ML (theory and practice)

Author: Daria Shutina

23-09-08

Data representation

Continuous variables are those that can take any real-numbered value within a certain range (e.g. height, temperature, weight).

Binary variables are categorical variables that can take one of two possible values.

Embeddings are a technique used to represent categorical or discrete data (e.g. words, categories, IDs) in a continuous vector space.

Datasets

Datasets are structured collections of data used for training, testing, and evaluating models.

Datasets consist of **instances**. Each instance represents a single data entry. **Features** (or attributes) are characteristics or properties associated with each instance. **Labels** (or target values) represent the desired output that the model should learn to make for each corresponding instance.

The **training dataset** is used to train the ML model. It consists of a large number of labeled examples, allowing the model to learn the relationships between the features and labels.

The **validation dataset** is a separate portion of the data used during the training process to assess the model's performance and make decisions about hyperparameters or model selection. It helps prevent overfitting.

The **testing dataset** is used to evaluate the final performance of the trained model. It should be distinct from the training and validation datasets and provide an unbiased assessment of how well the model works with new data.

Train, validation, test

Hyperparameters are like settings for the ML algorithm. At the beginning, hyperparam values are set based on what you believe would result in a good performance. This can be based on research or experience. After the hyperparams are set and the data is ready, we **train** the model on the training dataset.

The model's performance depends on the dataset it is being trained on. One of the main aims is to make the model robust, meaning the model gives consistent results that are correct most of the time.

After the training is done, results of the model might be not good enough, because hyperparameters do not have an optimal value. On the next step -- **tuning** -- different combinations of hyperparams are tested.

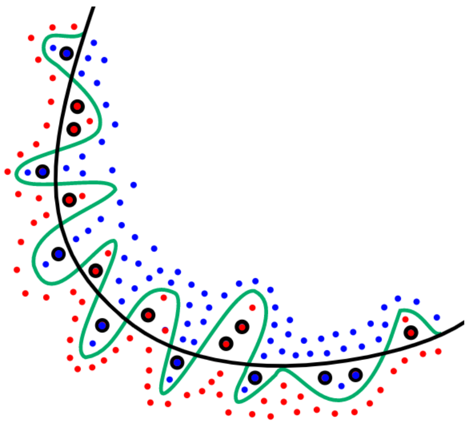
So, the main steps of preparing the model:

- **train**: find the model params. The model is trained with the training dataset.
- **validation**: tune hyperparameters. We tune hyperparams and train again, also checking for the performance. If needed, the tuning can be done again. This way the model is being optimized to show the best performance on the validation data.
- **test**: run the model on data the model has never seen.

We need to split data between train, validation and test datasets. It can be done randomly or sequentially.

Overfitting

In simple words, the model "overfits" when it is correct on every instance of the training dataset but has poor fit with new, unknown datasets.



Here, blue and red dots represent different classes, and the model is learning to distinguish between them. The green line is an overfitted curve, which means the model gives a correct answer on every instance from the training dataset. On the other hand, the black line is more general.

As a result, the model with the black curve has a higher probability of being correct on a new data, since the green curve is too fluctuating.

Loss functions

A **loss function** (or cost function) quantifies the difference between the predicted values generated by a model and the target values in the training dataset.

The goal in training a model is to minimize this loss function. The aim is to find the model's parameters that result in the lowest possible loss.

Example

We want know whether the user clicks the video. Lets use a **log loss function**:

$$L = -\log p \cdot y + \log(1-p) \cdot (1-y)$$

Here p is a prediction, it is calculated by our model, and y is a label (1 if click, 0 otherwise).

- $L \to \min$
- $y = 1 \;\Rightarrow\; -\log p \to \min$
    - $-\log 0 = +\infty$
    - $-\log 1 = 0$
- $y = 0 \;\Rightarrow\; \log(1-p) \to \min$
    - $\log(1-0) = 0$
    - $\log(1-1) = +\infty$

23-09-15

Linear models

Jupiter notebook practice

Moore-Penrose inverse method

We want to solve the equation $Xa = y$ => $a = X^{-1}y$. If X is not inverse (not squared), use $a = (X^TX)^{-1}X^Ty$.

```python
def solve_pseudo_inverse(X, y):
    matrix = np.linalg.inv(np.dot(X.T, X))
    return np.dot(np.dot(matrix, X.T), y)

print(solve_pseudo_inverse(X_train, y_train))
```

Stochastic Gradient Descent function

We have n features ($x_0, \dots, x_{n-1}$) and want to find coefficients $a_0, \dots, a_{n-1}$:

$$y_{pred} = f(A, X) = a_0x_0 + \dots + a_{n-1}x_{n-1}$$

The loss function will be the distance from real and predicted results:

$$L = (y_{pred} - y_{real})^2 = (\sum_0^{n-1} a_ix_i - y_{real})^2$$

$$L \to \min$$

We are going to use the gradient of the loss function to find coefficients. The gradient is a matrix consisting of derivatives, each of them has a form $\frac{\partial L}{\partial a_i} = L'_{a_i} = 2(a_0x_0 + \dots + a_{n-1}x_{n-1} - y_{real})x_i$.

```python
def solve_sgd(X, y, iterations, learning_rate):
    n = X.shape[1]
    a = np.zeros(n)

    for i in range(iterations):
        t = 2 * (np.dot(X, a) - y) * X.T
        t = np.mean(t, axis = 1)
        a -= t * learning_rate
    return a

print (solve_sgd(X_train, y_train, 1000, 0.01))
```

The `solve_sgd` function first initializes vector `a` with zeros and iteratively updates it in order to minimize the loss function error. In each iteration, it computes the gradient `t` for the current vector `a` and updates the coefficients.

`np.mean(t, axis = 1)` finds the average over all examples in the dataset. For example, t_1 is the average of derivatives' values for the coefficient a_1.

`learning_rate` is a hyperparameter that determines the step size in the direction of the gradient. Typically, it is a positive number between 0 and 1 that scales the gradient before applying it to update coefficients.

Decision trees

A **threshold** is a value that is used to make a decision or classification. For example, we want to assign a category for instances. The decision rule can be as follows:

$$Predicted\ class = \begin{cases} Class\ 1\ if\ Probability \geqslant Threshold \\ Class\ 0\ if\ Probability < Threshold \end{cases}$$

Parameters used in a **decision tree** are features, thresholds and conditions. Every node contains a condition - question of a form like `if feature1 < threshold1`.

The implementation for decision trees is `catboost`.

Boosting forest

$y = tree_0(x) + \ldots + tree_{n-1}(x)$.

Trees are not deep (3 to 6 levels). Each tree makes a tiny step towards the goal. When trees from 0 to $i-1$ are known, the i-th tree is found from the equation $y - tree_0(x) - \ldots - tree_{i-1}(x) = tree_i(x)$.

Unlike decision trees, boosted trees do not overfit quickly, since a tree with thousands of params is more likely to overfit.

```python
import catboost


def solve_catboost(X, y, iterations, learning_rate, depth):
    model = catboost.CatBoostRegressor(iterations = iterations,
                                       learning_rate = learning_rate,
                                       depth = depth)
    model.fit(X, y, verbose = False)
    return model


X_train, y_train = generate_data([1, 2, 3], 1000, 0.01)
X_test, y_test = generate_data([1, 2, 3], 100, 0.01)
model = solve_catboost(X_train, y_train, 1000, 0.1, 6)
```

A `CatBoostRegressor` model is trained using the `solve_catboost` function:

- `iterations` is the amount of decision trees
- `learning_rate` is used for controlling the step size in the direction of the gradient
- `depth` - depth of decision trees

Random forest

$$y = avg(tree_0(x), tree_1(x), \ldots, tree_{n-1}(x))$$

Each tree is trained on a sample of the original dataset.

To avoid overfitting, a technique called *bagging* is used:

- sample bagging: multiple decision trees are trained on random subsets (repeating data points is allowed) of the training data
- feature bagging: a random subset of features is selected for each tree

23-09-22

Language models

Language models are designed to understand and generate human language text. They are used for:

- classification
    - token classification is the task of assigning a label or category to each individual token or word in a given text sequence. Each token is typically classified independently based on its context within the sequence.
    - Sequence classification involves assigning a single label or category to an entire sequence of tokens or words.
- retrieval (you have a set of tokens and want to extract some data related to the given tokens)
- translation
- generation
    - Q&A: given a question and an article, come up with an answer based on the article
    - summarization: summarize data from the article
    - chat: e.g. ChatGPT

Word embeddings

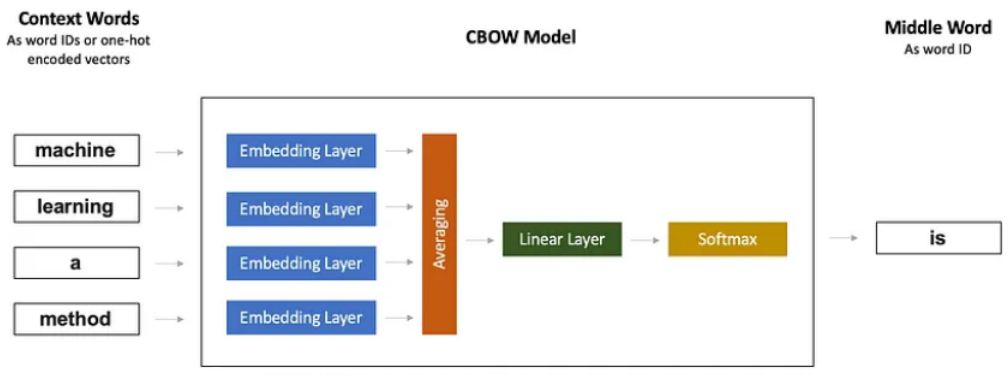
Imagine we have a document, each word from the document has a form $[0, \ldots, 0, 1, 0, \ldots, 0]$ as a vector. Depending on the document size and content, the size and amount of vectors can be huge.

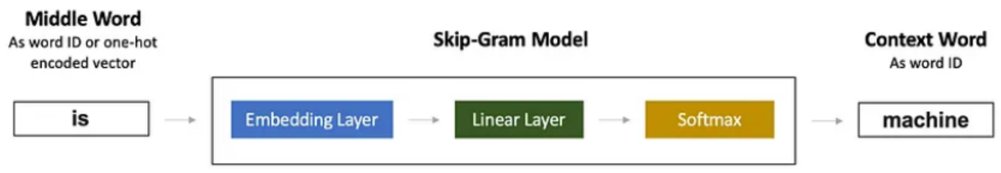
Embeddings are vector representations of words in a continuous vector space. The idea of using embeddings is to transform the word's vector $[0, \ldots, 0, 1, 0, \ldots, 0]$ into a n-dimensional vector $[e_1, \ldots, e_n]$.

Word2vec is an approach to create word embeddings. It is based on the idea that a word's meaning is defined by its context. Except for word2vec there exist other methods to create word embeddings, such as fastText, BERT, GPT-2, etc.

Word2vec model is very simple and has only two layers:

- Embedding layer which takes word ID and returns its 300-dimensional vector [1].
- Linear (Dense) layer with a Softmax activation.





Skip-gram and **CBOW (Continuous Bag of Words)** are two popular word2vec algorithms, used for training word embeddings. Skip-gram is often better at capturing semantic [2] relationships and is more suitable for tasks like word analogy (e.g. "king - man + woman = queen"). CBOW is faster to train and can perform well on syntactic [3] tasks.

- **CBOW**

  CBOW tries to predict the target word given a context. The input is a context (a set of surrounding words), and the output is the target word. The model attempts to maximize the probability of the target word given the context words.

  Example: given the same sentence "I love to eat pizza," if the context is "I, to, eat, pizza," CBOW aims to predict the target word "love."

- **Skip gram**

The main goal is to predict the context words given a target word. The input is a target word, and the output is a probability distribution over all the words in the vocabulary.

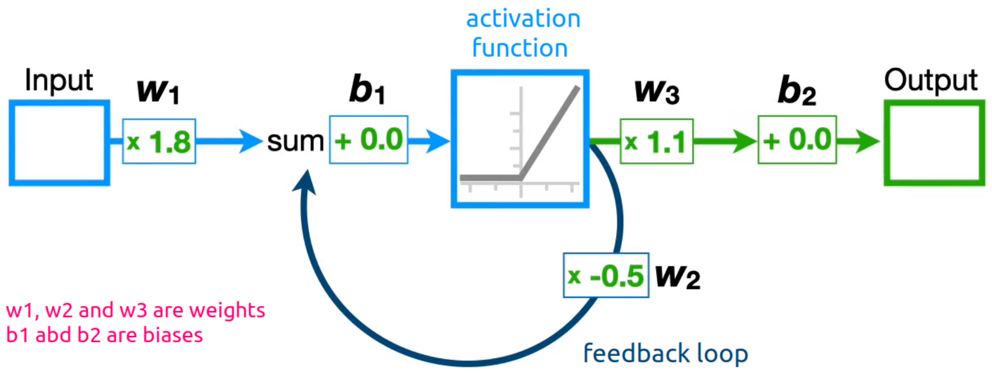
Example: given the sentence "I love to eat pizza," if the target word is "love," Skip-gram aims to predict the surrounding words like "I," "to," "eat," and "pizza."

RNNs

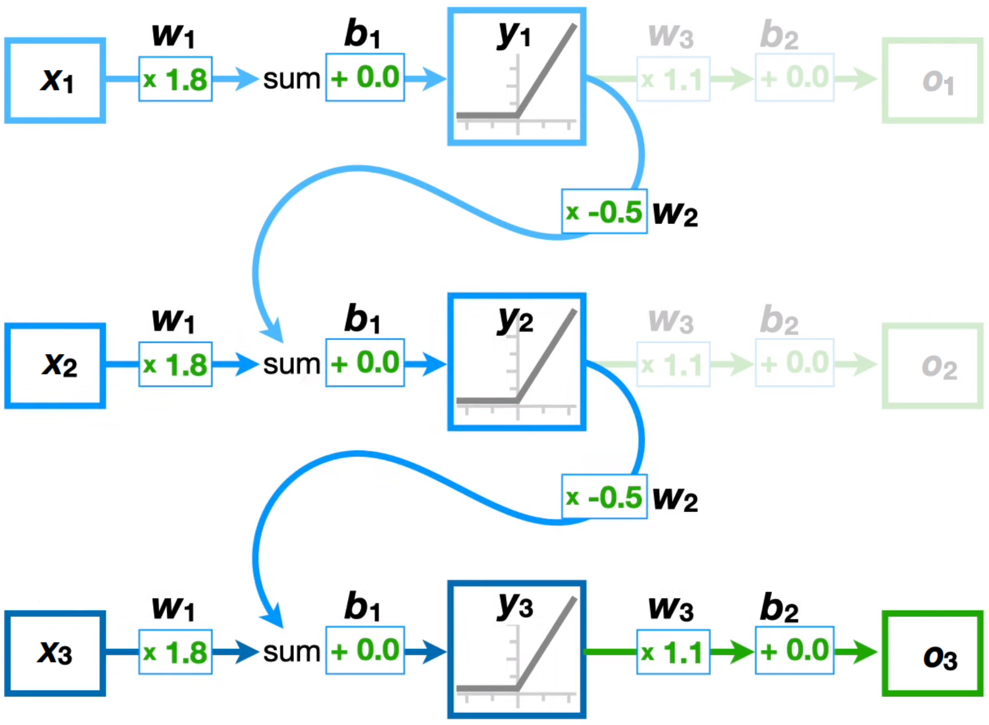
Activation functions are used to map scores to probabilities. There can be several activation functions, one per each level of the neural network. There are different types of activation functions:

- sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$. It is a probability for one score.
- softmax: $f(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$ It measures the probability for a group of scores.
- ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$
- log function: $f(x) = \log \frac{1}{e^x}$
- etc.

Recurrent neural networks deal with an arbitrary amount of input values. Like any other networks, RNNs have weights, biases, layers and activation functions. The difference is they also have **feedback loops**. The feedback loop makes it possible to use sequential input values to make predictions.



To understand how the feedback loop works, we can **unroll** it. Thus, we end up with a new neural network that has several inputs $x_1, \ldots, x_n$ and several outputs $o_1, \ldots, o_n$. The same weights and biases are shared across every input.



If we need the output o_n, intermediate results are simply ignored.

The problem with RNNs is that the more we unroll it, the harder it is to train. The problem is called **The Vanishing/Exploding Gradient Problem**.

When the gradient contains a huge number, relatively large steps are taken in the Gradient Descent algorithm. Thus, instead of finding the optimal parameter, we will just bounce around it. It explains the Exploding Gradient Problem.

One way to prevent The Exploding Gradient problem is to limit the parameter w_2 to values < 1. We end up taking steps that are too small and hit the maximum number of steps we are allowed to take before finding the optimal value. It explains the Vanishing Gradient Problem.

Transformers

Transformers represent a newer and more specialized type of neural networks, well-suited for tasks involving sequences of data, such as text, time series, and more. They can process input data in parallel, unlike RNNs, which are sequential.

Positional Encoding is a technique transformers use to keep track of the word order. Information about the position is added directly into the embedding.

23-09-29

Types of tasks

- classification
  - sequence

    `sequence -> label`

    used for spam classification, misinformation detection, result relevance
  - tokens

    `word/token/symbol -> label`. A sequence is divided into tokens.

    For example, "Mr. Johnson works in London" -> `[1, 2, 0, 0, 3]`, where `1, 2` = class "Person" (`1` is a beginning, `2` is a continuation of `1`) and `3` = class "Geo object".
- retrieval

  We have a query and a candidate for the answer. We want to process if the candidate is relevant.

  Retrieval is based on the token similarity. So, there are problems with detecting synonyms.

  The example is search. A query is vectorized, and appropriate responses are looked up in the vectorized space.
- generation

  `sequence -> sequence`

Local sensitive hashing

It is a technique to embed sentences. Strings that have a similar symbol representations have close vectors. The meaning of a string is not captured, only symbols matter, so this approach does not solve the problem with synonyms.

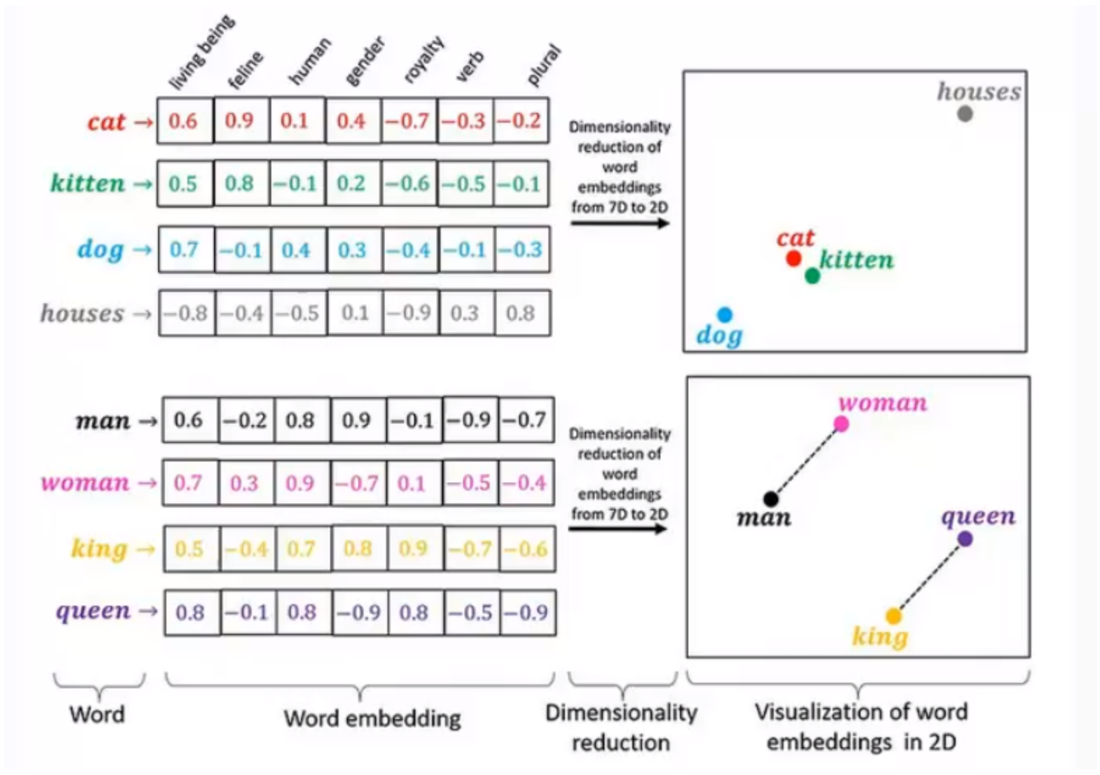
Word tokenisation

Words have plural forms. The common approach is to split a word by the most common tokens (main part, prefix, postfix, etc). For example, `cat -> [cat]` and `cats -> [cat, s]`.

This idea is implemented in the Byte-pair Encoding algorithm.

23-10-06

In the example below, the distance between similar items (e.g. `cat` and `kitten`) is small. Also, the distance between `man` and `woman` and the one between `king` and `queen` are the same.



	living being	feline	human	gender	royalty	verb	plural
cat	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8
man	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

Dimensionality reduction of word embeddings from 7D to 2D
Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

How to measure similarity of embeddings?

a and b are two vectors.

$$a \cdot b = \sum a_i b_i = |a||b| \cos \phi$$

$$sim(a, b) = \cos \phi = \frac{a \cdot b}{|a||b|}$$

If we do not want to deal with vectors' magnitudes, we can use a scaled dot product, where $\cos \phi = \frac{a \cdot b}{\sqrt{d}}$ and d is the dimensionality.

Capturing context: attention

Computes an attention distribution over the words -- a set of attention weights that determine how much focus or "attention" should be placed on each of the words in the sequence.

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Q, K, V are matrices packing together sets of queries, keys, and values, respectively.

$softmax$ is a function consisting of $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$. It transforms unscaled coordinates into probabilities:

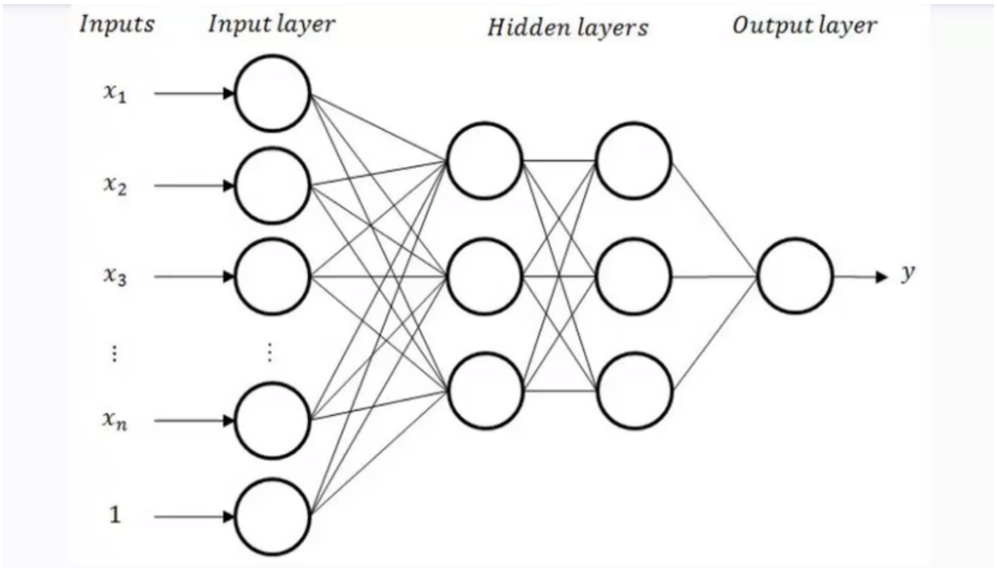
For example. we have a sentence `"buy an apple and an orage"`. Similarity between apple and orange is high: `sim(apple, orange) -> 1`.

Feed forward

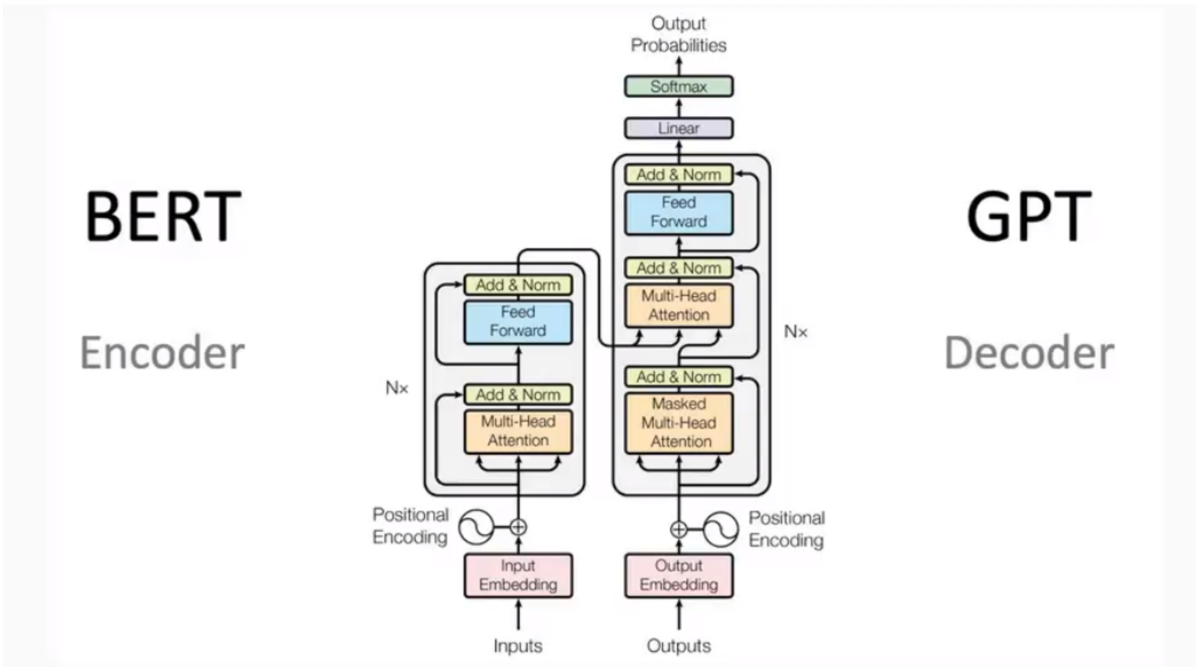
In general, the **forward** method defines the forward pass of the neural network. The forward pass is the process of taking the input, passing it through networks layers and then getting the final output. It may contain cycles or loops.

The **feed forward** provides a linear combination of the given embeddings. Information moves in the forward direction only -- from the input layer, through hidden layers (if any), and finally to the output layer. There are no cycles or loops in the network.

In transformers, two hidden layers are usually used for the feed forward.



Architecture of transformers \todo



Transformer consists of two main blocks -- the encoder and the decoder.

Positional encoding

It is possible to use only a bag of words. But is you want to get info about the order of words, it is better to use **positional encoding**. Info about positions is added directly to embeddings. As a result, the same word will have different embeddings on different positions.

$$f(pos, d, i) = \begin{cases} \sin(pos \cdot 1000^{-\frac{i}{d}}), \text{ if } i \mod 2 = 0, \\ \cos(pos \cdot 1000^{-\frac{i-1}{d}}), \text{ otherwise.} \end{cases}$$

- pos is position of the word in the sentence,
- d is dimension of the positional encoding,
- i is the i-th component of the result .

Activation functions

activations.ipynb

Activation functions are used to map scores to probabilities. There can be several activation functions, one per each layer of the neural network.

- threshold: $f(x) = 1 \; if \; x > T \; else \; 0 \; (T \text{ is threshold})$

- sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$. It is a probability for one score.
- softmax: $f(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$ It measures the probability for a group of scores.
- ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$
- softplus: $f(x) = \log(1 + e^x)$. A smooth version of ReLU

Example for using softmax function

Depending on the maximum value in `scores` array, we want to choose the corresponding element from `vectors` array.

First, we count probabilities with the softmax function. Then we use the probabilities to measure a candidate for each of three places in the `avg` array.

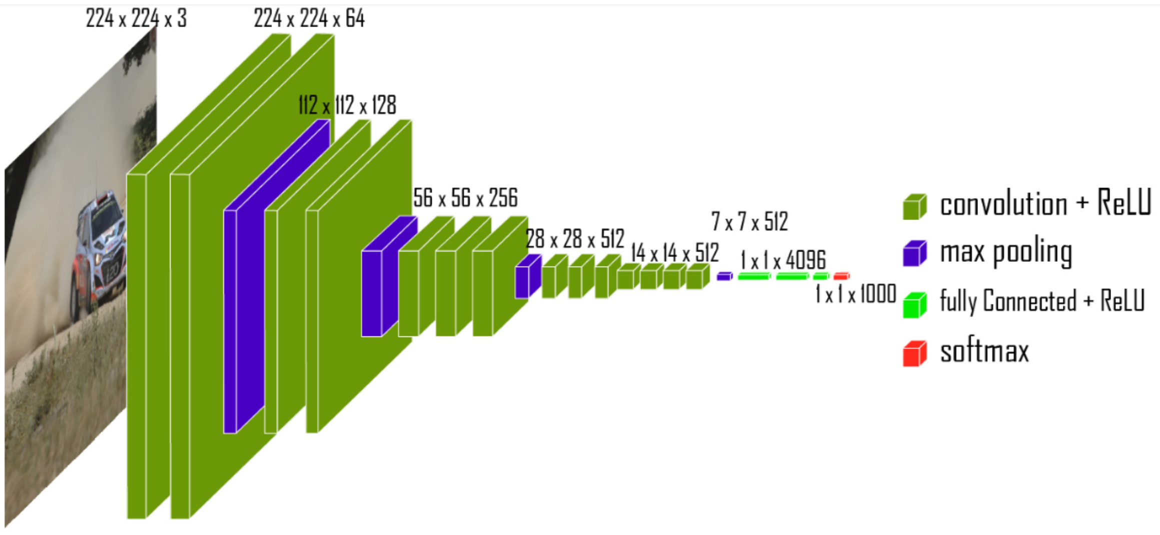
```python
# the maximum is `scores[1]`
scores = np.array([-5, 10, 2, 2, 3, 2])

probs = softmax(scores)
print(probs)

vectors = np.array([
    [1, 2, 3],
    [3, 2, 1],
    [0, 0, 0],
    [-1, -1, -1],
    [2, 2, 2],
    [2, 4, 8],
])

avg = np.sum(vectors.T * probs, axis = 1)
print(avg)
# the `avg` is close to `vectors[1] = [3, 2, 1]`
```

```
[0.012752 0.034663 0.000234 0.256127 0.696225]
[2.99641069 1.99899554 1.00225003]
```

23-10-13

Deep convolutional neural network: VGG

VGG stands for Visual Geometry Group. It is a deep convolutional neural network (CNN) which is effective in image classification tasks.



The architecture of the VGG neural network consists of 16+ layers of several types:

- **Convolution + ReLU**

  Convolution operation = apply a filter.

  ReLU is an activation function $f(x) = \max(x, 0)$. This helps the network learn complex features and allows for faster convergence during training.

- **Max pooling**

  Used to reduce the computational complexity. This is typically done by taking the maximum value from a small region of the feature map (e.g. 2×2 or 3×3).

- **Fully Connected + ReLU**

  In fully connected layers, each neuron in one layer is connected to every neuron in the previous layer. These layers are typically used for the final stages of feature extraction and decision making.

- **Softmax**

  Used as the final layer. Converts raw, unnormalized values of the previous layer into class probabilities.

Residual Neural Networks (ResNet)

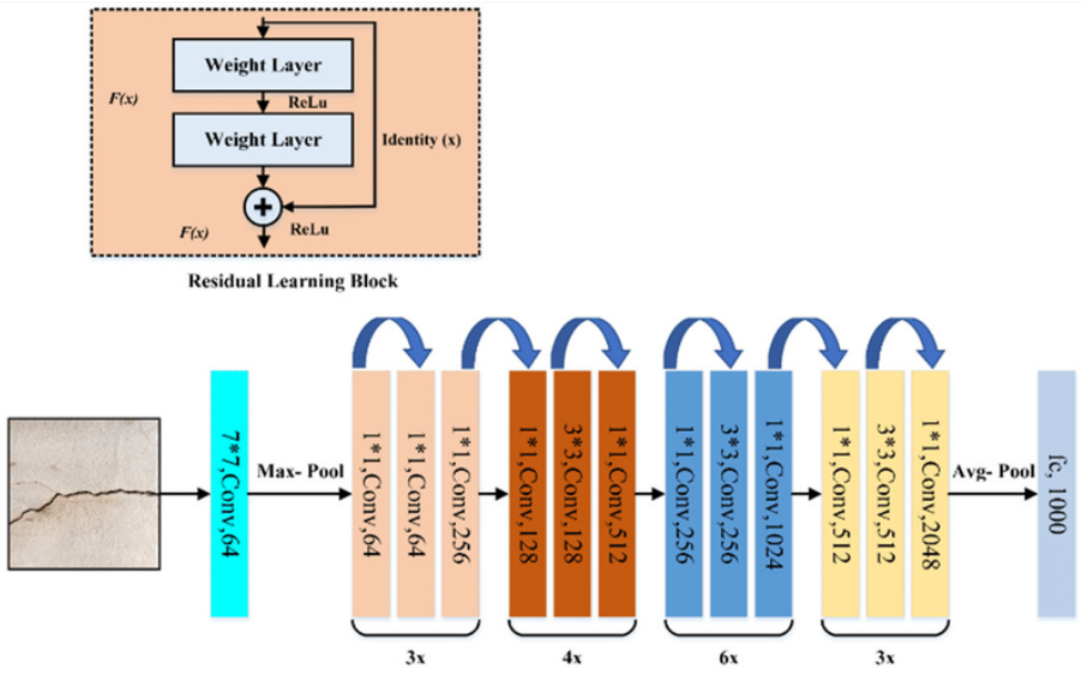
ResNets were developed to solve the problem of vanishing gradients in deep networks (which have a large number of layers). The key idea is to use **shortcut connections** that allow the gradient to flow more easily through the network during training.

They allow the input of the block to bypass one or more layers and be directly added to the output. For example, lets look at the identity shortcut connection. If x is the input to a residual block, and $F(x)$ is the output of the block (after passing through one or more layers), the shortcut connection is $x + F(x)$.

A **residual block** is the basic building block of ResNets. The key idea behind a residual block is the use of shortcut connections. Several residual blocks can be used in one ResNet.

The typical structure of a residual block can be described as follows:

- Main Path: consists of one or more convolutional layers.
- Shortcut Connection: provides a shortcut so that an input can be directly added to the output of the last layer.



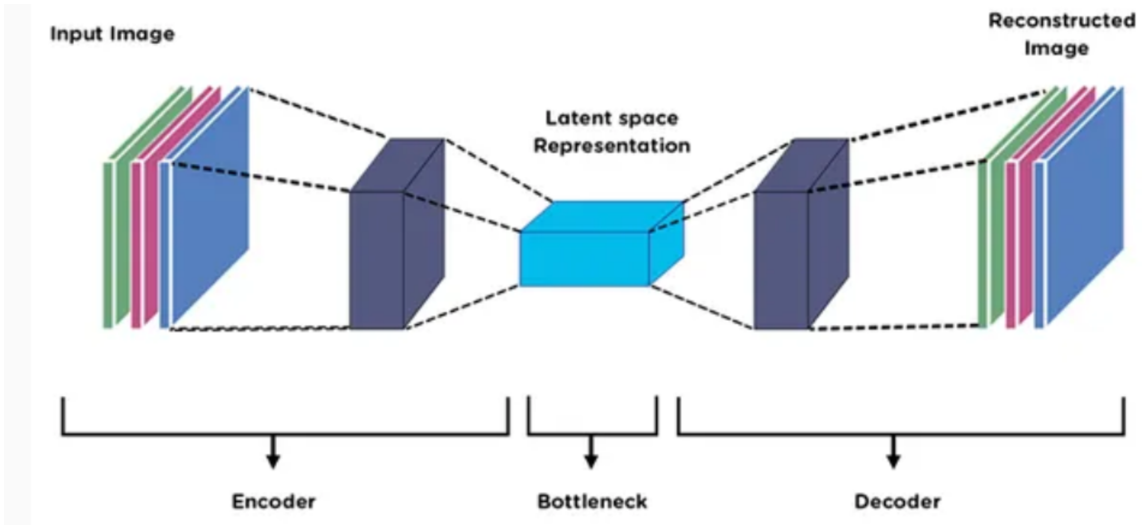
Multimodal approach, autoencoders

How can we combine different modalities, like text and image information? One of multimodal approaches is to create a **joint representation** of different modalities.

Suppose we have parallel corpus of text and images. Parallel means there is a mapping between text and images.

- We transform a source image and a source text into the same vector. This vector will be a joint representation of the corresponding image and text.
- The same is done in the opposite direction. We unpack the vector into the same source image and source text.

Multimodal approach idea is used in **autoencoders**. Autoencoder is a model which can be used to reduce dimensions of the input data. The compression of data is called a **bottleneck**. We compress input data into a smaller vector space which is called a **latent space representation**. Then we reconstruct data using a decoder and we want the result to be exactly the same as the input.



23-10-20 \todo

23-10-27

Perceptron

A **perceptron** is a fundamental building block in artificial neural networks. It is a simple binary classifier.

The basic **idea** of a perceptron is inspired by the way biological neurons work in the human brain. It takes multiple binary inputs (usually 0 or 1), each multiplied by a weight, and produces a binary output (1 or 0) based on a weighted sum and a threshold [4].

The output y of a perceptron for inputs $x_1, \ldots, x_n$ with weights $w_1, \ldots, w_n$ can be expressed as:

$$y = \begin{cases} 1\; if \;\sum\limits_{1}^{n} w_i x_i > threshold \\ 0\; otherwise \end{cases}$$

The threshold is a predefined value. The perceptron makes decision based on whether the sum exceeds the threshold.

For example, a set of inputs x_1, x_2 is given.

- First layer: $h_1 = ax_1 - bx_2 > 0.5$
- Second layer: $h_2 = cx_2 - dx_1 > 0.5$
- Third layer: $y = eh_1 + fh_2 > 0.5$
- Used weights are: $a = 1, b = -1, c = -1, d = 1, e = 1, f = 1$
- The threshold is $[t_1, t_2, t_3] = [0.5, 0.5, 0.5]$

IMPORTANT: functions used at layers of a perceptron should be activation functions. Otherwise it is simply a multiplication of matrices $\Rightarrow$ linear $\Rightarrow$ it is actually only one-layered perceptron. We want it to be multilayered.

Section about activation functions can be found here.

Backpropagation

Backpropagation, short for "backward propagation of errors," is an algorithm used to update weights of the model in order to minimize the error between predicted and target outputs. The algorithm allows a neural network to learn from its mistakes and adjust its weights accordingly.

Backpropagation is a part of the training process. In whole, the sequence of actions is as follows:

- **Forward Pass:**

  The input data is fed into the neural network, and the network computes a predicted output. The predicted output is then compared to the target output, and the **error is calculated**.

- **Backward Pass (Backpropagation)**

  The error calculated in the previous step needs to be quantified into a single scalar value. That is where the **loss function** is used: it computes a numerical value that represents the error between the predicted and target outputs.

  The algorithm then **calculates the gradient** of the error. This is done using the chain rule[5] of calculus.

- **Weight update:**

  The weights of the network are updated using the **gradient descent**. The gradient shows the direction where the loss function grows the most. So we move weights into the opposite direction -- towards the anti-gradient.

  The learning rate is a hyperparameter that determines the size of the step taken during the weight update.

  The code example can be found here.

- **Iterations:**

Steps 1-3 are repeated for multiple iterations (epochs) until the neural network converges to a state where the error is minimized, and the weights have been adjusted to produce accurate predictions.

Momentum

The **learning rate** determines the size of the step taken during the weight update. If the step is too large, we may end up in bouncing around the correct answer (Exploding Gradient problem). If it is too small, we may run out of steps allowed to be taken for finding the answer (Vanishing Gradient problem).

There are techniques that can speed up the training process. One of such optimizing algorithms is called **momentum**. The idea behind it: suppose we took several small steps. If all the steps go towards the same direction, then we can continue going with a higher speed. If the direction changed, we want to decrease the speed.

So, the momentum algorithm, we look on the previous gradient.

$$shift(k) = B \cdot shift(k-1) + (1-B) \cdot grad(k)$$

- $B \in (0,1)$ is a momentum coefficient -- a predefined constant,
- $shift(k)$ represents the update to be applied to the current weights at iteration k,
- $grad(k)$ is the gradient of loss function at the iteration k.

$shift(k)$ plays as role of a modified learning rate used during the k-th iteration.

23-11-03

Dimensionality reduction

Embeddings are usually high-dimensional vectors, and there are some methods that can reduce dimensionality. One of possible reasons why we need dimensionality reduction is about visualizing embeddings. High-dimensional vectors can be reduced to two-dimensional ones which, in turn, can be represented on the graph.

Dimensionality reduction is also used in Autoencoders.

Principal Component Analysis (PCA)

The goal is to transform high-dimensional data into a new coordinate system. This transformation is achieved by finding the principal components (like main directions) of the data.

- The transformation process is linear.
- New data can be easily adapted. Once the principal components are calculated, the transformation matrix remains fixed.

Suppose we have data with n exaples, each example has p features. Thus, we get a matrix X of size $n \times p$. Step-by-step explanation of PCA:

- **Data Standardization:** Before applying PCA, it's common practice to standardize the data. It ensures that all variables are on a similar scale, preventing variables with larger magnitudes from dominating the analysis.
- **Covariance Matrix** $X^T X$ (size if $p \times p$). It shows how the features are correlated with each other.
- **Eigenvalue Decomposition**: eigenvectors (of size $p \times 1$) are principal components we are looking for, they form a new basis for the data. Eigenvalues represent the magnitude of variance[6] in the data along the corresponding eigenvector: the higher the eigenvalue, the more important the eigenvector is in explaining the variance.
- **Selecting Principal Components:** The eigenvectors are ranked based on their corresponding eigenvalues. Top k eigenvectors are chosen (k is the desired dimensionality of the reduced data), they are used as a new basis for the data.
- **Projection:** The selected eigenvectors are used to create a transformation matrix V, its size is $p \times k$. The original data is projected onto the new basis: $X_{new} = X \cdot V$.

T-distributed Stochastic Neighbor Embedding (t-SNE)

The goal is to take high-dimensional data and represent it in a lower-dimensional space, often with two or three dimensions, while preserving the pairwise similarities between data points.

- The transformation is non-linear, allowing to capture some interesting cases.
- It is hard to integrate new data.

Step-by-step explanation of t-SNE:

- **Pairwise Similarities:**

  The algorithm starts by calculating pairwise similarities between all data points in the high-dimensional space. It uses a Gaussian distribution as a similarity function. The higher the probability, the more similar the points are.

  On the lecture, the magic function is used to measure similarity:

$$p(j|i) = \frac{e^{-dist^2(x_i,x_j)\ /\ s}}{\sum e^{-dist^2(x_i,x_k)\ /\ s}}$$

In the formula, the resulting probability is the similarity between points x_i and x_j, normalized by the sum of similarities between x_i and all other points x_k.

- **Probabilities in Low-Dimensional Space:**

  Next, t-SNE defines a similar set of pairwise probabilities in the lower-dimensional space. The similarity function is the same as in the previous step.

  The goal is to find a mapping that minimizes the divergence between the pairwise probabilities in the high-dimensional space and the pairwise probabilities in the low-dimensional space.

- **Optimization:**

  For each vector from the high-dimensional vector space, we want to find the best match from the low-dimensional vector space, such that the pairwise probabilities in the low-dimensional space approximate the pairwise similarities in the high-dimensional space.

  The choice of the best match is done by minimizing the Kullback-Leibler divergence between the two sets of probabilities.

Uniform manifold for approximation and projection (UMAP)

UMAP focuses on identifying the underlying **manifold**—a lower-dimensional space that captures the essential features and relationships within the high-dimensional data.

- New data can be easily integrated. UMAP constructs a weighted graph in high-dimensional space and its **projection** in the low-dimensional one.

  When a new point is added to the high-dimensional space, its relationship with existing points (neighbors) is computed. Then its projection in the low-dimensional space is found based on the neighbors' projections.

Step-by-step description of UMAP:

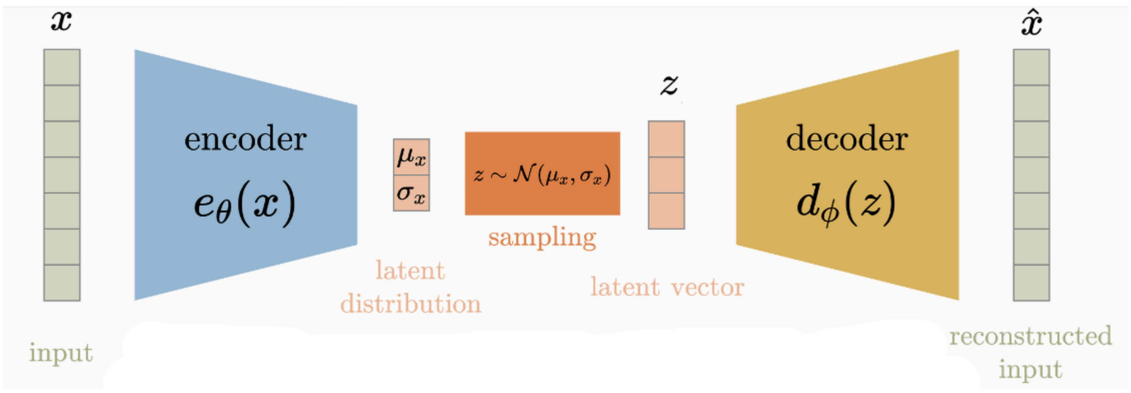
- **Define similarity based on topology**. For each data point, identify its nearest neighbors based on a distance metric. Then create a weighted graph, edges between nodes represent the strength of their similarity.
- **Do the same in lower-dimensional space**. UMAP creates an initial layout in the lower-dimensional space (often randomly) and builds a weighted graph. The algorithm then optimizes this layout, adjusting the positions of points iteratively.
- **Optimize lower-dimensional embedding**. After the configuration in lower-dimensional space was randomly chosen, we optimize it using stochastic gradient descent. As a result, we get a configuration that minimizes cross-entropy[7] between high-dimensional and lower-dimensional graph representations.

Variational autoencoder

The basic idea of the **autoencoder** is to create a latent space representation -- a vector space with reduced dimensionality. There is also an encoder and a decoder to work with the mentioned representation.

A **variational autoencoder** (VAE) is a specific type of autoencoder that also learns the **probability distribution** of the data. Inputs are mapped to a probability distribution in the latent space.

Variational autoencoders are powerful for generating new data that resembles the training data. The encoder of the VAE evaluates distribution and produces mean params, variance params and **samples** from it. The encoder takes the samples to reconstruct data.



23-11-10

Recommender systems

Recommender systems in are algorithms and techniques designed to predict a content a user might like, based on their preferences and behavior.

The basic idea of using recommender systems is to make recommendations personal. We extract some features from items, ask users about their preferences and then define similarity between items and users. After this we can rank items according to this similarity.

Examples of recommender systems are:

- e-commerce platforms: Amazon
- streaming services: Netflix, YouTube, Spotify
- social media: Facebook, Twitter
- Learning platforms: Coursera, Stepik
- Search engines: Google
- etc

Feedback

- Explicit (like/dislike, reviews, comments)
- Implicit
    - watching/listening a significant part of the video
    - subscription to the channels
    - adding to the playlist
    - buying the product
    - sharing a link

Measurement

- RMSE on ratings
- BCE on likes
- User behaviour metrics
    - long interactions
    - purchases
    - establishing connections

Content-based filtering

Expert choice

Expert choice is the algorithm used in content-based filtering.

1. **Expert Selection** -- choose people who are experts in a particular field.
2. **Item Evaluation**: Experts assess items based on various criteria such as quality, relevance, novelty, etc. This evaluation could be subjective (based on personal judgment) or objective (using predefined criteria).
3. **Aggregation**: Once experts have assessed items, their opinions are aggregated to form an overall assessment of each item. Aggregation may involve averaging scores, weighted voting, etc.
4. **Recommendation Generation**: The items with the highest overall scores from the aggregated expert evaluations are recommended to users.

The main disadvantages are 1) the result is non-personal and 2) the result is biased, since it is based on the experts' opinions.

`tf*idf`

When we have a text, our goal is to extract the most informative words. `tf*idf` is a method of formalizing the words according to their "meaningfulness". It is used to evaluate the importance of a word in a document relative to a collection of documents. TF measures how often a word appears in a document, while IDF measures how unique or rare a word is across multiple documents.

`tf(word, document)` - **term frequency** - number of occurrences of the word in the document. Usually normalized by max occurrence in some other doc.

`idf(word)` - **inverse document frequency** - number of documents containing the word. Usually normalized by $\log$.

Document representation `emb(doc) = [tf(word, doc) * idf(word) for word in doc]`.

Similarity of two docs = `cos(emb(doc1), emb(doc2))`.

Similarity of query and doc = `cos(emb(query), emb(doc2))`.

Collaborative filtering

This technique makes predictions about the interests of a user by collecting preferences from many users. It doesn't require any information about the items, only data about item-user interactions.

There are two main types of collaborative filtering:

1. **User-Based:** recommends items to a user based on preferences of similar users. For example, if User A and User B have similar tastes and User A liked a certain item, the system will recommend that item to User B.
2. **Item-Based:** recommends items similar to the ones a user liked in the past. If a user liked Item X, the system will recommend items similar to Item X.

   `score(user, item) = avg(rank(user, k) * sim(k, item))` for all items `k` ranked by the user.

User-item similarity

In collaborative filtering, predictions are based on user-item interactions. The idea of the **matrix factorization** method is to configure a latent representation (embeddings) for users and items from these interactions.

Training:

- Suppose we have `N` users and `M` items. `A[N, M]` is the interaction matrix. Rows correspond to users, columns correspond to items, values represent the interaction metric.
- `L` is the dimensionality of the latent representation.

  `U[N, L]` - matrix of user embeddings, `I[M, L]`- matrix of items embeddings.
- Factorize matrix `A` into the product of matrices `U` and `I^T`, so that `A = U * I^T`.

  During the training process, matrices `U` and `I` are optimized using techniques like gradient descent. The aim is to minimize the difference between the predicted interactions (`U x I^T`) and the actual interactions in matrix `A`.

Prediction:

To make recommendations for a user, you first find the user's embedding in the latent space which is formed by matrices `U` and `I`. Then, you calculate the similarity between the user's embedding and other embeddings.

Once you've calculated similarities, you create a ranked list of items. The top-N items are recommended to the user.

For recommendation purposes, you might look for the most similar items to the ones the user has interacted with previously. For example, a user interacted with items A, B, and C. You find the embeddings for A, B, and C in the latent space then calculate similarity between A, B, C and all other items. The items with the highest similarity scores to A, B, C would form the recommendations for the user.

1. Word2vec embeddings are 300-dimensional, as authors proved this number to be the best in terms of embedding quality and computational costs. ↩

2. Semantic refers to the meaning of words. It deals with the interpretation of words and their relationship to each other. Example: In the sentence "The cat chased the mouse," the semantic meaning is that the cat is pursuing the mouse, indicating a specific relationship between the two animals. ↩

3. Syntactic refers to the structure and arrangement of words in sentences and how they form grammatically correct sentences. Example: In the sentence "The cat chased the mouse," the syntactic structure follows the typical subject-verb-object order in English. ↩

4. A value used to make a classification. ↩

5. Chain rule describes how to find derivative of the composition of two (or more) functions. If $y = f(g(x))$, then $\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$. ↩

6. Variance (отклонение) of data shows how much individual data points in deviate from the mean (average) value. ↩

7. Cross-entropy is a difference between probability distributions (in ML, between predicted and actual ones). ↩