DAT300

Summary.

Artificial Neural Network (ANN)

A collection of collected neurons. Each neuron is then connected to every neuron in the next layer, where the connection strength is the weight w_i. Each neuron in the next value therefore has the "output" as the sum of each neuron-value of the previous layer times the weight, plus a bias term:

$$y_i = (\sum_{j=1}^{n} x_j \times w_{ji}) + b_i$$

where y_i is the output neuron, x_j the previous layer's neurons, w_{ji} the weight between x_j and y_i and b_i the bias of neuron y_i.

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(8*8,)))
model.add(Dense(10, activation='softmax'))
```

In this example, we set up a Sequential model with 64 input-neurons. The input shape is $8 \times 8 = 64$. In this model, we have ten outputs, and apply the softmax activation function in order to get the probability distribution. Here we assume that there are ten categories that the model tries to predict for a given input.

Softmax activation function

Normalises the output into a probability distribution.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

where $\sigma(z_i)$ is the output probability for neuron i, z_i the output of neuron i and N the total number of output neurons.

Epochs and Batch Size

Each complete presentation of the dataset is referred to as an epoch. The batch size determines how many data points are used in a single update of the model weights, and the number of batches in an epoch is given by the total number of data points divided by the batch size.

Backpropagation

The error in the prediction is computed, and propagated back through the network to adjust the weights.

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial h_j} \cdot \frac{\partial h_j}{\partial w_{ij}}$$

where $\frac{\partial L}{\partial \hat{y}_j}$ is how the Loss changes as the output changes, $\frac{\partial \hat{y}_j}{\partial h_j}$ how the output changes based on the changes in its inputs and $\frac{\partial h_j}{\partial w_{ij}}$ how much the input to the neuron changes with a weight change.

Keras and Basic TensorFlow

Sequential model

> The Sequantial model is a linear stack of layers. The layers are added one-by-one sequentially.

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(8,)))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])

model.fit(X_train, y_train, epochs=5, batch_size=128)
```

Compiling

Optimizer

> The optimizer determines how we adjust the parameters (weights and biases) of our model based on the gradients of the loss with respect to those parameters. The optimizer's job is to minimize the loss.

Loss

> The loss function is a measure of how well the model's predictions match the true labels. The goal is to minimise the loss.

Metrics

> Used to monitor and evaluate the performance of the model during training and testing. These metrics are **NOT** used during the optimization process but are simply for observation.

Layers

Dense

> A dense layer is a **fully connected** layer, meaning that each neuron in the layer is connected to all neurons in the previous layer and all neurons in the subsequent layer.
>
> ```
> keras.layers.Dense(NEURONS, activation=FUNCTION)
> ```
>
> Here is an example of a Dense layer. $NEURONS$ is the number of neurons in the layer, and $FUNCTION$ is the activation function of these neurons.

Dropout

```
keras.layers.Dropout(FRACTION)
```

Here is an example of Dropout. $FRACTION$ is how many (randomly chosen) of the neurons should be ignored during training. This helps prevent overfitting.

Flatten

Used to Flatten the input. For instance if the input is a multi-dimensional tensor (like an image), it is flattened to a one-dimensional tensor (vector).

$$\text{Flatten}(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}) \Rightarrow [a, b, c, d, e, f, g, h, i]$$

Conv2D

2D convolutional layer. Performs a convolution-operation on the inputted data using a set of learnable filters (or kernels) to produce feature maps which highlight certain features in the inputted data.

```
keras.layers.Conv2D(filters=32, kernel_size=(3,3))
```

Stride

The stride size can be specified when convolving. This means the step size wiht which the convolutional filter is moved across the input data (both in terms of width and height). For example, if the stride is set as $(2, 2)$, then the filter jumps two pixels at a time. This results in a smaller output feature map.

$$\text{output height} = \frac{\text{input height} - \text{filter height} + 2 \times \text{padding}}{\text{stride height}} + 1$$

$$\text{output width} = \frac{\text{input width} - \text{filter width} + 2 \times \text{padding}}{\text{stride width}} + 1$$

MaxPooling2D

Used to reduce the spatial dimensions of the feature maps, which is useful in order to reduce the number of parameters which in turn leads to a more efficient training process and reduced risk of overfitting.

```
keras.layers.MaxPooling2D(pool_size=(2, 2))
```

In this exaple, the MaxPooling is applied to a 2×2 grid at a time, therefore reducing the feature map by 2.

Padding

Padding adds (often zeros) to the edges of the input, in order to control the spatial dimension of the output when for instance convolving.

```
model.add(Conv2D(filters=32, kernel_size(3,3), padding='valid')

model.add(Conv2D(filters=32, kernel_size(3,3), padding='same')

model.add(ZeroPadding2D(padding=(2,2))
```

where *valid* means no padding and *same* means to make the output shape the same as the input shape.

Batch Normalization

> Standardize the inputs (batch) to a layer. This is done to improve the gradient descent, by keeping the inputs stationary (and relatively low). This can in turn let us be able to have higher learning rates without making the network unstable. Normalization therefore acts as a sort of regulatization technique.

```
model.add(keras.layers.BatchNormalization())
```

Training

Batch gradient descent

> For batch gradient descent, the whole dataset is used when computing the gradient of the loss function. In this context, "batch" refers to the entire training set. The computational expense therefore increases with the size of the dataset.

Mini-batch gradient descent

> In order to overcome the computational cost of using the whole dataset, mini-batch is introduced. Here, the training set is split into smaller subsets ("mini-batches"). For every iteration, a single mini-batch is used to compute the loss and gradient. Over one epoch (complete pass through the entire dataset), each mini-batch is used once.

Stochastic gradient descent

> For stochastic gradient descent, each sample is used for calculation of loss and gradient. This single operation is faster compared to using multiple samples at a time, but lead to variable results due to the variability in the data.

Comparison

> Over the course of an entire training process, both methods will process each sample in the dataset. However, the way they process these samples leads to differences in computational efficiency.
>
> In SGD, the frequent but less computationally intensive updates can lead to faster convergence in the early stages of training.
>
> In Batch Gradient Descent, the updates are less frequent but each requires more computation, as the entire dataset influences each update.

Momentum

> A technique used to overcome local minima, is to use the momentum of the gradient descent. This means that previous gradient vectors are used in future gradients.

■

Convolutional Neural Networks (CNN)

Used for image processing. As the name states, the networks involve convolutional layers, in order to extract meaningful features of the inputs.

Example

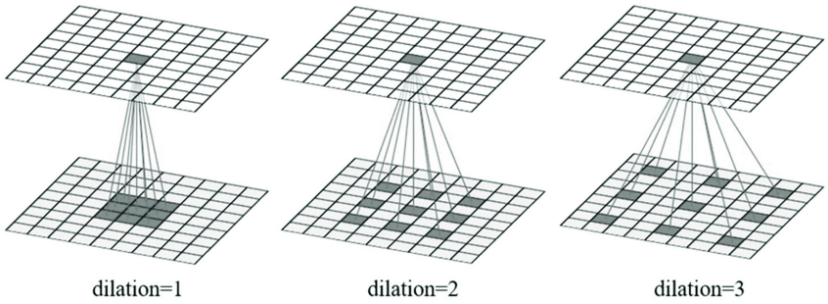
```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu',
    input_shape=(64,64,3)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Stride

A convolutional filter usually convolves with a step size of 1. For a 2x2 filter convolving over an image, this means that it starts at the top-left corner, and moves one pixel to the right or down for every step. If the stride size had been for instance 2, it would move two pixels for every step.

Dilated/sparse

A dilated convolutional filter acts in the same way as a regular filter, but adds "zeros" in-between in its receptive field.



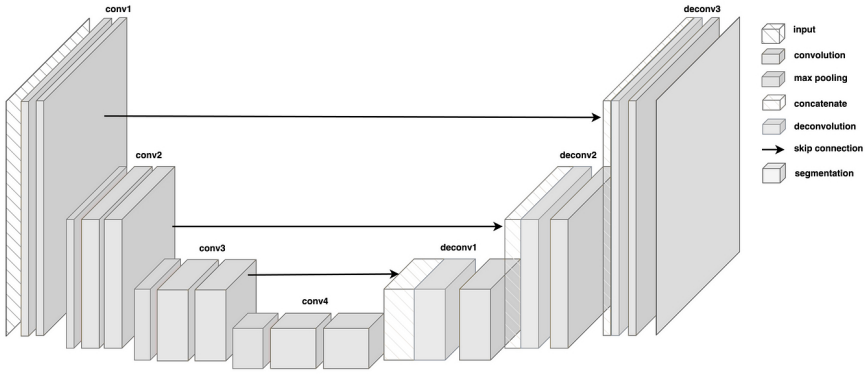
Output size and receptive field (input size)

$$O = \frac{I - K + 2P}{S} + 1 \qquad I = O \times S + (K - S) - 2 \times P$$

- **O** Output size (1D)
- **I** Input size (1D)
- **K** Kernel size (1D)
- **P** Padding
- **S** Stride

U-Net

The basic idea is that the input is concatenated to the upscaled output, in order to retain as much information as possible.
The architecture consists of a contractive path, a bottleneck, and an expansive path; the U-shape.



Contraction

Consists of convolutional layers followed by max-pooling layers. This path captures the context of the image.

```
inputs = Input(input_shape)

c1 = Conv2D(64, (3, 3), activation='relu', padding='same')(
    inputs)
p1 = MaxPooling2D((2, 2))(c1)

c2 = Conv2D(128, (3, 3), activation='relu', padding='same')(p1)
p2 = MaxPooling2D((2, 2))(c2)

# Bottleneck
c3 = Conv2D(256, (3, 3), activation='relu', padding='same')(p2)
```

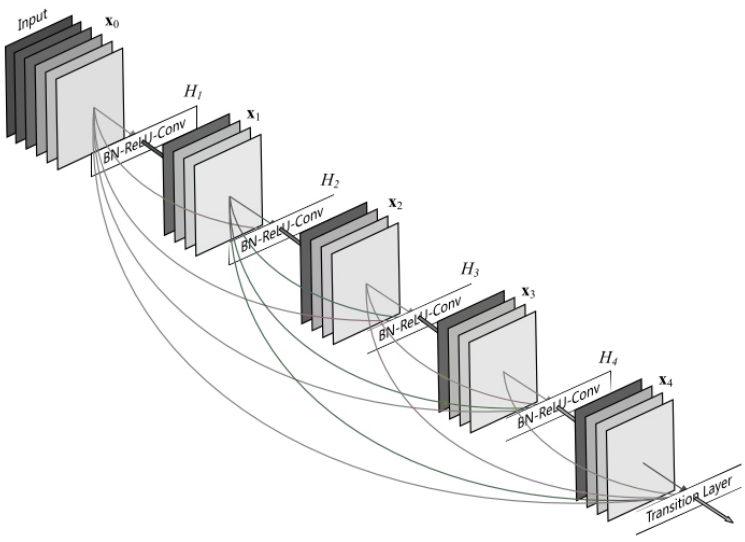
Expansion

Upscales the input, and concatinates the upscaled output with the corresponding part of the contraction-path.

```
u1 = UpSampling2D((2, 2))(c3)
c4 = Conv2D(128, (3, 3), activation='relu', padding='same')(
    concatenate([u1, c2]))

u2 = UpSampling2D((2, 2))(c4)
c5 = Conv2D(64, (3, 3), activation='relu', padding='same')(
    concatenate([u2, c1]))

outputs = Conv2D(1, (1, 1), activation='sigmoid')(c5)
```

ResNet (Residual Network)

Introduces shortcuts in the model architecture. The idea behind these shortcuts is to diminish the vanishing gradient problem, by allowing the activation from one layer to bypass one or more layers and be summed up with the actvation of a later layer ("*skip connection*").
ResNet architectures are often used with models that have a lot of layers.



In the following example, a residual block is created where the input-tensor is added to the output after two convolutional layers.

```
def residual_block(input_tensor, kernel_size, filters):
    x = Conv2D(filters, kernel_size, padding='same')(input_tensor)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(filters, kernel_size, padding='same')(x)
    x = BatchNormalization()(x)

    x = add([x, input_tensor])
    x = Activation('relu')(x)

    return x

input = Input(shape=(64, 64, 3))
x = Conv2D(64, (3, 3), padding='same')(input)
x = residual_block(x, (3, 3), 64)
x = residual_block(x, (3, 3), 64)
output = Conv2D(3, (3, 3), padding='same')(x)
```

Other architectures

- **LeNet**: Pretrained CNN.
- **Inception**: Pretrained CNN consisting of multiple "sub-networks".

Generative Models

Generative models are a subclass of machine learning models primarily focused on generating new data that is similar to the training data. They have applications in a variety of fields including image synthesis, natural language processing, and more.
The idea is that we assume our observations (train-dataset) follows an unknown distribution. By mimicing this distribution, we can therefore generate samples that appear to follow the same distribution as our

observations as well as being different from the original observations.

Autoencoders (AE)

> Autoencoders are a type of neural networks designed to learn compressed representations of data, and to reconstruct the inputs. They work by taking an input, encoding it into a lower-dimensional space, and then decoding it back to reconstruct the original input as closely as possible with a **deterministic approach**. The encoder captures the most important features of the input data, while the decoder uses this encoded form to rebuild the data. Autoencoders are commonly used for dimensionality reduction, feature learning, and data reconstruction.

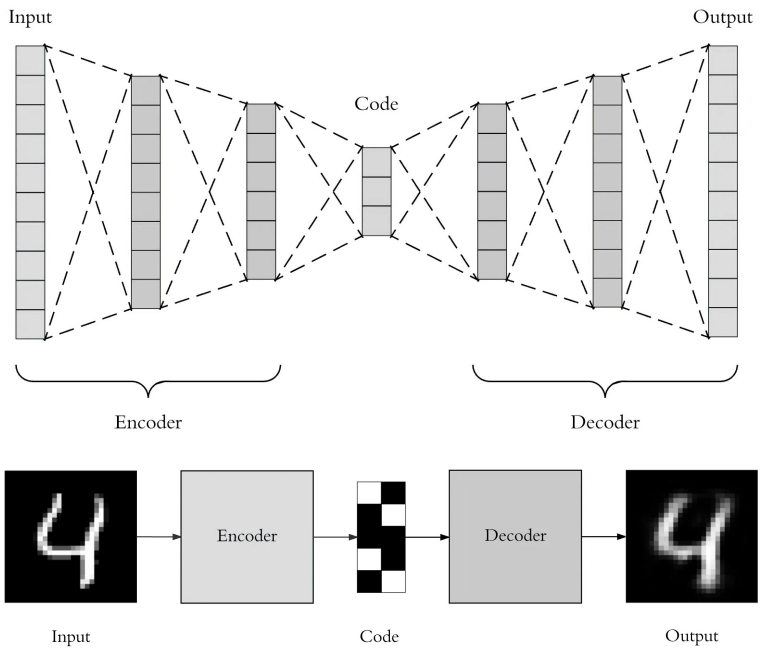
Encoder

> Compresses the input into a latent-space representation[1].

Decoder

> Reconstruction of the encoded input-data.

```
input = Input(shape=(INPUT_DIMENSION,))

encoder = Dense(LATENT_DIMENSION)(input)
decoder = Dense(INPUT_DIMENSION)(encoder)

autoencoder = Model(input, decoder)
```



Variational Autoencoders (VAE)

[1]Latent space: abstract multi-dimensional space.

Variational Autoencoders are a **probabilistic** extension of autoencoders. Instead of mapping an input directly to a point in the latent space, VAEs map it to a distribution. This adds a layer of stochasticity, making the model capable of generating new data that's similar to the training data.

Encoder

Instead of compressing the input to a single point in the latent-space, variational autoencoders encodes the input to parameters of a probability distribution (usually Gaussian). I.e. a mean vector $\vec{\mu}$ and a standard deviation vector $\vec{\sigma}$, which make up a distribution z.

Decoder

When decoding, a sample is drawn from the latent-space distribution to generate the reconstruction.

Reparameterization

Sample from the latent-space and at the same time allowing for backpropagation introduces a random constant that the $\vec{\sigma}$ is scaled by.

$$\vec{z} = \vec{\mu} + \vec{\sigma} \times \vec{\epsilon}$$

where z is the sampled latent-vector, and $\vec{\epsilon}$ is a randomly normally sampled vector $\vec{\epsilon} \sim \mathcal{N}(0, 1)$.
The equation above is often replaced by

$$\vec{z} = \vec{\mu} + \exp\left(\frac{\log(\vec{\sigma^2})}{2}\right) \times \vec{\epsilon}$$

making sure that the scale parameter is non-negative, which is a requirement for a standard deviation in a Gaussian distribution.

```
# Simplified VAE. Does not include loss etc.

class VAE(tf.keras.Model):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder  # Encoding neural network
        self.decoder = decoder  # Decoding neural network

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=mean.shape)
        return eps * tf.exp(logvar * 0.5) + mean

    def call(self, x):
        mean, logvar = self.encoder(x)
        z = self.reparameterize(mean, logvar)
        return self.decoder(z)
```

Objective function

The objective function, also known as the loss function, for VAEs is the sum of the reconstruction loss and the regularization term. Mathematically,

$$\mathcal{L} = \text{reconstruction loss} + \text{regularization term}$$

This function ensures that the model not only reconstructs the data well but also that the latent space has good properties that enable data generation.

Kullback-Leiber divergence

The regularization term is calculated by the Kullback-Leiber divergence, which is a measure of how one probability diverges from a second.

$$\text{KL}\left[\mathcal{N}(\mu, \sigma) || \mathcal{N}(0, 1)\right] = \frac{1}{2} \sum \left(1 + \log(\sigma^2) - \mu^2 - \sigma^2\right)$$

Applications

Both *AE* and *VAE* are used to compress the original data whilst retaining the maximum amount of information (variance). Can therefore remove the noise in the data, by filtering out unwanted information.
VAE's can also be used to generate new data samples, based on their probabailistic approach.

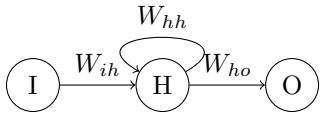
Recurrent Neural Networks (RNN)

Recurrent Neural Networks are a type of neural network designed for sequence-based data. Unlike traditional feedforward neural networks, RNNs have connections that loop back within the network, providing a way to maintain internal state.

Basic structure

A basic RNN has an input layer, a hidden recurrent layer, and an output layer. The recurrent layer processes sequences of data by looping its output back into its input, allowing it to learn from past information.

$$\begin{aligned} h_t &= f_W(h_{t-1}, x_t) \\ h_t &= \tanh(W_{hh} h_{t-1} + W_{xh} x_t + b_h) \\ y_t &= f_{\text{activation}}(W_{hy} h_t + b_y) \end{aligned}$$



Where x is the input and h the hidden layer. h_{t-1} represents the hidden output for the previous time-step.

Vanishing and Exploding gradients

RNNs are susceptible to the vanishing and exploding gradient problems, making it difficult to train them on long sequences. Techniques like gradient clipping and architectures like LSTM and GRU are often used to mitigate these issues.

Long Short-Term Memory (LSTM)

LSTM is a special type of RNN that can learn long-term dependencies in the data. It has a more complex internal structure involving gates that control the flow of information.
The LSTM structure consists of four gates, which combine or remove information. The operations done are linear, being

- $\oplus$ Element-wise addition. $\begin{bmatrix}0.8\\0.8\\0.8\end{bmatrix} \oplus \begin{bmatrix}1.0\\0.5\\0.0\end{bmatrix} = \begin{bmatrix}0.8+1.0\\0.8+0.5\\0.8+0.0\end{bmatrix} = \begin{bmatrix}1.8\\1.3\\0.8\end{bmatrix}$
- $\otimes$ Element-wise multiplication. $\begin{bmatrix}0.8\\0.8\\0.8\end{bmatrix} \otimes \begin{bmatrix}1.0\\0.5\\0.0\end{bmatrix} = \begin{bmatrix}0.8\cdot 1.0\\0.8\cdot 0.5\\0.8\cdot 0.0\end{bmatrix} = \begin{bmatrix}0.8\\0.4\\0.0\end{bmatrix}$

By inspecting these operations, we can see that th $\otimes$ gate is able to either block or allow information through (by being 0.0 or 1.0) or something in-between. This means, that the network can learn previous state values, and take these into account when filtering values of new inputs.
A LSTM network has a **cell state** C, which acts as the "memory" of the network. This state is being transferred across the time-steps.

Γ_f Forget gate layer

The first step in an LSTM is the "forget gate layer". This is a neural network which takes in the previous output along with current input. This layer has a sigmoid activation function, where inputs resulting in 0's lead to variables being left out of the cell state C.

$$\Gamma_f = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \qquad\qquad C_t^* = \Gamma_f \otimes C_{t-1}$$

Γ_u Update/input gate layer

The next step is to decide what new information to store in the cell state.

$$\Gamma_u = \sigma(W_u \cdot [h_{t-1}, x_t] + b_u) \qquad\qquad \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

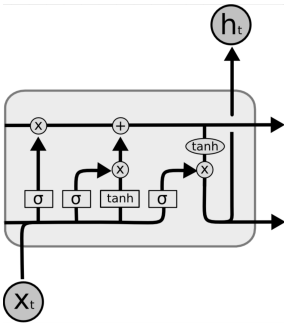
These two are then pairwise multiplied together, and added to the forgotten state.

$$\begin{aligned}\textbf{CELL STATE} \quad C_t &= C_{t-1}^* \oplus (\Gamma_u \otimes \tilde{C}_t)\\ &= (\Gamma_f \otimes C_{t-1}) \oplus (\Gamma_u \otimes \tilde{C}_t)\end{aligned}$$

Γ_o Output gate layer

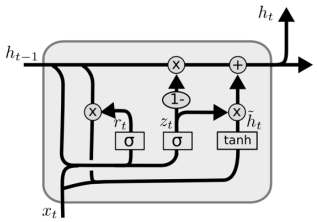
The output of the model is then calculated based on both the cell state and previous ouput as well as input.

$$\begin{aligned}&\Gamma_o = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)\\ \textbf{OUTPUT} \quad &h_t = \Gamma_o \otimes \tanh(C_t)\end{aligned}$$



Gated Recurrent Unit (GRU)

GRU is a variant of RNN that aims to solve the vanishing gradient problem. It is simpler than LSTM and computationally more efficient but may not perform as well on tasks requiring the modeling of long-term dependencies.
The GRU combines the forget and input gates into a single "update gate". It also merges the cell state and hidden state (previous predictions) along with some other changes.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Applications

RNNs are widely used in various sequence-based tasks like language modeling, machine translation, speech recognition, and time-series forecasting.

Sentiment Analysis

Subset of Natural Language Processing (NLP) that focuses on determining the sentiment expressed, to classify whether it for instance is positive, neutral or negative. (Or more advansed emotions.)

Continuous bag of words (CBOW)

Does **not** consider the ordering of the words. One word is predicted at a time (given the context words). Surrounding words is used to predict a target word. Takes an input of one-hot-encoded vectors fed to a network which predicts the probability distribution for the target word.
Given the sentence "The cat ___ on the mat", the context words are then ["the", "cat", "on", "the", "mat"] – which is first transformed to vectors. The CBOW model then tries to predict a word given this context (without knowing where the predicted word should be).

Skip-Gram

Opposite to CBOW, Skip-Gram is given a target word and predicts the context (e.g. surrounding words). For the sentence "The cat sat on the mat", given the word "sat", Skip-Gram will try to predict the context words "The", "cat", "on", "the", "mat".
The input layer takes the one-hot encoded vector of the target word. This is passed through the network and the output layer gives multiple probability distributions, each predicting a word in the context.

Window size

The window size (parameter) determines how many words the model should predict. E.g. if the window size is 2, the model predicts two words before the inputted word and two after.

Embedding (vectorisation of words)

Word2Vec

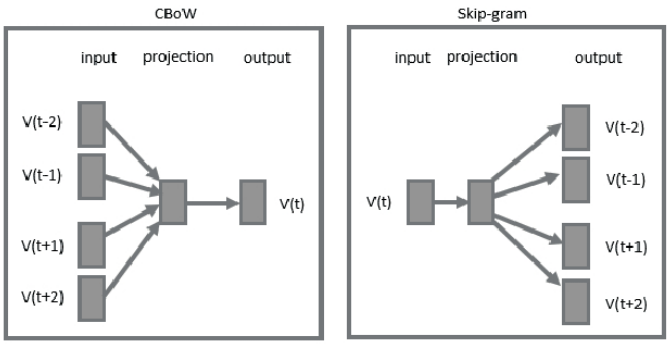
Neural network-based technique of producing dense word embeddings, transforming words into a continuous vector space. The training-process is based on **either** CBOW or Skip-Gram.

CBOW

When training on the sentence "The cat sat on the mat", we choose for instance "sat" as the target, and input the rest to CBOW – which will then try to predict "sat". Therefore, the model learns to generate word vectors such that the average of context word vector can be used to predict the target word vector.

Skip-Gram

In the same (but opposite) way, we can use the Skip-Gram method to predict the context words based on the target. The model therefore also learns the context of the word when creating the embedding.



Term Frequency (TF)

This measures how frequently a term occurs in a document. It's calculated by dividing the number of times the term appears in the document by the total number of terms in the document. The idea is that the more frequently a term appears in a document, the more important it is for that document.

$$\mathrm{TF}(\mathrm{word}, \mathrm{document}) = \frac{\sum \mathrm{word}}{\sum \mathrm{words\ in\ document}}$$

Inverse Document Frequency (IDF)

This measures how important a term is across a set of documents. It's calculated by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient. This helps to diminish the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely.

$$\mathrm{IDF}(\mathrm{word}, \mathrm{documents}) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents with word}}\right)$$

TF-IDF score

This is simply the product of TF and IDF. A high TF-IDF score indicates a term is important both in the document and within the entire document set.

$$\text{TF-IDF}(\text{word}, \text{document}, \text{documents}) = \text{TF}(\text{word}, \text{document}) \times \text{IDF}(\text{word}, \text{documents})$$

Example sentiment analysis

RNNs are good at capturing sequence information, making them well-suited for NLP tasks, including sentiment analysis. LSTM (Long Short-Term Memory) is a special kind of RNN that can remember long sequences, usually achieving better performance.

```
txt = {'I love you': 1, 'This is awful': 0,
       'I feel great': 1, 'This is terrible': 0}

tokenizer = Tokenizer(num_words=100,
                      oov_token="<OOV>").fit_on_texts(txt.keys())
padded = pad_sequences(tokenizer.texts_to_sequences(txt.keys()),
                       maxlen=5)

X_train, X_test, y_train, y_test = split(padded, txt.values(),
                                         test_size=0.2)

model = Sequential([Embedding(100, 16, input_length=5),
                    LSTM(32), Dense(1, activation='sigmoid')])

model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy'])
```

■

Trainable parameters

ANN

$$\begin{aligned} \textit{parameters} &= \textit{input shape} \times \textit{output shape} + \overbrace{\textit{output shape}}^{\text{bias}} \\ &= (\textit{input shape} + 1) \times \textit{output shape} \end{aligned}$$

CNN

$$\begin{aligned} \textit{parameters} &= \textit{filter area} \times \textit{input channels} \times \textit{output channels} + \textit{output channels} \\ &= (\textit{filter area} \times \textit{input channels} + 1) \times \textit{output channels} \end{aligned}$$

$$\textit{output size} = \frac{\textit{input size} - \textit{kernel size} + 2 \times \textit{padding}}{\textit{stride}} + 1$$

RNN

$$\begin{aligned} parameters &= \overbrace{input\_dim \times units}^{\text{ih}} + \overbrace{units}^{\text{bias}} + \overbrace{units \times units}^{\text{hh}} \\ &= units \times (input\_dim + units + 1) \end{aligned}$$

LSTM **Four gates**

$$\begin{aligned} parameters &= 4 \times (input\_dim \times units + units + units \times units) \\ &= 4 \times units \times (input\_dim + units + 1) \end{aligned}$$

GRU **Three gates**

$$\begin{aligned} parameters &= 3 \times (input\_dim \times units + units + units \times units) \\ &= 3 \times units \times (input\_dim + units + 1) \end{aligned}$$

Bidirectionality

For bidirectional models, the number of parameters is doubled.

■

Example calculations

CNN

KERNEL = (5, 5) **STRIDE = 1** **PADDING = 0** **(32, 32, 3) → (28, 28, 8)**

$$\begin{aligned} parameters &= 5^2 \times 3 \times 8 + 8 \\ &= (5^2 \times 3 + 1) \times 8 \\ &= 608 \end{aligned}$$

$$\begin{aligned} output\ size &= \frac{32 - 5 + (2 \times 0)}{1} + 1 \\ &= 28 \end{aligned}$$

$$\begin{aligned} receptive\ field &= 28 \times 1 + (5 - 1) - 2 \times 0 \\ &= 28 + 4 \\ &= 32 \end{aligned}$$

KERNEL = (5, 5) STRIDE = 1 PADDING = 2 (32, 32, 3) → (32, 32, 8)

$$
\begin{aligned}
\textit{parameters} &= 5^2 \times 3 \times 8 + 8 \\
&= (5^2 \times 3 + 1) \times 8 \\
&= 608
\end{aligned}
$$

$$
\begin{aligned}
\textit{output size} &= \frac{32 - 5 + (2 \times 2)}{1} + 1 \\
&= 32
\end{aligned}
$$

$$
\begin{aligned}
\textit{receptive field} &= 32 \times 1 + (5 - 1) - 2 \times 2 \\
&= 32 + 4 - 4 \\
&= 32
\end{aligned}
$$

KERNEL = (2, 2) STRIDE = 2 PADDING = 2 (32, 32, 3) → (18, 18, 8)

$$
\begin{aligned}
\textit{parameters} &= 2^2 \times 3 \times 8 + 8 \\
&= (2^2 \times 3 + 1) \times 8 \\
&= 104
\end{aligned}
$$

$$
\begin{aligned}
\textit{output size} &= \frac{32 - 2 + (2 \times 2)}{2} + 1 \\
&= 18
\end{aligned}
$$

$$
\begin{aligned}
\textit{receptive field} &= 18 \times 2 + (2 - 2) - 2 \times 2 \\
&= 36 - 4 \\
&= 32
\end{aligned}
$$

RNN

```
units = 64
input_dim = 10

model = Sequential()
model.add(layers.SimpleRNN(
    units,
    dropout=0.2,
    recurrent_dropout=0.2,
    input_shape=(None, input_dim)
))
```

$$
\begin{aligned}
parameters &= 10 \times 64 + 64 + 64^2 \\
&= 64 \times (10 + 64 + 1) \\
&= 4.800
\end{aligned}
$$

LSTM

```
                    units = 64
                    input_dim = 10

                    model = Sequential()
                    model.add(layers.LSTM(
                        units,
                        dropout=0.2,
                        recurrent_dropout=0.2,
                        input_shape=(None, input_dim)
                    ))
```

$$
\begin{aligned}
parameters &= 4 \times (10 \times 64 + 64 + 64^2) \\
&= 4 \times 64 \times (10 + 64 + 1) \\
&= 19.200
\end{aligned}
$$