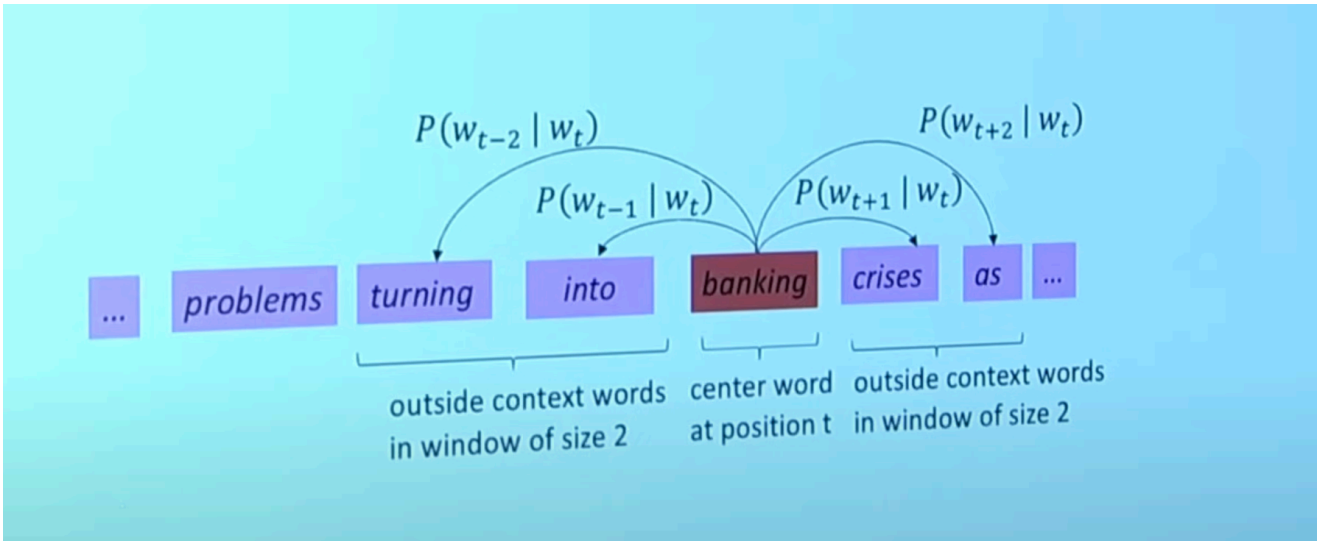
word embeddings

random notes to make sense

- converting words into numbers i.e vectors
- similar words which have similar meaning will have similar numbers
- each word should have multiple numbers so different number can represent different context of same word like singular, plural, positive, negative
- words are represented by multidimensional vector
- similar words will have similar vectors
- similarity can be measured by dot product of two vectors. if two vectors have large dot product that mean they are similar
- we use context to decide the meaning of words so words that come into same context will have similar meaning and hence they will also be closer in vector space
- *corpus of words* is means long list of words
- we will go through each word in text we will assume that word itself as C and word after it as context O
- will find probability of O given C and vice versa and will keep adjusting word vectors so we can increase this probability



word embeddings (from origin)

so for llm's to understand language we need to convert human language into medium that llm's can understand. that medium is numbers and math.

so we start with assigning random unique numbers to each word.
for example lets take three sentence :

1. this movie is great

2. this movie is trash
3. this movie is best

lets give them numbers

this - 1
movie - 2
is - 3
great - 10
trash - 11
best - 20

now we did assigned unique number to each word but if we look at numbers we can see that great and trash are very close mathematically and model can misunderstood it as they have similar meaning and best even tho has similar meaning is further away mathematically from great.

also performing any mathematical operation on this numbers don't give any semantically meaningful results.

also another problem is in real word our vocabalry very huge and each word can have different meanings. only one number can not describe the different meaning of word.

so to deal with this problem we come up with new solution which is **one hot encoding method**

Imagine our same three sentences:

1. this movie is great
2. this movie is trash
3. this movie is best

we have six unique word in total {this, movie, is, great, trash, best}
in one hot encoding we assign vector to each word. length of this vector is equal to number of unique word in our vocabulary in above case its 6.
for each word, we place a '1' at the index corresponding to that word and '0' everywhere else.

Let's see how our words would be represented using one-hot encoding:

- this: [1, 0, 0, 0, 0, 0]
- movie: [0, 1, 0, 0, 0, 0]
- is: [0, 0, 1, 0, 0, 0]
- great: [0, 0, 0, 1, 0, 0]
- trash: [0, 0, 0, 0, 1, 0]
- best: [0, 0, 0, 0, 0, 1]

one hot encoding is better than randomly assigning number because it solves the problem of mistakenly thinking two word are similar like in previous method our model could've misunderstood that great and trash is similar but after applying one hot encoding method we can see that they are not similar (because their dot product will be zero)

but the problem with this method is it represents each word independently and does not understand any semantic meaning between words.
for example words 'great' and 'trash' are distant but words 'great' and 'best' are also distinct on similar level. it does not capture a meaning that word 'great' and 'best' are closer and have similar semantic meaning.

another big problem is dimensionality. in real world we have tens of thousands of words and hence we will have to assign each word a vector of tens of thousands of length.
this is computationally inefficient, making it difficult to process and store these representations, especially for large datasets

No Ordinal Information: Similar to the random number approach, one-hot encoding doesn't capture any information about the order or position of words in a sentence.

so in short it solves the problem of misunderstanding meaning of words but it does not solve problem of understanding meaning of word. it represent each word independently. and also increases computational cost.

so hence we move on to new method called **word embeddings**

The core idea behind word embeddings is to represent each word as a dense vector of real numbers in a relatively low-dimensional space (compared to the size of the vocabulary in one-hot encoding).

These vectors are learned from large amounts of text data in such a way that words that appear in similar contexts or have similar meanings are located close to each other in this vector space.

Each dimension in the vector is thought to capture some latent (hidden) aspect of the word's meaning

Let's revisit our example sentences:

1. this movie is great
2. this movie is trash
3. this movie is best

Instead of a long, sparse one-hot vector for each word, with word embeddings, each word might be represented by a much shorter, dense vector, say of 50 or 300 dimensions. For instance:

- this: [0.1, -0.5, 0.8, 0.2, -0.3, ...] (50 numbers)

- movie: [0.9, 0.3, -0.1, 0.7, 0.4, ...]
- is: [-0.2, 0.6, 0.5, -0.4, 0.1, ...]
- great: [0.4, 0.9, 0.2, 0.6, 0.8, ...]
- trash: [-0.7, -0.8, 0.1, -0.5, -0.9, ...]
- best: [0.5, 0.8, 0.3, 0.7, 0.9, ...]

the **process** of learning these representations/embeddings

- choose unique vocabulary :
  - we have vocabulary of 6 unique words {this, movie, is, trash, great, best} so we have 6 inputs.
- then we create training data
  - it contains context (input) and target(output).
  - to create context we chose size of context window (e.g., a window of 5 words means 2 words before and 2 words after the target word form the context)
  - For each window, you create a training sample where the input is the one-hot encoded vectors of the context words, and the output is the one-hot encoded vector of the target word.
  - For example, in the sentence "this movie is great and best", with a window size of 3 (1 before, 1 after):
    - Context: [this], [is] -> Target: [movie]
    - Context: [movie], [great] -> Target: [is]
    - Context: [is], [and] -> Target: [great]
    - ... and so on.
- network architecture:
  - input layer
    - it takes context as inputs. if our context window size is C our input layer will contain C inputs words each word represented with one hot encoded vector
  - embedding layer
    - the input (usually a one-hot encoded vector) is directly multiplied by a weight matrix. This weight matrix is the embedding layer(weight matrix is of $v * d$ where v is number of words we have and d is length of vector of each word). Each row of this weight matrix corresponds to the learned embedding vector of a word in the vocabulary (starting with random numbers). When a one-hot vector is multiplied by this matrix, it effectively selects the row corresponding to that word.
    - So, if we have C context words as input, the output of the embedding layer will be C embedding vectors, each of size D.
  - Average Pooling Layer (or Summation):
    - To get a single representation of the context, we typically average (or sum) the embedding vectors of all the context words. If we have C context embeddings

(each of size D), the output of this layer will be a single D-dimensional vector representing the combined context.

- outptut layer:
  - It has a weight matrix W_{output} of size $D \times |V|$ where D is dimension of vector and V is size of our vocabulary i.e number of words in vocabulary
  - The output of this layer, after the softmax function, is a probability distribution over the |V| words in the vocabulary. The goal is for this probability distribution to have a high probability for the actual target word.

we train this model using cross entropy loss function and backpropagation. and get the word embeddings

paper

- previously we used to use N-gram models to understand words. they were simple systems
- but then with neural networks we can create better representations of words
- words can have multiple **degrees of similarity. similarity** not only means they are similar in meaning like 'man' and 'woman' represent gender similarity but they can be also similar in a way that they are nouns showing grammatical similarity. they can also be similar in showing opposition and difference between them can be same as difference between 'rich' and 'poor'

different models architectures to create embeddings of words

1. Feedforward Neural Net Language Model (NNLM):

   ai explanation

   Imagine we want our NNLM to predict the next word in a sequence.

   Step 1: Input - The Context

   - The model starts with a sequence of preceding words as its input. This sequence is often called the "context window."
   - **Example:** If our context window size is 2, and we want to predict the word after "the quick," the input to the model would be the sequence ["the", "quick"].

   Step 2: Tokenization and Vocabulary Lookup

   - Each word in the input sequence is first converted into a numerical representation. This involves:
     - **Tokenization:** Splitting the raw text into individual words (tokens).
     - **Vocabulary Lookup:** Mapping each token to a unique index in the model's vocabulary. The vocabulary is a predefined list of all the words the model

knows.

- **Example:** Assuming our vocabulary contains "the" at index 5 and "quick" at index 12, the input might be represented as the sequence of indices [5, 12].

Step 3: Embedding Layer - From Indices to Vectors

- Each word index is then used to retrieve its corresponding **embedding vector** from an **embedding matrix**. This matrix is learned during the training process and represents each word in a dense, low-dimensional space. Words with similar meanings or that appear in similar contexts tend to have embedding vectors that are close to each other in this space.
- **Example:**
  - The index 5 ("the") is used to look up the 5th row in the embedding matrix, resulting in an embedding vector $\mathbf{v}_{\text{the}}$ (e.g., [-0.2, 0.5, -0.1, 0.8]).
  - The index 12 ("quick") is used to look up the 12th row, resulting in an embedding vector $\mathbf{v}_{\text{quick}}$ (e.g., [0.7, -0.3, 0.9, 0.1]).

Step 4: Processing the Input Embeddings - Hidden Layers

- The embedding vectors of the context words are then fed into one or more **hidden layers** of the neural network.
- **Concatenation (Simple Case):** In simpler NNLMs, the embedding vectors might be concatenated into a single, larger input vector.
  - **Example:** $[\mathbf{v}_{\text{the}}; \mathbf{v}_{\text{quick}}] = [-0.2, 0.5, -0.1, 0.8, 0.7, -0.3, 0.9, 0.1]$.
- **Neural Network Layers:** This combined vector is then passed through the hidden layers. Each hidden layer performs a series of linear transformations (matrix multiplications with weight matrices and addition of bias vectors) followed by a non-linear activation function (like $\tanh$, ReLU, etc.). These layers learn complex relationships and patterns in the input context.
  - **Example (First Hidden Layer):** $\mathbf{h}_1 = \text{activation}(W_1[\mathbf{v}_{\text{the}}; \mathbf{v}_{\text{quick}}] + \mathbf{b}_1)$, where W_1 is the weight matrix and $\mathbf{b}_1$ is the bias of the first hidden layer. Subsequent hidden layers would process the output of the previous layer in a similar way.

Step 5: Output Layer - Generating Raw Scores (Logits)

- The output from the final hidden layer is then fed into the **output layer**. This layer is a linear layer that produces a vector of raw scores, called **logits**.
- **Key Point:** The size of this logits vector is equal to the size of the vocabulary. Each element in the vector corresponds to a word in the vocabulary and represents the model's prediction of how likely that word is to be the next word.
- **Example:** $\mathbf{o} = U\mathbf{h}_{\text{final}} + \mathbf{c}$, where U is the weight matrix connecting the final hidden layer to the output layer, $\mathbf{h}_{\text{final}}$ is the output of the final hidden layer, and $\mathbf{c}$ is the output bias vector. The resulting $\mathbf{o}$ might look like:

$$\mathbf{o} = \begin{bmatrix} 1.5 \\ -0.3 \\ 2.1 \\ \vdots \\ -0.8 \end{bmatrix}$$

where the first element corresponds to the first word in the vocabulary, the second to the second word, and so on.

Step 6: Softmax Activation - Converting Scores to Probabilities

- The logits vector is then passed through the **softmax function**. The softmax function converts these raw scores into a probability distribution over the vocabulary.
- **Properties of Softmax Output:**
  - Each probability is between 0 and 1.
  - The sum of all probabilities in the output vector is equal to 1.
- **Example:** Applying softmax to our logits vector $\mathbf{o}$ would give us a probability distribution $\hat{y}$:

  $\hat{y} = \text{softmax}(\mathbf{o})$, where $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{|V|} e^{z_j}}$

  The resulting probability distribution $\hat{y}$ might look like:

  $$\hat{y} = \begin{bmatrix} 0.15 \\ 0.04 \\ 0.28 \\ \vdots \\ 0.02 \end{bmatrix}$$

  This means the model predicts the first word in the vocabulary has a 15% chance of being the next word, the second word has a 4% chance, the third word has a 28% chance, and so on.

Step 7: Making a Prediction (Optional)

- Based on the probability distribution, we can make a prediction for the next word. The simplest way is to choose the word with the highest probability.
- **Example:** If the highest probability in $\hat{y}$ is 0.28, and this corresponds to the word "brown" in our vocabulary, then the model's prediction for the next word after "the quick" would be "brown".

Training the NNLM (The Learning Part):

It's important to remember that the weight matrices in the embedding layer and all the hidden and output layers are not fixed initially. They are learned during the **training process**. This involves:

1. **Feeding a large corpus of text** to the model in sequences.
2. **Comparing the model's predicted probability distribution** for the next word with the actual next word in the training data (using a **loss function** like cross-entropy).
3. **Adjusting the weights and biases** of the network using an optimization algorithm (like gradient descent) to minimize this loss, making the model's predictions more accurate over time.

Through this training process, the NNLM learns the statistical relationships between words and develops meaningful word embeddings that capture semantic and syntactic information.

low effort summery

```
so according to this nnlm first converts its text data into part first being context and second being target we convert context into one-hot encoding vector and then multiply it with embedding matrix which is randomly initiated weight matrix (which is what we actually want to learn) and then pass them through linear and nonlinear layers then ouput layer and softmax layer and predict probability of target given context and then we check if we match the probability and how wrong we are using loss function and then using back propagation we change weights and we get embedding matrix. also i had one question when we give context words to our model then it predicts probability of each word in vocabulary right i mean it gives probability of each index in vocabulary and we make sure that context words get such weights that they can give high predictibility for right index of word
```

2. Recurrent Neural Net Language Model (RNNLM)

   they have less complexity then nnlm's because they have ability to time delayed connection
   so this is basically attempt to reduce the computational complexity that nnlm have to deal with it uses recurrent neural network architecture instead of simple feed forward neural network and successfully reduces complexity of output layer

in the paper they introduce two new architectures

here we use following approach -> first we train a simpler model (a neural network with less complexity like less hidden layers) on our text data to learn the word embeddings.
and then we use those word embeddings to train more advanced model like n-gram NNLM.
i.e we first pretrain word embeddings on simpler model and then use those word embedding to capture even more relationships between data.

1. Continuous Bag-of-Words Model

   The CBOW model aims to **predict a target word given its surrounding context words**.

- we take a number of words and convert them in two parts context and target. target is middle word and context is words surrounding it like 2 words before our target and 2 words after our target.

Architecture

- input layer:
  - we take one hot encoding vectors of context words
- embedding layer (also called projection layer):
  - here we multiply those one-hot encoding vectors with $V \times D$ where V is number of words in vocabulary and D is number of dimensions we want to represent each vector of word
- average laye:
  - here we take average of all these vectors and as output we get only one vector with $1 \times D$ dimesions
- output layer:
  - here we multiply our average vector with $D \times V$ matrix were V is number of words in the vocabulary. where gives us logits
- softmax layer:
  - here we then use softmax function over those logits to get probabilities of each word in vocabulary

so this is the architecture of CBOW model. after getting probability we find loss and use backpropagation to adjust weights

2. Skip-gram Model:
   The Skip-gram model aims to **predict the surrounding context words given a target word.** It's essentially the reverse of CBOW.

   Architecture
   - input layer:
     - here context is middle word and target is words surrounding it.
     - so in input layer we have one-hot encoding vector of that word (context).
   - embedding layer:
     - here we multiply that one-hot encoded vector with embedding matrix of $V \times D$ dimensions.
   - output layer:
     - as we have only one vector we skip the averaging step here
     - we multiply this vector with matrix of $D \times V$ which converts our vector of size $1 \times D$ into size of $V \times 1$ i.e column vector of size V
     - this output vector gives us logits
   - softmax layer:
     - we covert this raw logits into probabilities by applying softmax function on it

- and we want our target vectors to have most probability