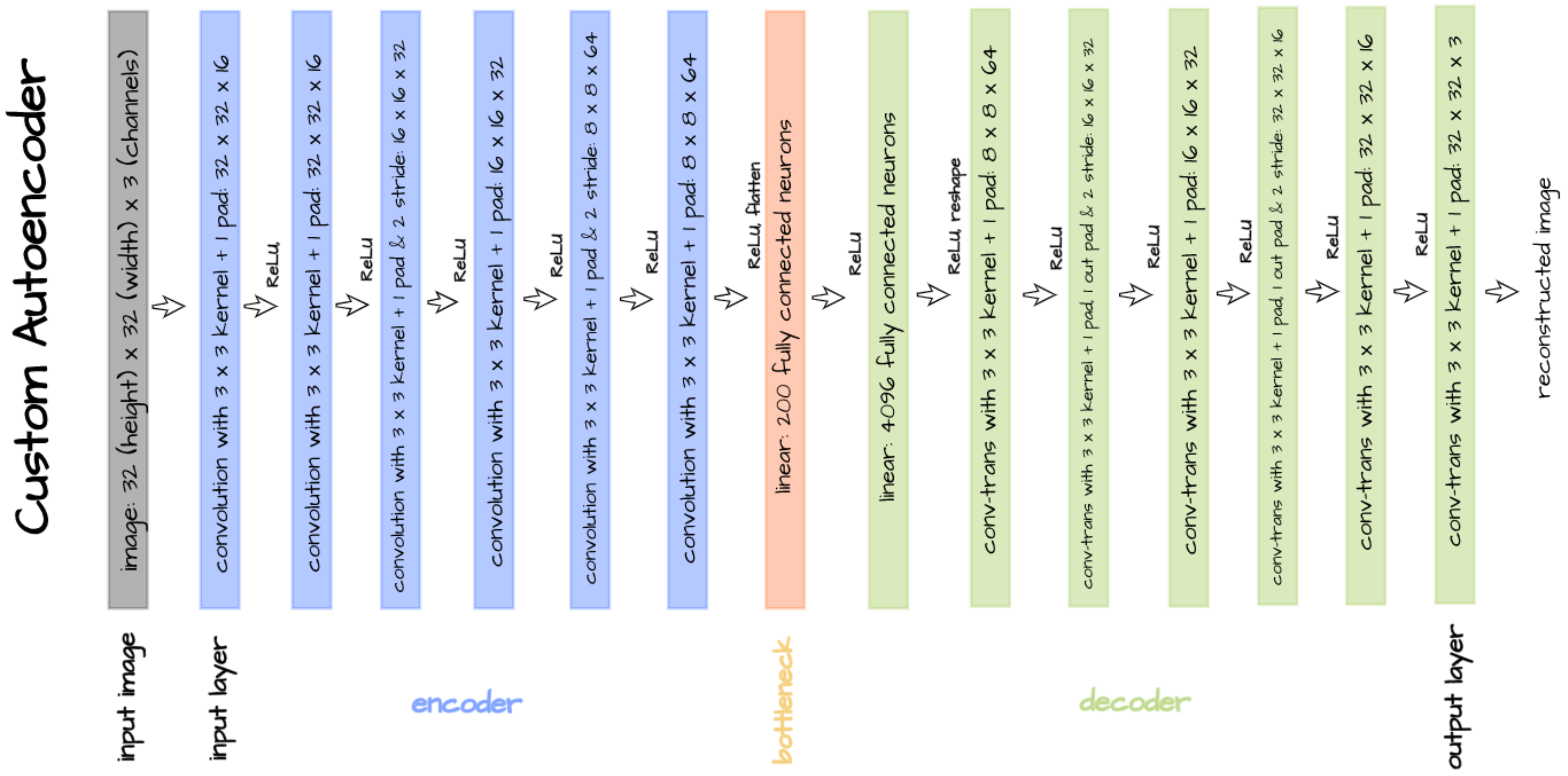
Convolutional Autoencoder



- Deconvolution (DeConv) is the process of reversing convolution when multiple filters are applied. It can be complex, as it involves reversing the effect of convolution and sequentially undoing the effects of all applied filters.

Guo, Xifeng & Liu, Xinwang & Zhu, En & Yin, Jianping. (2017). Deep Clustering with Convolutional Autoencoders. 373-382. 10.1007/978-3-319-70096-0_39.

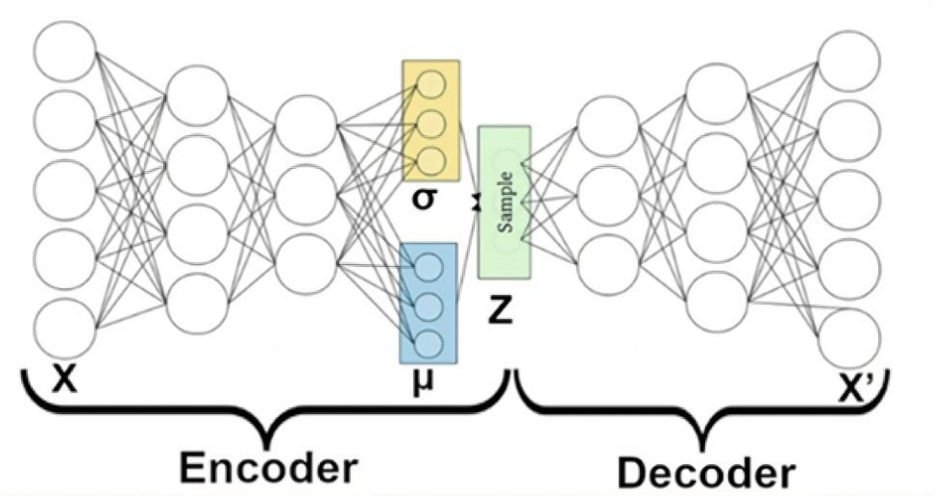
Deeper Autoencoder


Custom Autoencoder
image: 32 (height) x 32 (width) x 3 (channels)
convolution with 3 x 3 kernel + 1 pad: 32 x 32 x 16
ReLu
convolution with 3 x 3 kernel + 1 pad: 32 x 32 x 16
ReLu
convolution with 3 x 3 kernel + 1 pad & 2 stride: 16 x 16 x 32
ReLu
convolution with 3 x 3 kernel + 1 pad: 16 x 16 x 32
ReLu
convolution with 3 x 3 kernel + 1 pad & 2 stride: 8 x 8 x 64
ReLu
convolution with 3 x 3 kernel + 1 pad: 8 x 8 x 64
ReLu, flatten
linear: 200 fully connected neurons
ReLu
linear: 4096 fully connected neurons
ReLu, reshape
conv-trans with 3 x 3 kernel + 1 pad: 8 x 8 x 64
ReLu
conv-trans with 3 x 3 kernel + 1 pad, 1 out pad & 2 stride: 16 x 16 x 32
ReLu
conv-trans with 3 x 3 kernel + 1 pad: 16 x 16 x 32
ReLu
conv-trans with 3 x 3 kernel + 1 pad, 1 out pad & 2 stride: 32 x 32 x 16
ReLu
conv-trans with 3 x 3 kernel + 1 pad: 32 x 32 x 16
ReLu
conv-trans with 3 x 3 kernel + 1 pad: 32 x 32 x 3
reconstructed image
input image
input layer
encoder
bottleneck
decoder
output layer

Source: https://blog.paperspace.com/convolutional-autoencoder

Variational Autoencoder (VAE)

- Instead of directly converting input data to specific points in the latent space, VAE converts them to parameters that define a **probability distribution**.
- This approach dictates where a data point will likely be positioned in the latent space based on its characteristics.
- The VAE encoder outputs a probability distribution for **each latent attribute**.
- So, How does this help?
  - The VAE learns to rebuild not just from specific encoded points, but also from their surrounding space, enabling it to create new data by sampling from regions in that space rather than just replicating existing data tied to fixed points.



Source: Chouikhi, F., Abbes, A.B., Farah, I.R. (2023). Desertification Detection in Satellite Images Using Siamese Variational Autoencoder with Transfer Learning. In: Nguyen, N.T., et al. Computational Collective Intelligence. ICCCI 2023. Lecture Notes in Computer Science(), vol 14162. Springer, Cham.

$$\boxed{\begin{aligned} Z &= \mu + \sigma \odot \epsilon \\ \epsilon &\sim \mathcal{N}(0,1) \end{aligned}}$$

Utilizing the Power of Latent Space

- The latent space preserves the most crucial attributes of the input data, providing a condensed yet rich representation.
- Leveraging the Latent Space
  - **Generation** — The variational Autoencoder (VAE) can generate new, unique data samples by learning distribution parameters, enabling creativity and innovation.
  - **Denoising** — By training autoencoders to clean up intentionally corrupted data, they become adept at removing noise, resulting in clearer, more precise data.
  - **Anomaly Detection** — Autoencoders excel at identifying outliers by reconstructing normal data well but struggling with anomalies, making them invaluable in detecting unusual patterns or defects.
  - **Feature extraction** — By training an autoencoder to reconstruct the input data, it can learn to extract meaningful features from the data.

Wrapping up

- it's important not to make the autoencoder too capable.
- It might memorize the training data exactly if it becomes overly powerful, leading to overfitting. This means it will perform well on the training data but poorly on new, unseen data, which isn't our goal (**over-complete autoencoder**).
- On the other hand, an **under-complete autoencoder** doesn't have the capacity to replicate its inputs exactly at the output. This limitation makes the identification and learning of significant features, which prevents perfect reconstruction of the input data.

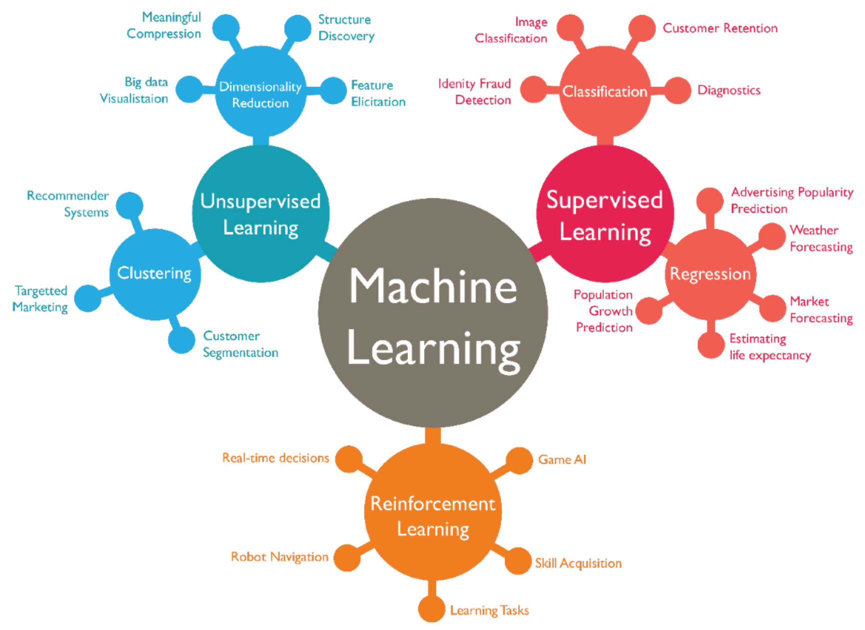
Attendance

You can use the provided link if you don't have a cell phone or if your phone lacks a QR-Code reader.

Hyperparameters Optimization (HPO)

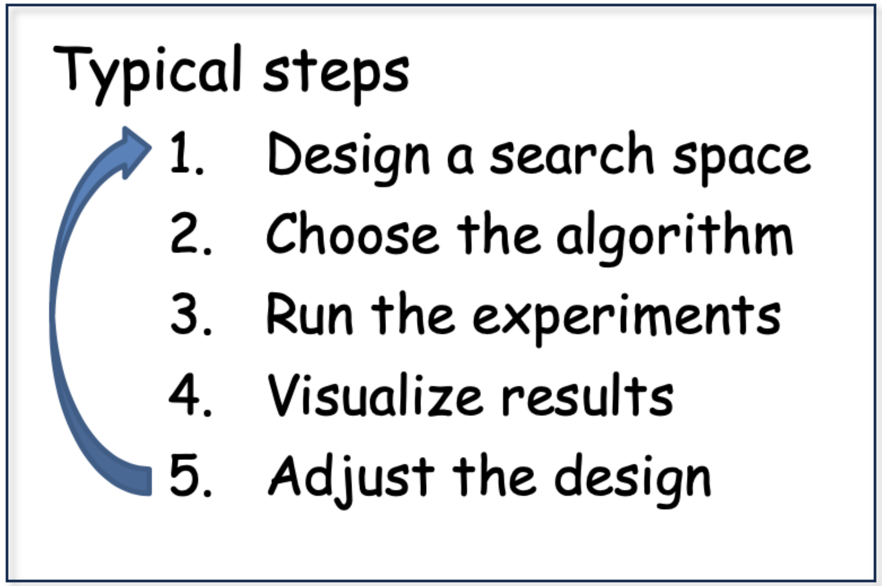
HPO Problem Statement

- Machines require precise settings, known as '**hyperparameters**', to optimize their learning capabilities.
- **Hyperparameter Optimization** (**HPO**) focuses on enhancing ML model performance through optimal hyperparameter tuning.
- It involves identifying:
  - Key hyperparameters significantly affect model outcomes;
  - selecting appropriate optimization methods for various model types;
  - Utilizing effective tools and libraries to facilitate the HPO process.
- HPO is essential for <u>developing a systematic approach</u> to hyperparameter(s) selection and tuning to achieve the highest model performance and efficiency.



Hyperparameter Tuning

- The initial hyperparameter settings often fail to achieve optimal performance, necessitating fine-tuning (Like adjusting a guitar to produce the ideal tone) ;
- Tuning hyperparameter(s) enables your ML models to operate at their peak efficiency, ensuring they deliver the most accurate results possible.
- **Experimenting** with various hyperparameter settings can enhance the performance of ML models, but manual adjustments are labor-intensive.
- To simplify this process, employing optimization techniques that autonomously seek the best hyperparameter(s) combination can significantly refine model outcomes, making the tuning process efficient and effective.

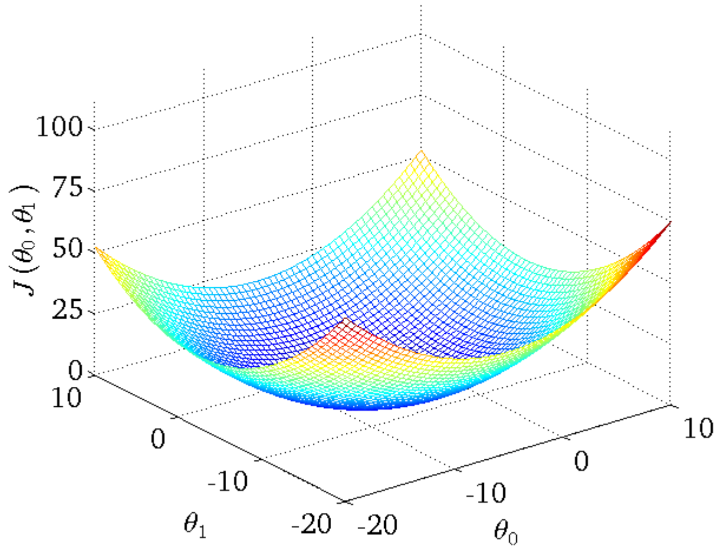


Typical steps

1. Design a search space
2. Choose the algorithm
3. Run the experiments
4. Visualize results
5. Adjust the design

Design Search Space

- In the context of HPO for ML models, the search space defines all the potential values each hyperparameter can take.
- Designing a search space involves *identifying* and *structuring* the range of possible values for each hyperparameter of a machine learning model.
- The goal is to navigate the search space to find the hyperparameter values that optimize the model's performance.



Minimize function over all possible θ_0 and θ_1

$$\min_{\theta_0,\theta_1} \sum_{j=1}^{m} \left[y^{(j)} - (\theta_0 + \theta_1 x^{(j)})\right]^2$$

Design Search Space (cont.)

Common considerations for designing an effective search space:

- **Identify Hyperparameters** — Identify key hyperparameters affecting ML model performance, such as *learning rate*, *number of layers* in neural networks, *number of trees* in random forests, and *regularization* parameters.
- **Categorize Hyperparameters —** Classify hyperparameters into types: *continuous* (e.g., learning rate), *discrete* (e.g., number of trees), *categorical* (e.g., activation functions), and *conditional* (dependent parameters, e.g., the type of regularization influences the regularization parameter to tune).
- **Prior Knowledge and Expertise** – Use domain knowledge and empirical evidence to constrain the search space effectively.
- **Use of Tools and Libraries**
- **Iterative Refinement**

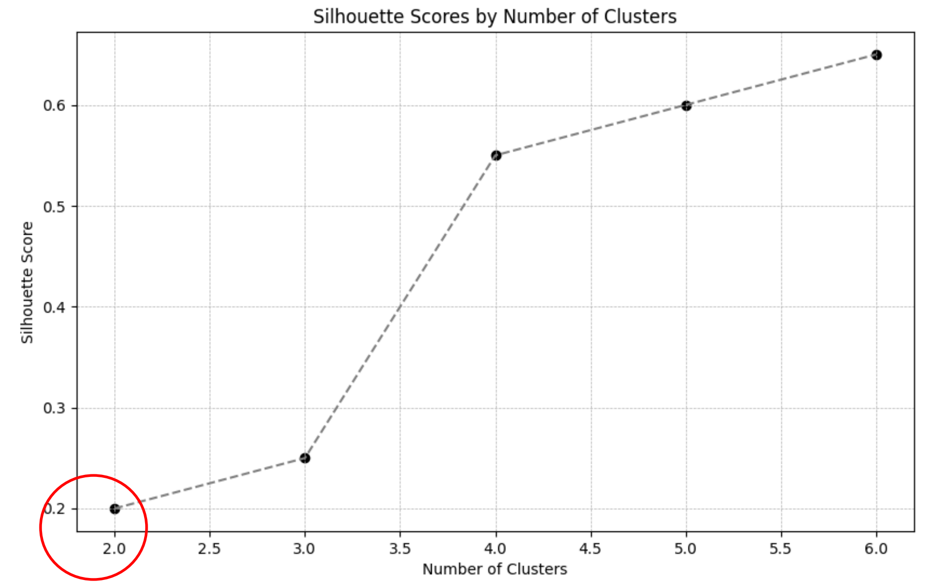
- **Define Ranges and Distributions**
  - Specify minimum and maximum values for continuous hyperparameters and choose appropriate distributions (uniform, etc.).
  - List all possible values for discrete hyperparameter(s).
  - Estimate available options for categorical hyperparameters.
  - Detail conditions under which conditional hyperparameters become relevant.

In-depth Search of the Search Space

- Several methods are used to in-depth search for the **Search Space**, including:
  - **Grid Search**: Evaluating a predefined range of hyperparameter values systematically. However, this can be computationally expensive for large search spaces.
  - **Random Search**: Randomly selecting hyperparameter values within the defined search space is often more efficient than grid search for large search spaces.
  - **Hyperband and Successive Halving**: Allocating a budget to different configurations and iteratively refining the search based on performance.
  - **Gradient-based Optimization**: Use gradients to guide the search towards optimal parameters for differentiable models.
  - **Evolutionary Algorithms**: Using strategies inspired by biological evolution, such as mutation, crossover, and selection, to explore the search space.
  - More …,

Grid Search (GS)

- An extensive search method that evaluates all combinations of hyperparameters in the given search space
- Easily parallelizable
- Drawbacks:
  - Computational costly and inefficient, especially for a large number of hyperparameters (the number of configurations to try exponentially increases)
  - Some evaluations are unnecessary



Example:

- Using the K-Means algorithm to cluster customers by shopping behavior, we focus on # of clusters for optimization.
- Discovering low silhouette scores with 2 or 3 clusters suggests poor quality, making further tests with k=2 or k=3 redundant as they likely won't yield the best solution

Random Search (RS)

- Unlike grid search, which systematically goes through all the possible combinations of hyperparameters, random search selects combinations randomly.
- Faster than GS
- Easily Parallelizable
- Drawbacks:
  - Still many unnecessary evaluations
  - Results are not guaranteed, especially when the number of evaluations is small.
  - hyperparameters are treated independently (their correlations are not considered)

Surrogate and Bandit Algorithms

- Sequential Model-Based Optimization (SMBO), such as Bayesian optimization (BO), determines the future *evaluation of HP configurations based on the previous results*.
- BO is a **sequential** method and is hard to parallelize (semi-parallelizable)
- BO is more efficient than GS & RS because it avoids many unnecessary evaluations

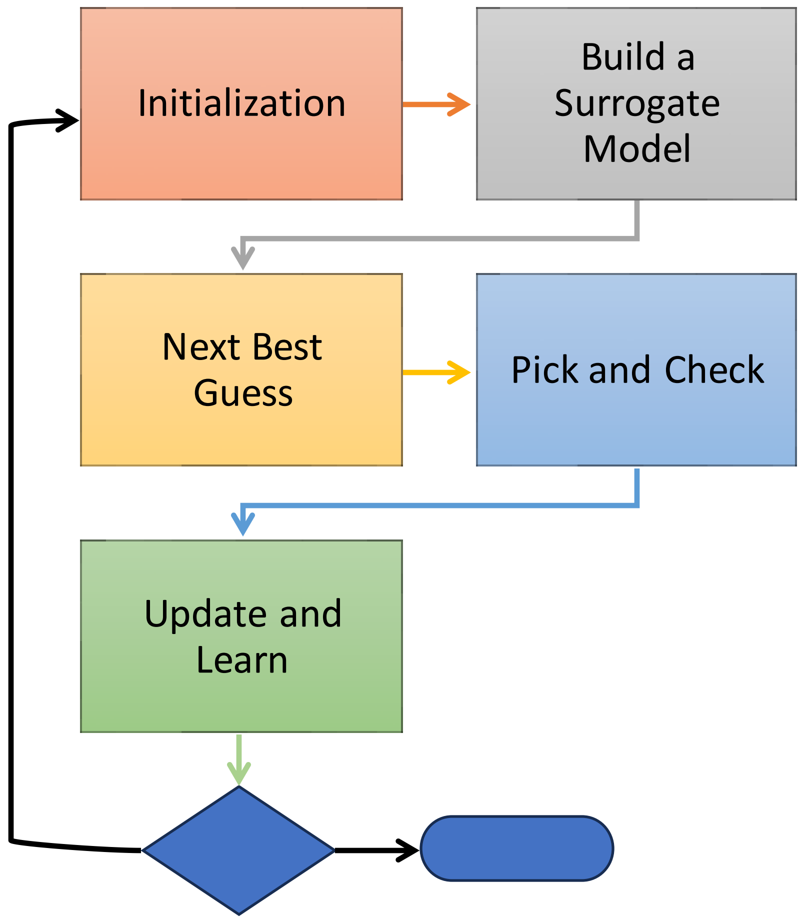
- Bandit Algorithms were developed specifically for hyperparameter tuning (like Hyperband).
- These methods introduce a form of early stopping for bad runs and continue better trials for longer periods of time.
- It can be parallelized.

Sequential Model-Based Optimization

- SMBO is like a treasure-hunting game where you make smart guesses to find the treasure quickly without digging up the whole sandbox. It helps people solve tricky problems without wasting time or resources.
- Sequential Model-Based Optimization (SMBO) is a strategy for optimizing expensive-to-evaluate functions. It's particularly useful in scenarios where the **function evaluations** take <u>significant time or resources</u>, such as hyperparameter tuning in ML models.
- Advantages of SMBO:
  - **Efficiency** – reduce the number of expensive evaluations
  - **Flexibility** – applied to a wide variety of optimization problems
  - **Effectiveness** – often finds better solutions with fewer evaluations

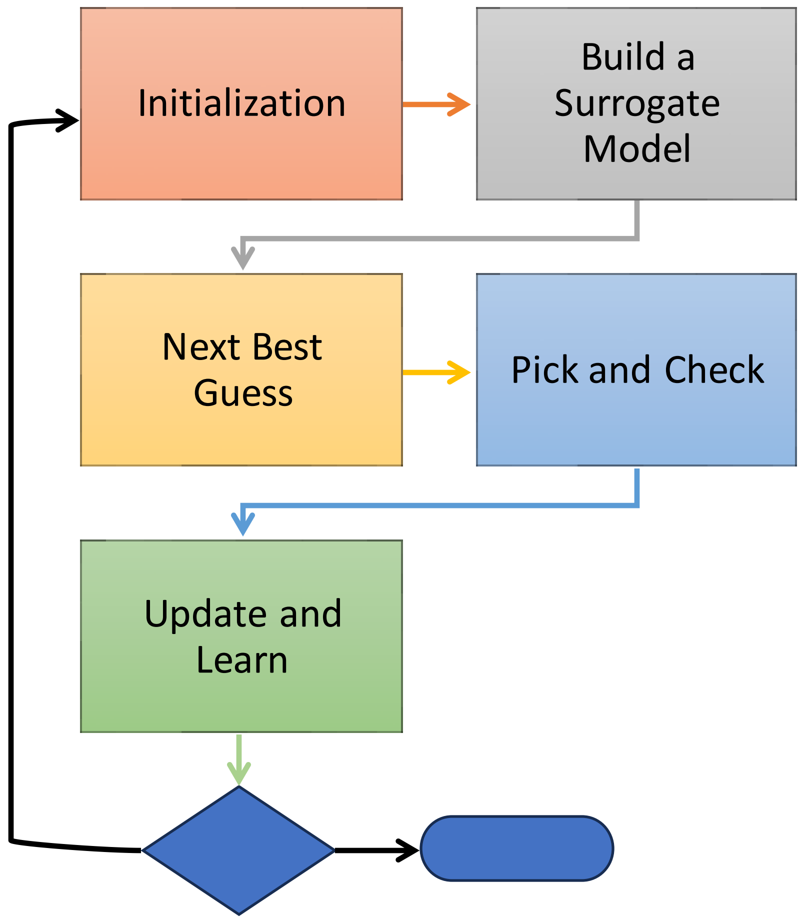
Sequential Model-Based Optimization

- **Initialization**: Start by selecting *initial values* in the search space, either randomly or through a heuristic, and evaluate the expensive objective function at these values.
- **Build a Surrogate Model**: Create a **simpler model** that mimics the expensive objective function's behavior. This model is called a surrogate and can be a **Random Forest** or **Gradient Boosting** that roughly approximates the objective function.
- **Next Best Guess**: Now, use a special function (called an **acquisition** function) to pick the next spot to try out. This function, like Expected Improvement (EI), helps decide which new spot might offer us the best improvement over what we've found so far.

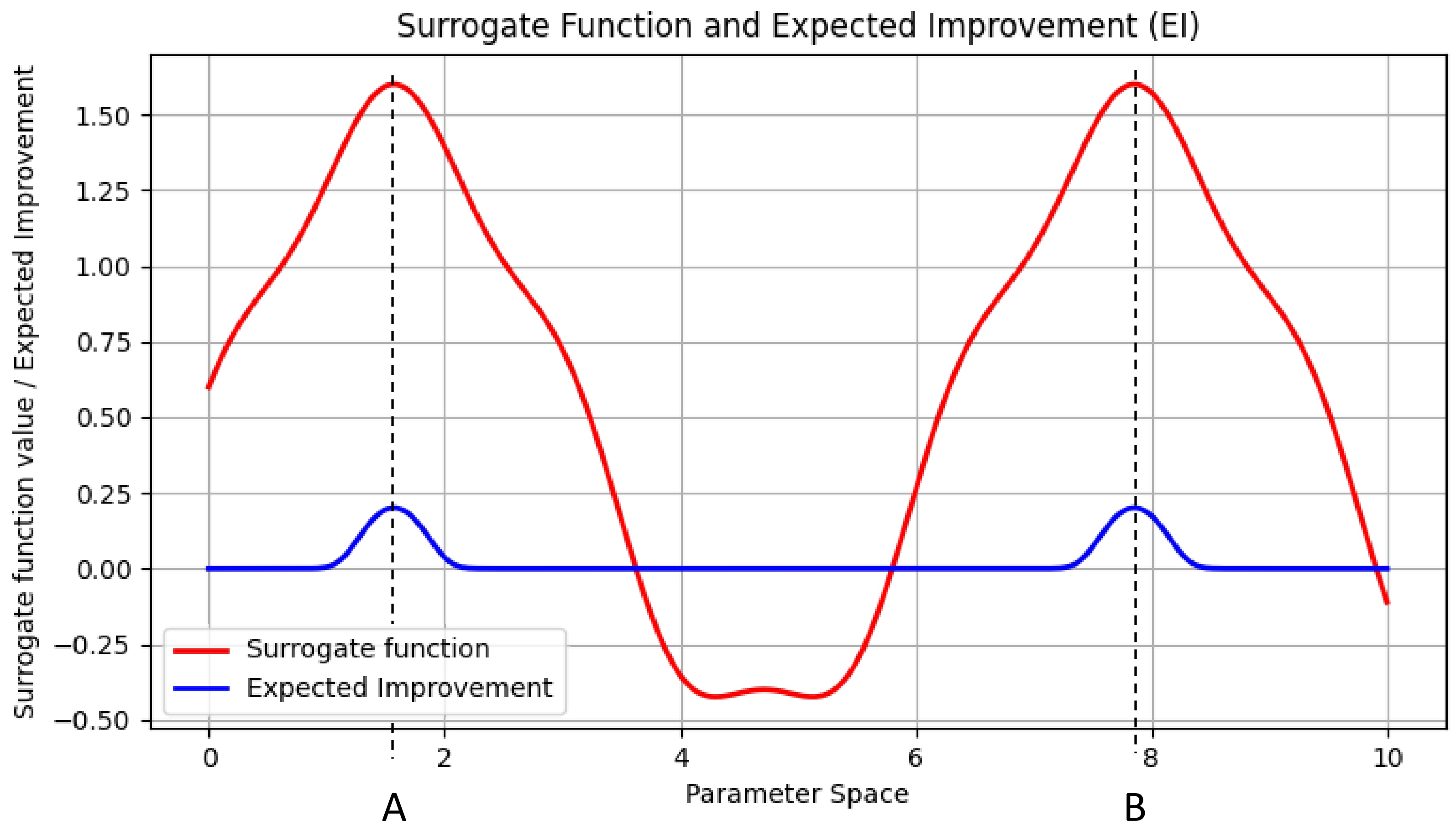


Sequential Model-Based Optimization

- **Pick and Check**: Once we have our next best guess, we return to our expensive function to see how good this new spot is.
- **Update and Learn**: Update the surrogate model with new information from the acquisition function.
- **Loop Back**: Repeat the process of guessing, checking, and updating until we're satisfied with the results or hit a limit, like a certain number of tries or a specific goal that can be reached.
- Want to read more: https://arxiv.org/abs/1402.0796



Expected Improvement (EI) Acquisition Function



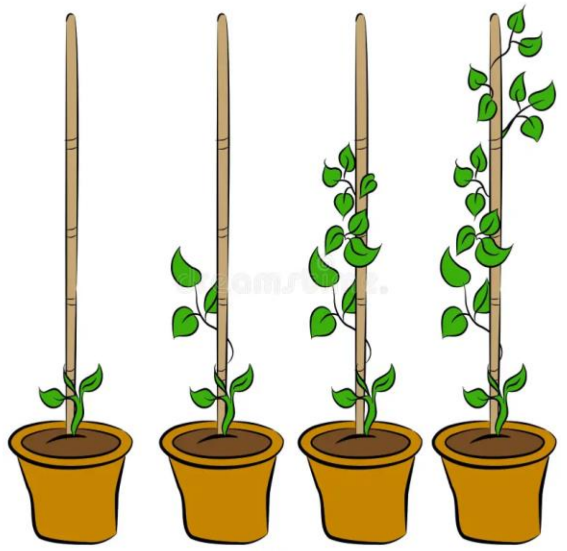
- This surrogate function represents the mean predicted accuracy of an ML model across different hyperparameter values.
- EI estimates the expected improvement over the current best at each point, factoring in prediction and uncertainty.
- Higher EI points are preferred for evaluation, suggesting a mix of potential gain and exploration.

Bayesian Optimization

- Bayesian optimization is an example of Sequential Model-Based Optimization (SMBO), which is tailored for the efficient optimization of black-box functions that are expensive to evaluate.
- Bayesian optimization typically employs a **Gaussian Process (GP)** as its surrogate model, which captures both the expected value and the uncertainty of the objective function's predictions.
- The GP is updated after each evaluation, progressively improving its accuracy.

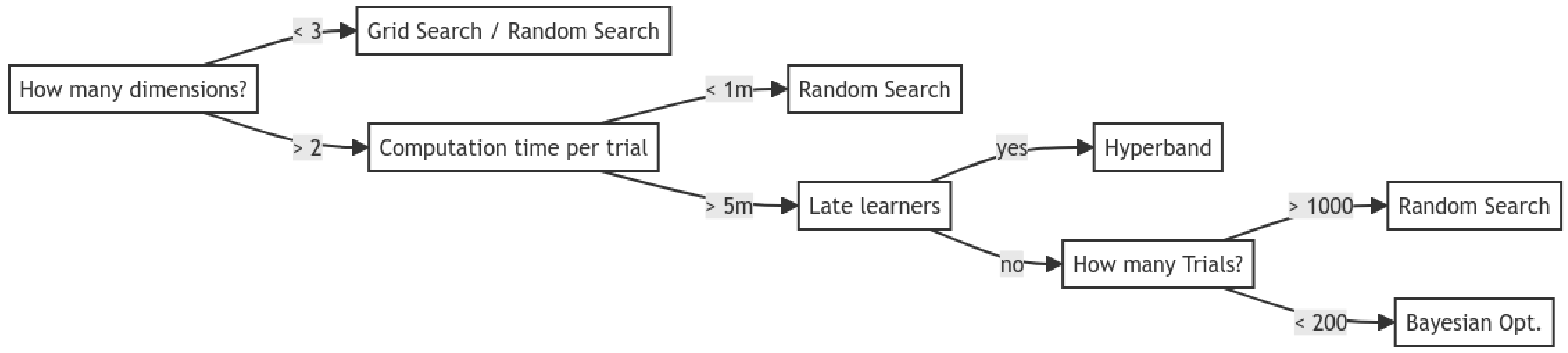
Bandit Algorithms

- Bandit algorithms are like smart gardeners. They first give each plant a little bit of water. Then, as they watch the plants grow, they notice some are growing faster than others. Instead of watering all the plants the same, they give more water to the fastest-growing plants and less to the others. Sometimes, they stop watering the slowest ones altogether.
- Bandit algorithms, like **Hyperband**, were specifically designed for hyperparameter tuning. They incorporate an efficient twist: they cut off poor-performing model configurations early and allocate more resources to the promising ones.
- This dynamic allocation accelerates the tuning process and makes parallelization highly flexible.



Source: https://www.dreamstime.com/illustration/growing-plant-clipart.html

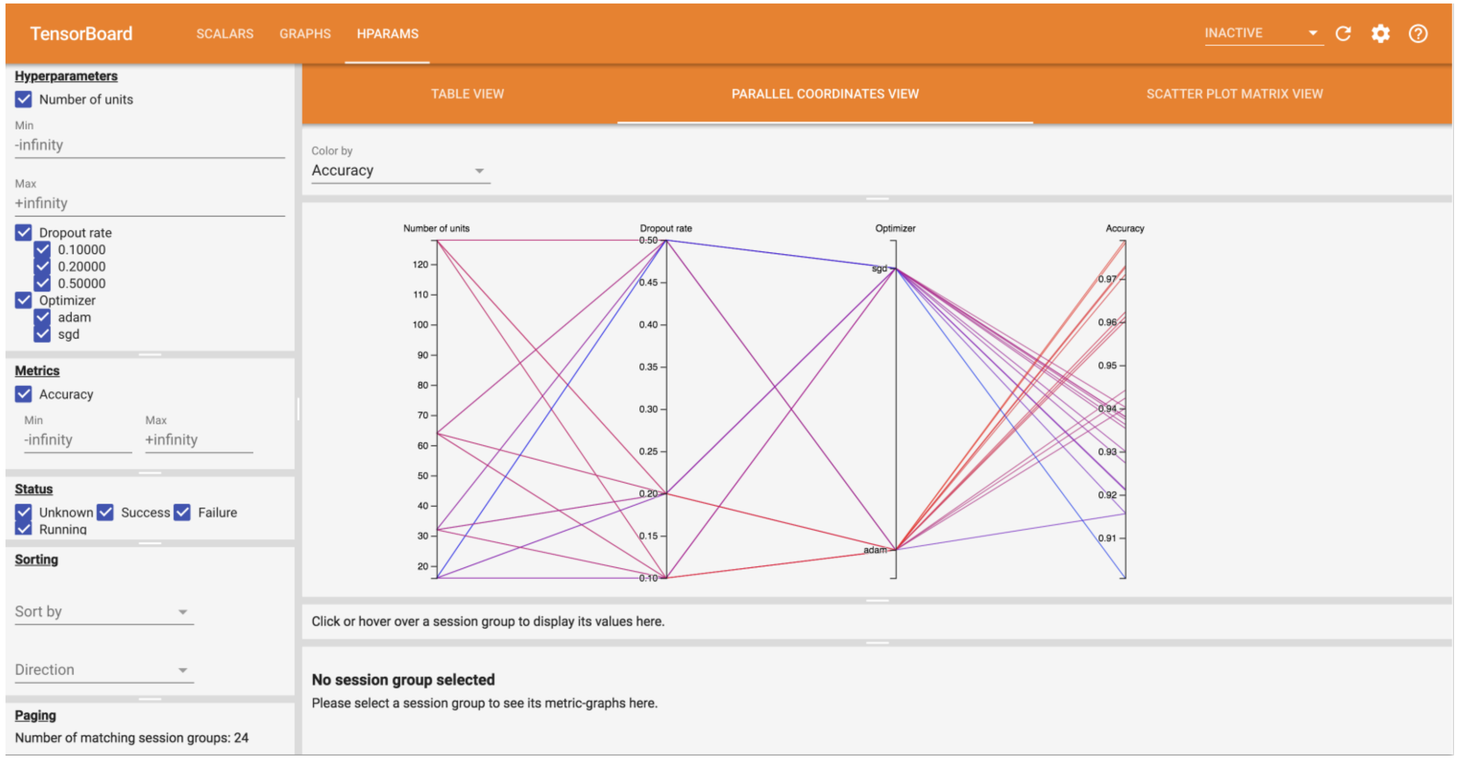
Choosing the HPO Technique



HPO Method	Pros	Cons
Grid search (GS)	Simple	Time-consuming
Random search (RS)	• Faster than GS • Enable parallel execution	• Not considering previous results
Bayesian optimization (BO)	• Fast convergence for continuous HPs	• Poor capacity for parallel execution
Bandit algorithms	• Fast because it evaluates on subsets. • Enable parallel execution.	• Require representative subsets.

Use Visualization Tools to Guide Your Tuning

- Use plots to understand your model.
- ***TensorBoard*** is a popular open-source visualization suite from Google. It is commonly used with TensorFlow and offers many tools for visualizing model performance, hyperparameters, and training data.
- **Weights & Biases (W&B)** is a cloud-based platform specifically designed for experiment tracking and visualization in machine learning.
- **Cave** is an open-source visualization tool built specifically for ML.
- **Neptune** is another cloud-based platform for experiment tracking and visualization in ML. It offers visualizations of model performance and hyperparameters.



Hyperparameters
✓ Number of units
Min -infinity
Max +infinity
✓ Dropout rate
✓ 0.10000
✓ 0.20000
✓ 0.50000
✓ Optimizer
✓ adam
✓ sgd

Metrics
✓ Accuracy
Min -infinity Max +infinity

Status
✓ Unknown ✓ Success ✓ Failure
✓ Running

Sorting
Sort by
Direction

Paging
Number of matching session groups: 24

Color by: Accuracy

Click or hover over a session group to display its values here.

No session group selected
Please select a session group to see its metric-graphs here.

The TensorBoard's Hyperparameters (HParams) dashboard

HPO Tools and Libraries

- Hyperparameter Optimization (HPO) tools and libraries are designed to help you select the best hyperparameters for an ML model.
- Different tools have different strengths, and many support multiple methods of **searching the hyperparameter space**.
- In the table, a checkmark (✓) indicates that the tool supports the corresponding method.
- BO-GP stands for Bayesian Optimization using Gaussian Processes, and PBT stands for Population-Based Training.

Here's a listing of HPO tools and the methods they support:

HPO Tool	**Grid Search (GS)**	**Random Search (RS)**	**Bayesian Optimization - Tree-based**	**Hyperband**	**BO-GP**
Sklearn	✓	✓			
Hyperband				✓	
Skopt		✓			✓
Sherpa	✓	✓			✓
BayesOpt					✓
Spearmint			✓		
Hyperopt		✓	✓		
Optunity	✓	✓	✓		
RayTune				✓	

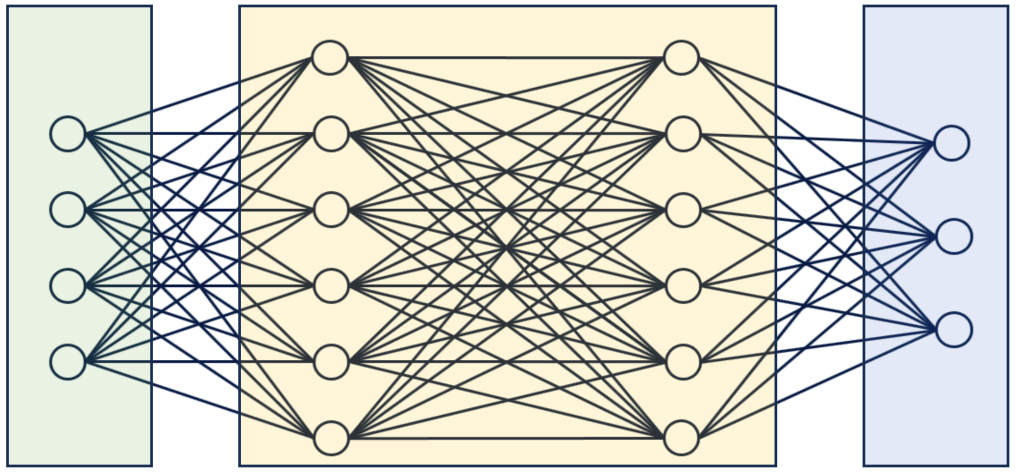
HPO Experiments

- The sample code for the HPO implementation is available at the following link: https://github.com/LiYangHart/Hyperparameter-Optimization-of-Machine-Learning-Algorithms
- Sample code for both regression & classification problems
- Datasets used: Boston-Housing & MNIST
- ML algorithms, including RF, SVM, KNN, ANN
- HPO methods, including GS, RS, hyperband, BO

Recurrent Neural Networks (RNN)

Recall: Feed-forward Neural Network

- A feed-forward neural network consists of multilayer neural networks that feed signals forward into a neural network and pass through different layers of the network in the form of activations, concluding in predictions at the output layer.
- The following animation illustrates how a feed-forward neural network processes input signals, categorizing them into one of three classes in the output.



- Feed-forward networks are the **simplest** form of artificial neural networks and serve as the foundation for understanding more complex neural network architectures.
- Their straightforward architecture makes feed-forward networks easier to understand and implement than complex models.
- **Limitations** – They learn a static mapping from inputs to outputs and cannot adapt to changing input patterns over time

Image source: https://vitalflux.com/feed-forward-neural-network-python-example/

Sequence Analysis

- Numerous events and processes unfold over time, and understanding their patterns is crucial. This includes the rise and fall of stock market values, changes in weather, the structure of languages, and the rhythm of daily human activities.
- We use specialized neural network architectures to learn from such inherent temporal dynamics data to predict what might happen next in these sequences.
- Examples:
  - In recommendation systems, understanding the sequence of user actions allows for more personalized and relevant content suggestions.
  - Many Natural Language Processing (NLP) tasks, such as text generation, machine translation, and sentiment analysis, require understanding the sequence of words for accurate processing.

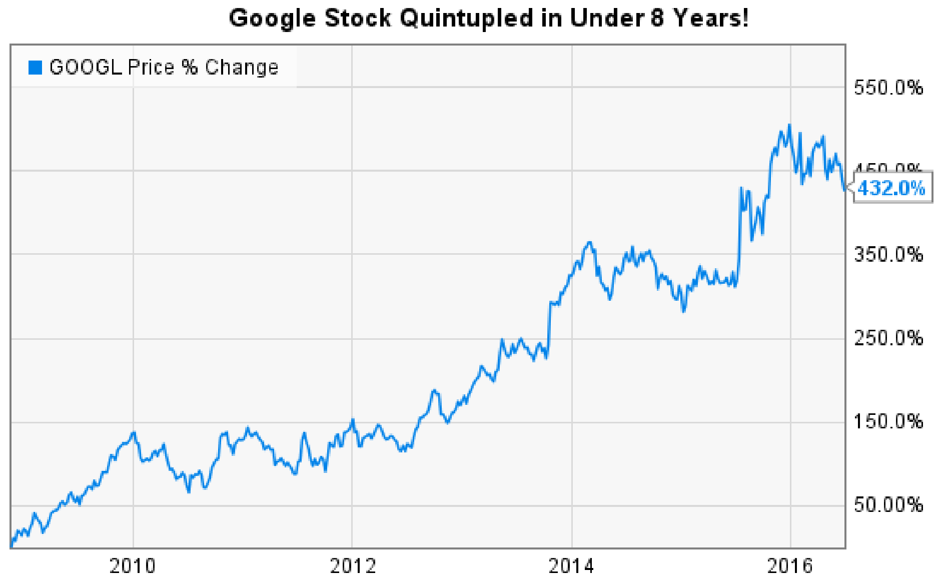
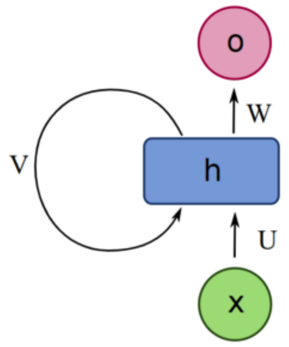


Image source: https://www.fool.com/investing/2016/06/30/google-stock-history-where-has-it-been-where-is-it.aspx

Sequence Analysis - Design Criteria

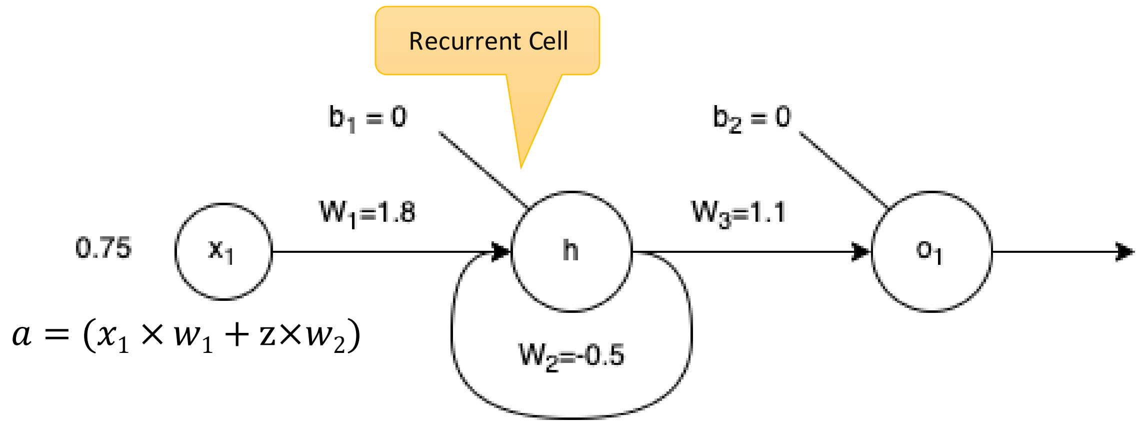
- How do we predict sequential data using a neural network?
- Needed criteria:
  - Handle **variable length sequences** (flexible NN regarding the sequence length is needed).
  - Track **long-term dependencies** and maintain information about the **order**
- To achieve this, we can use:
  - **Recurrent Neural Networks** (RNNs): At each step, they combine the new input with the previous hidden state to update their internal state, theoretically retaining information from the beginning of the sequence.
  - **Long Short-Term Memory** (LSTM) Units: These RNN types include mechanisms (**gates**) to add and remove information from the memory cell, making it easier to maintain long-term dependencies.
  - **Gated Recurrent Units** (GRUs): Similar to LSTMs but with fewer gates, GRUs effectively manage long-term dependencies but are computationally less intensive.



Recurrent Neural Networks (RNN)

- RNNs combine the new input with the previous hidden state to update their internal state, theoretically retaining information from the beginning of the sequence.
- RNNs differ from traditional feedforward neural networks because they have a **feedback loop**.

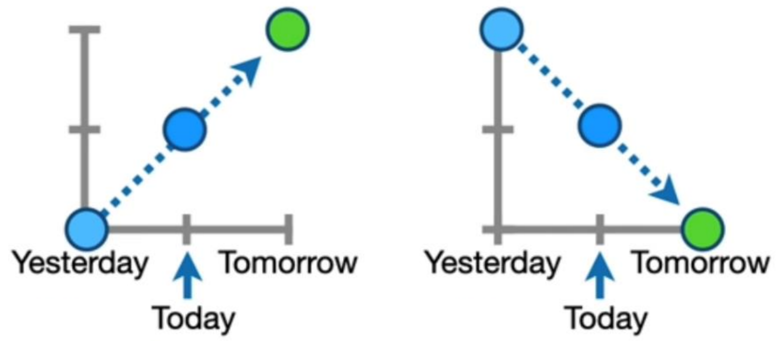
- **Input (X)**: This represents the RNN input, typically a **sequence of vectors**.
- **RNN Cell**: The recurrent cell has internal mechanisms allowing it to process one sequence element (one timestep) at a time and pass on information to the next.
- **Output (Y)**: This is the output from the RNN after processing the sequence.

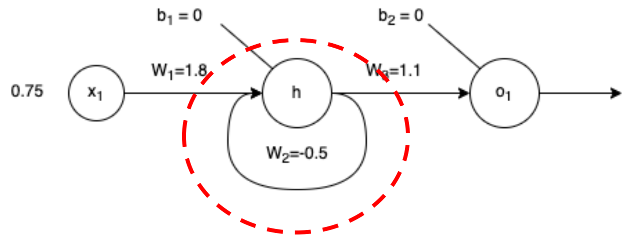


$a = (x_1 \times w_1 + \mathrm{z} \times w_2)$

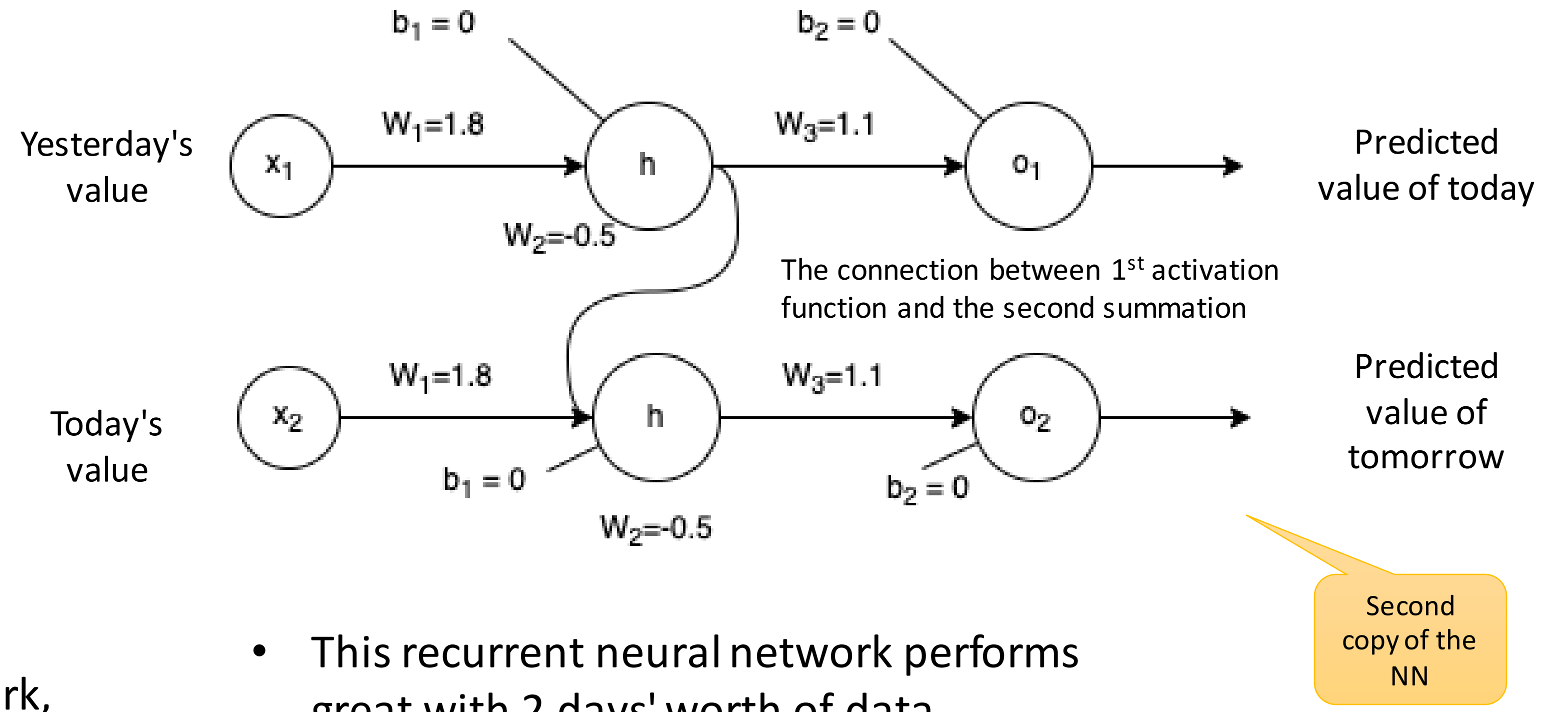
The RNN Cell

- Example: Stock Market





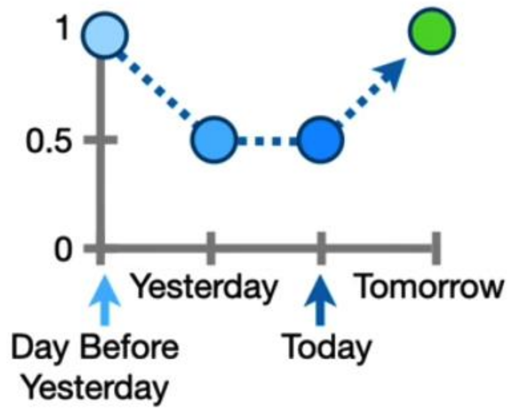
- By unrolling the recurrent neural network, we end up with a new network that has two inputs and two outputs.

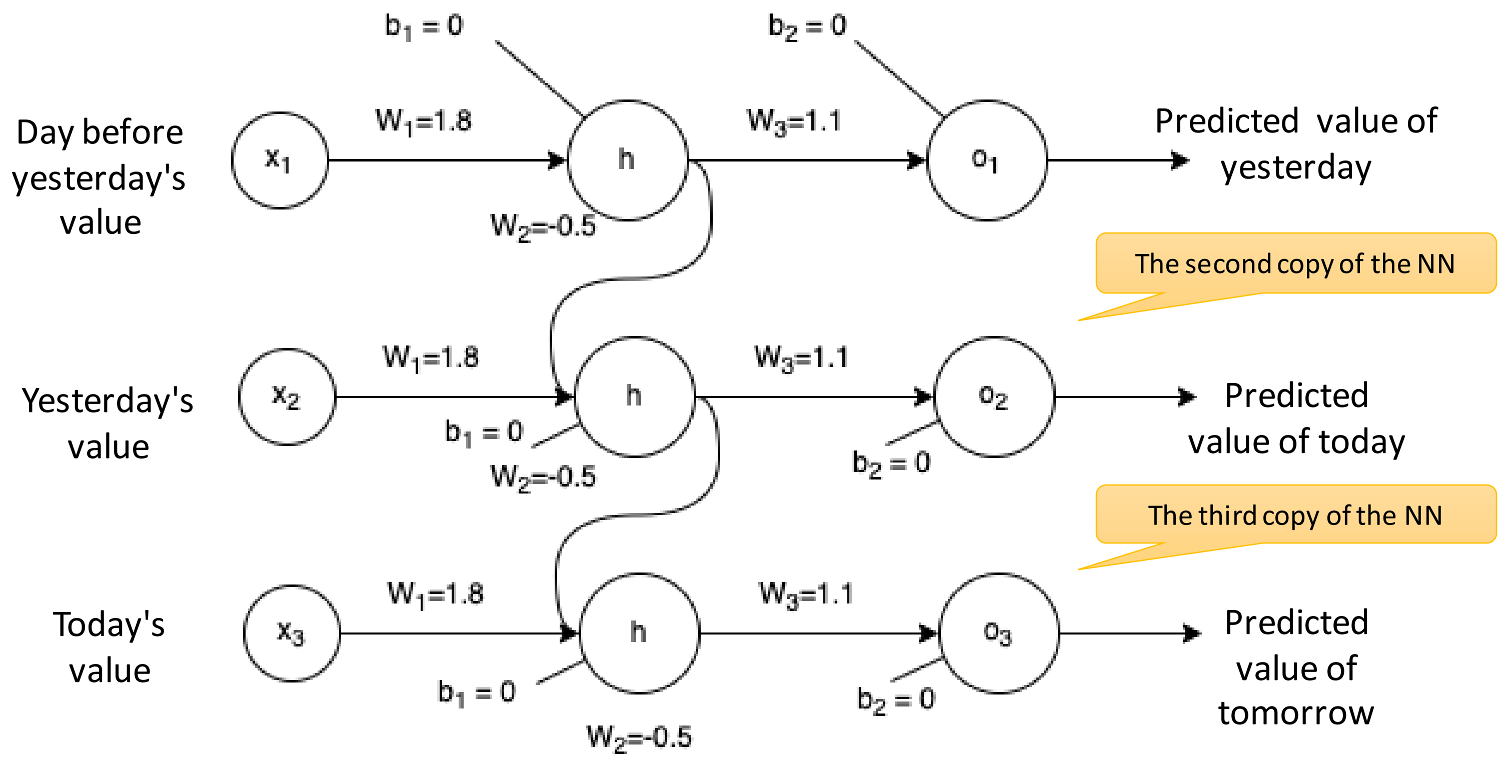


- This recurrent neural network performs great with 2 days' worth of data

The RNN Cell (cont.)

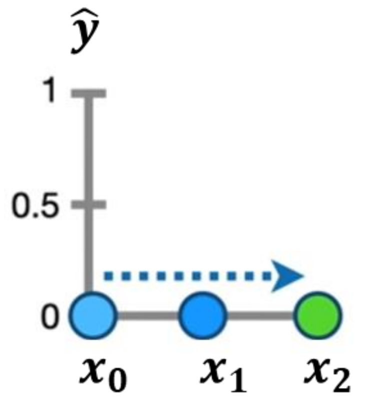
- This recurrent neural network performs great with 3 days' worth of data

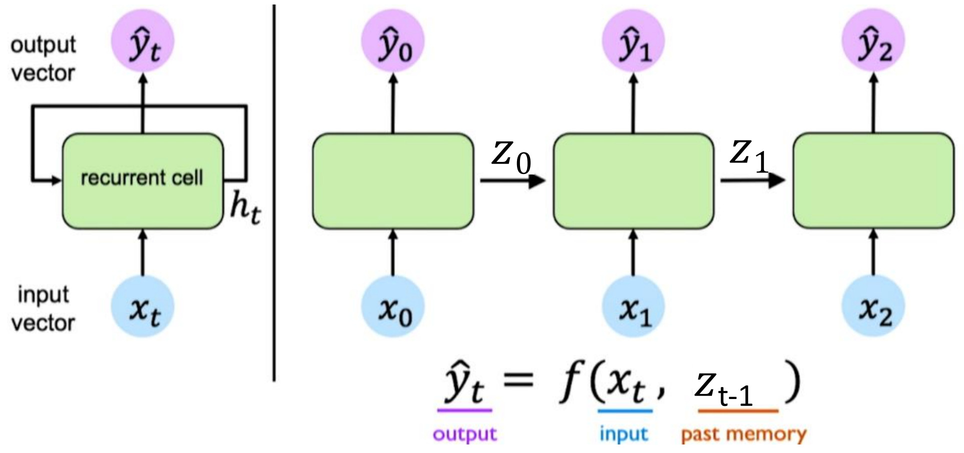




RNN Computational Graph

- For generalization, it's easier to visualize the enrollment process as a computational graph with many time steps.





$$\underbrace{\hat{y}_t}_{\text{output}} = f(\underbrace{x_t}_{\text{input}},\ \underbrace{z_{t\text{-}1}}_{\text{past memory}})$$

- We use the same set of weights for every time step of the computation.

Next

- RNN, LSTM

Thank you