Representation Learning

Representation learning is a set of techniques that **allow a system to automatically discover the representations needed for feature detection or classification** from raw data.

- A key feature of NNs is their ability to **learn representations** of the **data** $\phi(x)$;
- Standard linear models require **hand-crafted features**;
- **Representations** are useful for several reasons:
  - Can make models more **expressive and accurate**;
  - They may allow transferring representations from one task to another.

Hierarchical Compositionality

- Deep NNs learn **coarse-to-fine** representation layers;
- **Hierarchical compositionality** is the idea that **complex concepts** are composed of **simpler ones**;
- Layer closer to inputs learn **simple concepts** - edges, corners, etc.;
- Layer closer to outputs learn **more abstract representations** - shapes, forms, objects, etc.

But now some questions arise:

- How can a NN so effectively **represent** and **manipulate** knowledge, if it has only a few hidden units?
- What is each hidden unit actually **representing**?
- How can a NN **generalize** to objects that is has not seen before?

Distributed Representations

- **Local representations (one-hot)** - one dimension per object;
- **Distributed representations** - one dimension per property;
  - No single neuron encodes everything - **groups of neurons work together**;
  - **More compact and powerful**;
  - Hidden units should capture **diverse properties** of objects - not all capturing the same property - ensured by **random initialization**;
  - Initializing all units to the same weights would never break the symmetry;
  - Initializing hidden layers using **unsupervised learning** can help break the symmetry - force network to **represent latent structure** in the data; encourage hidden layers to encode **useful features**.

This can be done by using **auto-encoders**.

Auto-Encoders

Auto-encoders are feed-forward NNs trained to reproduce its input at its output layer.

- **Encoder** - maps input to a hidden representation : $h = g(Wx + b)$;
- **Decoder** - maps hidden representation to a reconstruction : $\hat{x} = W^T h(x) + c$;
- **Loss function** - $\mathcal{L}(\hat{x}, x) = \frac{1}{2}||\hat{x} - x||^2$;
- **Objective** - $\hat{W} = argmin_W \sum_i ||W^T g(Wx_i) - x_i||^2$.

Single Value Decomposition (SVD)

- **SVD** is a matrix factorization method that decomposes a matrix $A \in \mathbb{R}^{m \times n}$ with $m \geq n$ into the product of three matrices U, Σ and V such that $A = U \Sigma V^T$;
  - $U \in \mathbb{R}^{m \times m}$ - columns are an orthonormal basis of $R(A)$ (left singular vectors);
  - $\Sigma \in \mathbb{R}^{m \times n}$ - diagonal matrix with singular values of A;
  - $V \in \mathbb{R}^{n \times n}$ - columns are an orthonormal basis of $R(A^T)$ (right singular vectors);

- $sigma_1 \geq ... \geq \sigma_r$ - square roots of the eigenvalues of $A^T A$ or AA^T - **singular values** of A;
- $U^T U = I$ and $V^T V = I$.

Linear Auto-Encoder

- Let $X \in \mathbb{R}^{N \times D}$ be a data matrix with N samples and D features ($N > D$);
- Assume $W \in \mathbb{R}^{K \times D}$ ($K < D$);
- We want to minimize $\sum_{i=1}^{N} ||x_i - \hat{x}_i||_2^2 = ||X - XW^T W||_F^2$;
  - $|| \cdot ||_F^2$ - **Frobenius norm**;
  - $W^T W$ has rank K;
- From the **Eckart-Young theorem**, the minimizer is **truncated SVD** of X^T;
  - $\hat{X}^T = U_K \Sigma_K V_K^T$;
  - $W = U_K^T$;
- This is called **Principal Component Analysis (PCA)** - fits a **linear manifold** to the data.
- By using **non-linear activations**, we obtain more sophisticated codes (representations).

There are some variants of auto-encoders:

- **Sparse auto-encoders** - add a **sparsity penalty** $\Omega(h)$ to the loss function;
  - Typically the number of hidden units is larger than the number of inputs;
  - The sparsity penalty is a **regularization** term that encourages the hidden units to be **sparse**;
- **Stochastic auto-encoders** - encoder and decoder are **not deterministic**, but involve some **noise/randomness**;
  - Uses distribution $p_encoder(h|x)$ for the encoder and $p_decoder(x|h)$ for the decoder;
  - The auto-encoder can be trained to minimize $-log(p_decoder(x|h))$;
- **Denoising auto-encoders** - use a **perturbed version of the input** $\tilde{x} = x + n$, where n is a **random noise**;

  - Instead of minimizing $\frac{1}{2}||\hat{x} - x||^2$, we minimize $\frac{1}{2}||\hat{x} - \tilde{x}||^2$;
  - This is a form of implicit regularization that ensures **smoothness**: it forces the system to represent well not only the data points, but also their perturbations;

- **Stacked auto-encoders** - several layers of auto-encoders stacked together;

- Variational auto-encoders - learn a **latent variable model** of the data.

Regularized Auto-Encoders

- We need some sort of **regularization** to avoid **overfitting**;

- To regularize auto-encoders, **regularization** is added to the loss function;

- The goal is then to minimize $\mathcal{L}(\hat{x}, x) + \Omega(h, x)$;

- For example:
  - Regularizing the code: $\Omega(h, x) = \lambda||h||^2$;
  - Regularizing the derivatives: $\Omega(h, x) = \lambda \sum_i ||\nabla_x h_i||^2$.

Unsupervised Pre-training

- **Unsupervised pre-training** is a technique for **initializing** the weights of a **deep NN**;

- A **greedy, layer-wise procedure**:
  - Train one layer at a time, from first to last, using **unsupervised criterion (e.g. auto-encoder)**;
  - Fix the parameters of previous hidden layers;
  - Previous layers viewed as **feature extractors**.

- After pre-training, the **whole network** is **fine-tuned** using **supervised learning - fine-tuning**;
  - Performed as in a regular **feed-forward NN** - forward propagation, backpropagation and update of weights.

Word Representations

- **Word representations** are a **key component** of **LLMs (Large Language Models)**;
- Learning representations of **words in natural language** - also called **word embeddings**;
  - An extremely successful application of representation learning;
- **Distributional similarity** - represent a word by means of its neighbors - *you shall know a word by the company it keeps*;
- The objective is to obtain a **vector representation** for each word in a **vocabulary**; there are two main approaches:
  - Factorization of a co-occurrence word-context matrix;
  - Directly **predicting** a word from its neighbors in a **continuous word-space - word2vec**.

Neural Language Models

- **Embedding matrix**: assign a vector to every word in the vocabulary;
- Each word is associated with a **word embedding** - a vector of real numbers;
- Given the **context** (previous words), the **next word** is predicted;
- The word embeddings in the context window are **concatenated** into a vector that is fed to a **neural network**;
- The output of the NN is a **probability distribution** over the vocabulary - **softmax**;
- The network is trained by a **SGD with backpropagation**.

Word2Vec

- Often, we are not concerned with language modeling, but with the **quality of the word embeddings**;
  - We do not need to predict the probability of the next word, just make sure that the true word is more likely than a random one;
- **Word2Vec** is a **shallow, two-layer NN** that is trained to **predict** the **current word** from the **context**; it comes with two variants:

- **Continuous Bag-of-Words (CBOW)** - predict the current word from the context;
- **Skip-gram** - predict the context from the current word - **more popular**.

Skip-Gram

- **Objective**: maximize the log probability of any context word given the central word:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} log p_\theta(x_{t+j}|x_t)$$

- There are 2 sets of parameters (2 embedding matrices - $\theta = (u, v)$):
  - Embeddings for each word o appearing as the center word - u_o;
  - Embeddings for each word c appearing in the context of another word - v_c;
- Uses a **log-bilinear model**: $p_\theta(x_{t+j} = c|x_t = o) \propto exp(u_o^T v_c)$;
- Every word gets two vectors;
- In the end, we only care about the **word vectors** u, the **context vectors** v are discarded.

Large Vocabulary Problem

- The **softmax** is expensive to compute for large vocabularies, so there are some alternatives:
  - Stochastic sampling;
  - Noise contrastive estimation;
  - **Negative sampling**.

Negative Sampling

- **Key idea**: replace the **softmax** by **binary logistic regressions** for a true pair **(center word, context word)** and k random pairs **(center word, random word)**:

$$J_t(\theta) = log\sigma(u_o^T v_c) + \sum_{i=1}^{k} log\sigma(-u_o^T v_{j_i}), j_i \sim P(x)$$

- There are several strategies for sampling the random words;
- Negative sampling is a **simple form of unsupervised pre-training**.

Linear Relationships

- **Word embeddings** are good at encoding dimensions of similarity;
- **Word analogies** can be solved well simply via subtraction in the embedding space;
- A simple way to visualize the word embeddings is to use **PCA** to project them into 2D;
- There are other methods for obtaining word embeddings:
  - **GloVe -** Global Vectors for Word Representation;
  - **FastText -** subword embeddings.