

ML (theory and practice)

Author: Daria Shutina

ML (theory and practice)

23-09-08

Data representation

Datasets

Train, validation, test

Overfitting

Loss functions

23-09-15

Linear models

Moore-Penrose inverse method

Stochastic Gradient Descent function

Decision trees

Boosting forest

Random forest

23-09-22

Language models

Word embeddings

RNNs

Transformers

23-09-29

Types of tasks

Local sensitive hashing

Word tokenisation

23-10-06

How to measure similarity of embeddings?

Capturing context: attention

Feed forward

Architecture of transformers \todo

Positional encoding

Activation functions

23-10-13

Deep convolutional neural network: VGG

Residual Neural Networks (ResNet)

Multimodal approach, autoencoders

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:

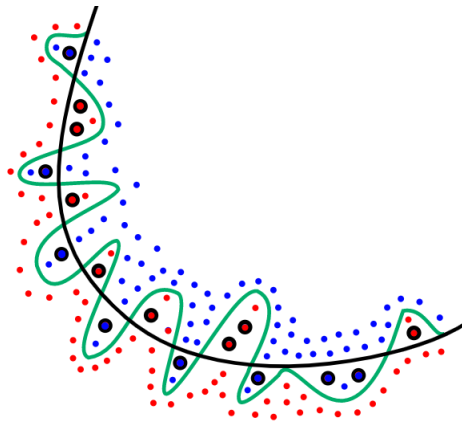
- **train:** find the model params. The model is trained with the training dataset.
- **validation:** tune hyperparameters. We tune hyperparams and train again, also checking for the performance. If needed, the tuning can be done again. This way the model is being optimized to show the best performance 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 to know whether the user clicks the video. Let's 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).

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

23-09-15

Linear models

[Jupyter notebook practice](#)

Moore-Penrose inverse method

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

```

1 def solve_pseudo_inverse(X, y):
2     matrix = np.linalg.inv(np.dot(X.T, X))
3     return np.dot(np.dot(matrix, X.T), y)
4
5 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} :

$$y_{pred} = f(A, X) = a_0 x_0 + \dots + a_{n-1} x_{n-1}$$

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

$$L = (y_{pred} - y_{real})^2 = \left(\sum_0^{n-1} a_i x_i - y_{real} \right)^2$$

$$L \rightarrow \min$$

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

$$\frac{\partial L}{\partial a_i} = L'_{a_i} = 2(a_0x_0 + \dots + a_{n-1}x_{n-1} - y_{real})x_i.$$

```

1  def solve_sgd(X, y, iterations, learning_rate):
2      n = X.shape[1]
3      a = np.zeros(n)
4
5      for i in range(iterations):
6          t = 2 * (np.dot(X, a) - y) * X.T
7          t = np.mean(t, axis = 1)
8          a -= t * learning_rate
9      return a
10
11 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

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

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

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

$$y - tree_0(x) - \dots - tree_{i-1}(x) = tree_i(x).$$

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

```

1  import catboost
2

```

```

3
4 def solve_catboost(X, y, iterations, learning_rate, depth):
5     model = catboost.CatBoostRegressor(iterations = iterations,
6                                         learning_rate = learning_rate,
7                                         depth = depth)
8     model.fit(X, y, verbose = False)
9     return model
10
11
12 X_train, y_train = generate_data([1, 2, 3], 1000, 0.01)
13 X_test, y_test = generate_data([1, 2, 3], 100, 0.01)
14 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

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

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

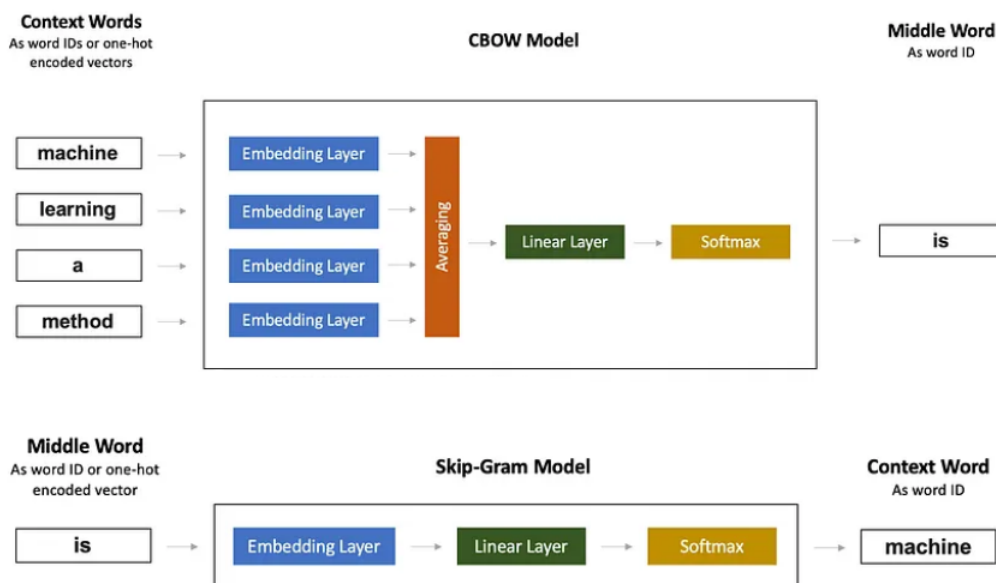
Imagine we have a document, each word from the document has a form $[0, \dots, 0, 1, 0, \dots, 0]$ as a vector. Depending on the document size and content, the size and amount of vectors can be huge.

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

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 vector¹.
- 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 semantic² 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 syntactic³ 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.

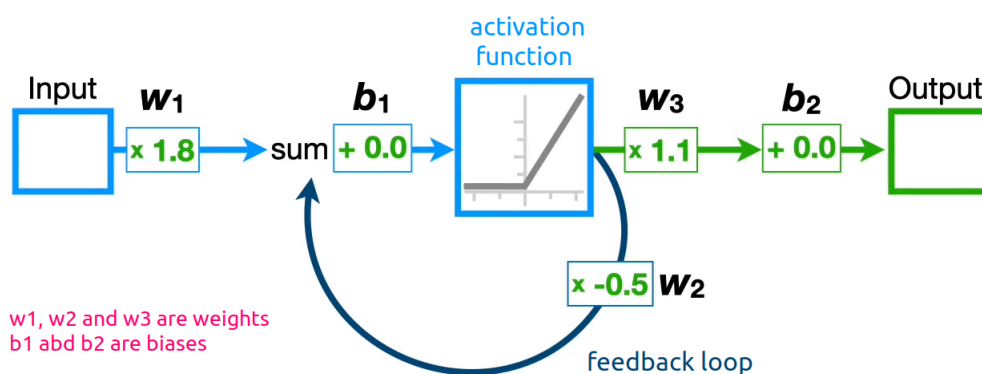
Example: given the sentence "I love to eat pizza," if the target word is "love," Skip-gram aims to predict the surrounding words like "I," "to," "eat," and "pizza."

RNNs

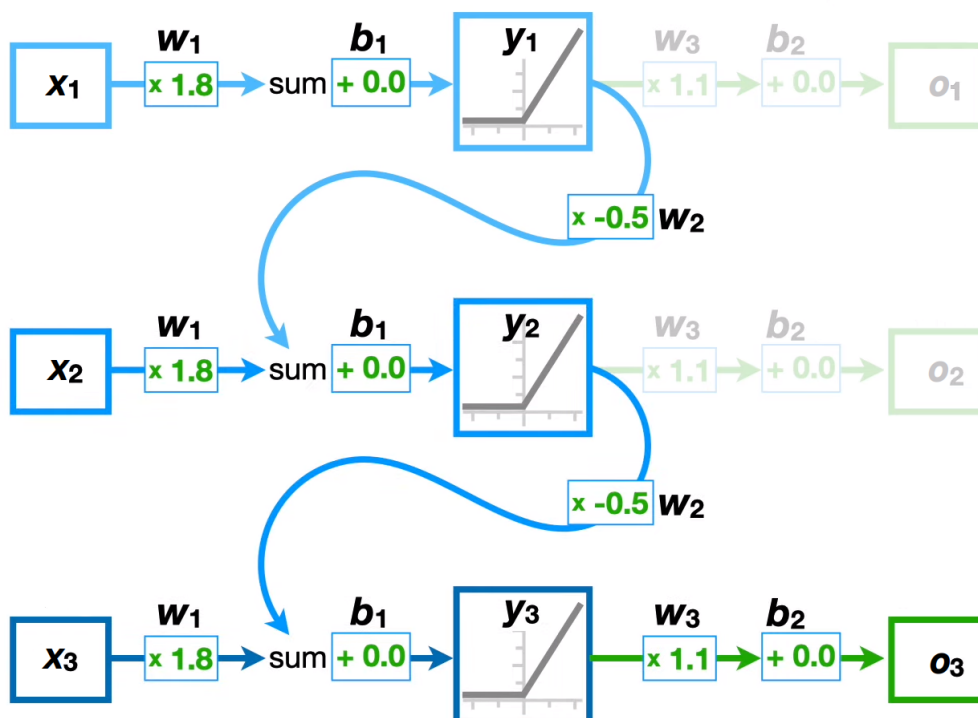
Activation functions are used to map scores to probabilities. There can be several activation functions, one per each level of the neural network. There are different types of activation functions:

- 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, \dots, x_n and several outputs o_1, \dots, 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.

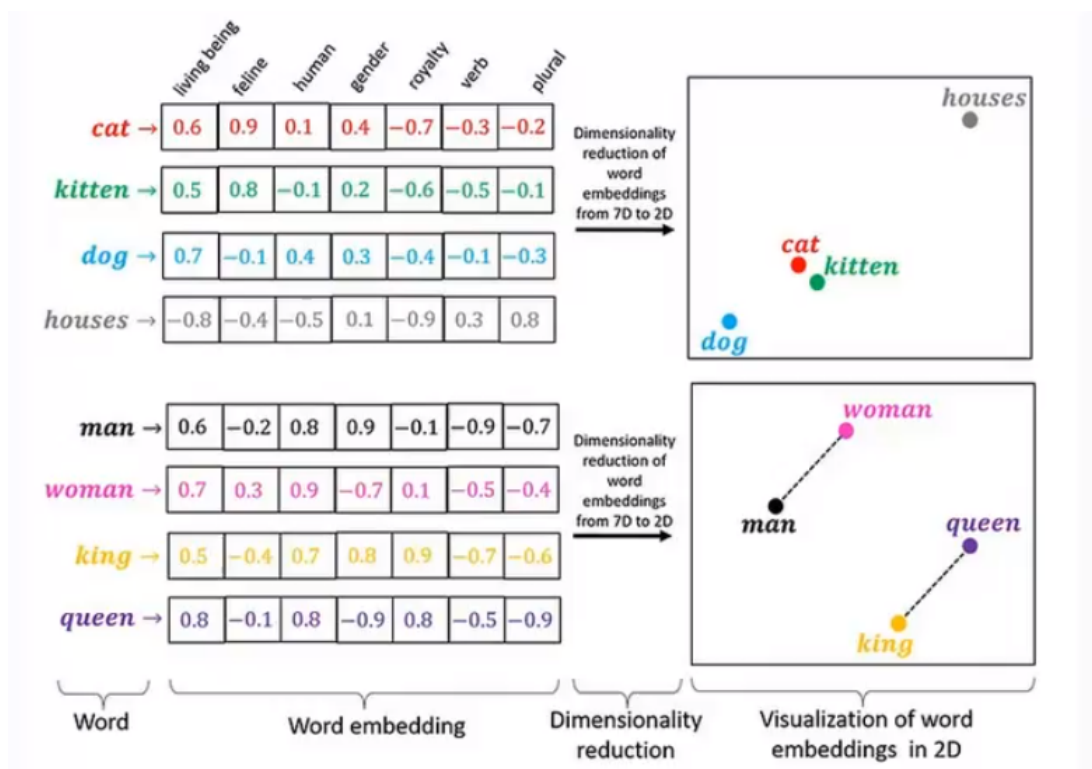
Word tokenisation

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.



How to measure similarity of embeddings?

a and b are two vectors.

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

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

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:

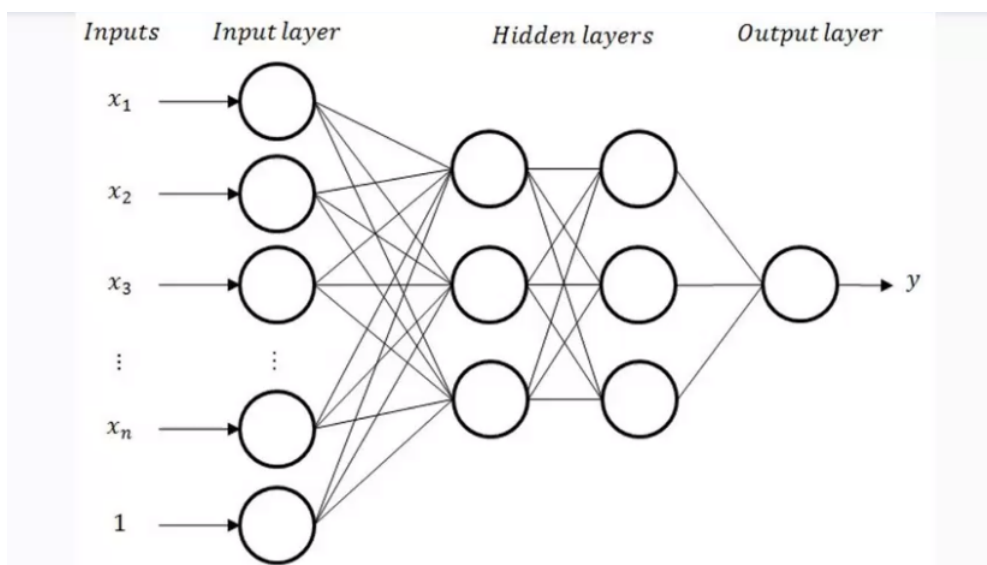
For example, we have a sentence "buy an apple and an orange". Similarity between apple and orange is high: `sim(apple, orange) -> 1`.

Feed forward

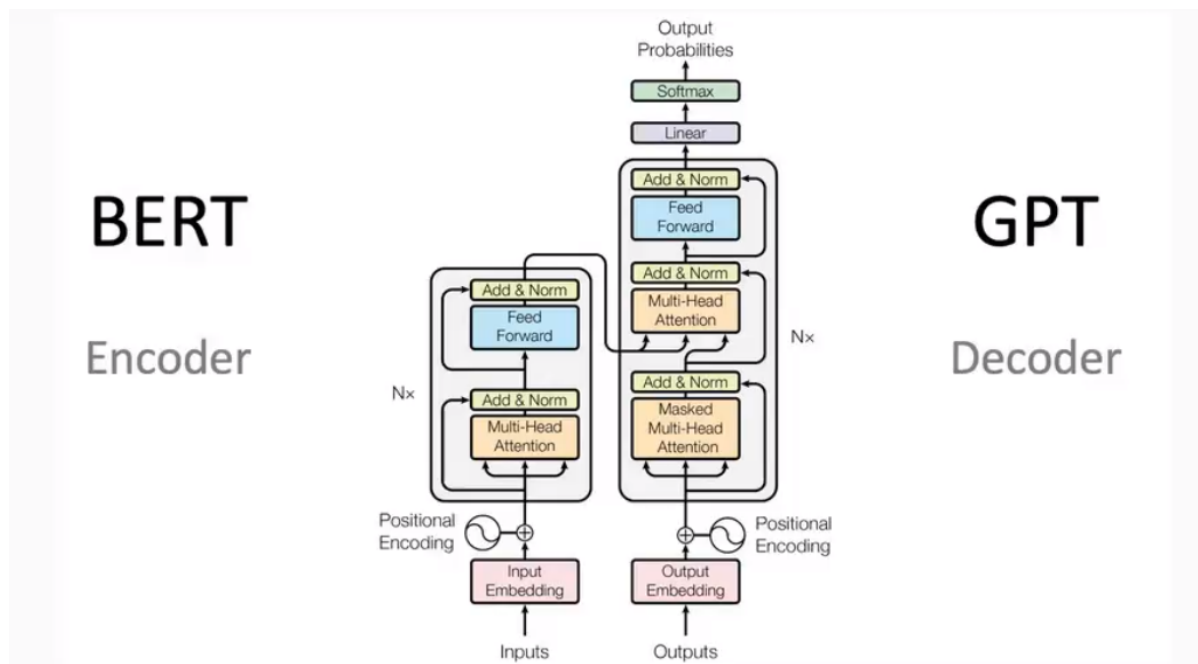
In general, the **forward** method defines the forward pass of the neural network. The forward pass is the process of taking the input, passing it through networks layers and then getting the final output. It may contain cycles or loops.

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 if 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.

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

- 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.

- 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.

```

1  # the maximum is `scores[1]`
2  scores = np.array([-5, 10, 2, 2, 3, 2])
3
4  probs = softmax(scores)
5  print(probs)
6
7  vectors = np.array([
8      [1, 2, 3],
9      [3, 2, 1],
10     [0, 0, 0],
11     [-1, -1, -1],
12     [2, 2, 2],
13     [2, 4, 8],
14 ])
15
16 avg = np.sum(vectors.T * probs, axis = 1)
17 print(avg)
18 # the `avg` is close to `vectors[1] = [3, 2, 1]`

```

```

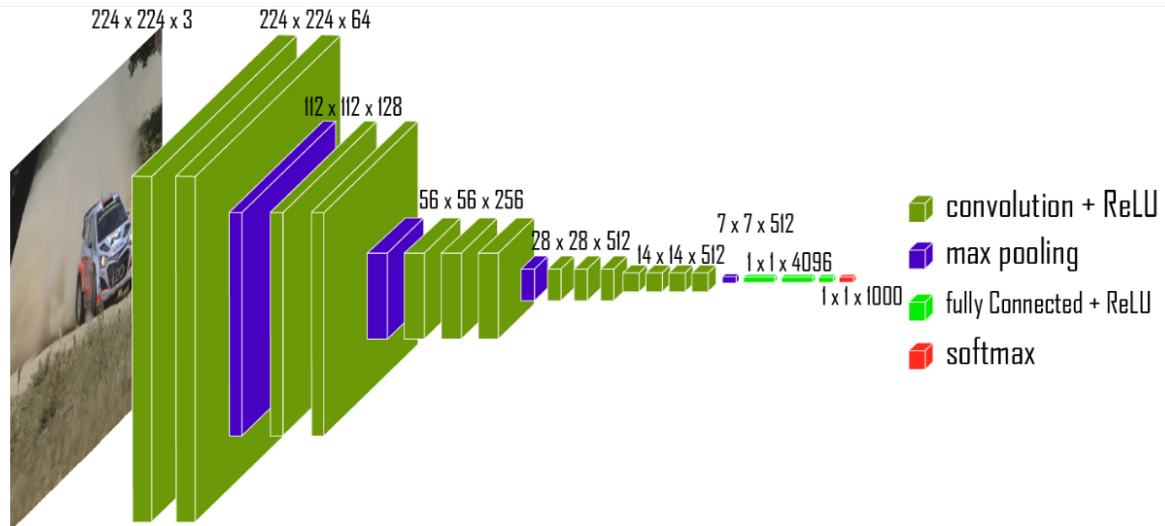
1  [0.012752 0.034663 0.000234 0.256127 0.696225]
2  [2.99641069 1.99899554 1.00225003]

```

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 equal to $f(x) = \max(x, 0)$. 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)

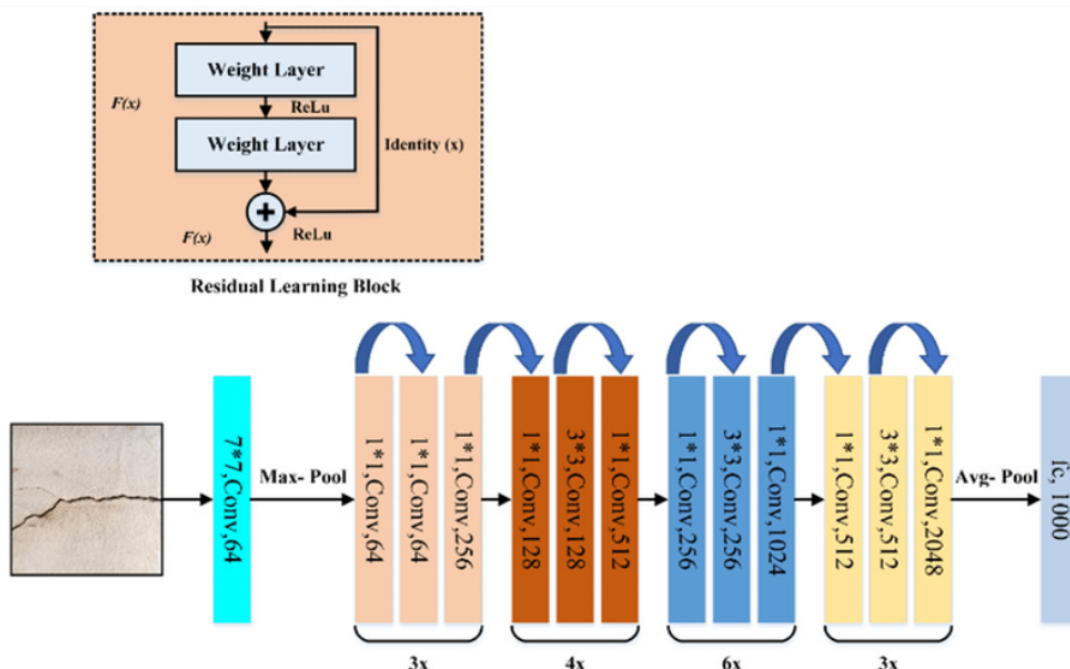
ResNets were developed to solve the problem of vanishing gradients in deep networks (which have a large number of layers). The key idea is to use **shortcut connections** that allow the gradient to flow more easily through the network during training.

They allow the input of the block to bypass one or more layers and be directly added to the output. For example, let's 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.



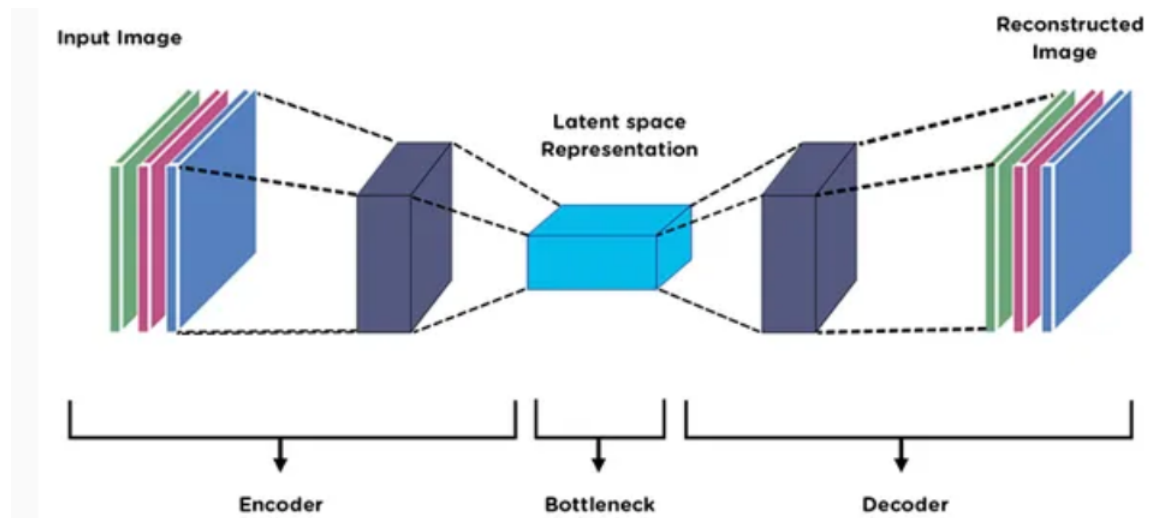
Multimodal approach, autoencoders

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.



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. [↵](#)