PRELAB #7

Several data is collected by companies in the form of text

The subfield of ML that manages text is called **NATURAL LANGUAGE PROCESSING (NLP)**

NATURAL LANGUAGE PROCESSING (NLP): Branch of AI that helps machines process and understand human language in speech & text form.

- In order for machine learning models to process words and blocks of text, the text must be first transformed into numerical features without it losing its underlying meaning.
- Ex of NLP: Automated translation services
  Social networks use text and posts to rank newsfeeds to make them relevant to you.
- Applications of NLP:

Application	Description	Methods
Sentiment Analysis	Is the tone of this text positive, negative, or neutral?	Classification
Topic Modeling	What topic does this text belong to?	Classification; Clustering; Laten Dirichlet Allocation
Translation	What is the best translation of text in language K to text in language J?	Classification; Sequence-2-Sequence Modeling
Text Summarization	Generate a short sequence of text to summarize a longer sequence	Sequence-2-Sequence Modeling
Language Generation	Given a sequence of text, what is the next most likely sequence of new text?	Classification; Generalized Adversarial Networks

Most are solved using a specialized algorithms called **RECURRENT NEURAL NETWORKS (RNN)**

There are other methods which are specialized algorithms called: **SEQUENCE TO SEQUENCE MODELING** **GENERALIZED ADVERSARIAL NETWORKS**

- Many NLP applications are performed with standard ML methods where the specific methodology happens in the preprocessing stage
- When we solve NLP problems with standard ML techniques. The tech specific components are in the data preprocessing.

COMMON PIPELINE:

Tokenize:
Parse documents into sentences, parse sentences into words or phrases, parse words into subwords or characters.

→ Map every word in our training data to a future position

Next step

Pre-Process:
Remove likely unpredictive words and standardize word forms to reduce the vocabulary size.

→ We don't often want to keep the OG text in its entirety or in its OG form

EX) Lots of the "the" and "and" don't add much predictive value ∴ its normal to remove them before doing any modeling.

EX) Some words have the same meaning S.A. learn & learning ∴ we want to convert them to the same term to get better results

Next step

Feature-ize:
Convert remaining tokens to numeric representation.

→ Attach a numerical value to each word to make it a feature

Once we complete this step we end up with a standard data matrix that can be used in classification tasks.

Next step

Model:
Learn relations between input token representations and output labels or relate tokens among themselves.

Different approaches of converting text to numerical features.

- Strategies to map individual word tokens to a number is **VECTORIZATION** & **EMBEDDING**
- We can use NLP techniques such as **TOKENIZATION** & **VECTORIZATION**
- **VECTORIZATION:** simple process of representing the binary presence or frequency of a word in a given example. converts words into individual features with either a binary or frequency based value.

EMBEDDING: Converts words to dense vector representation from a preprocessed from a pre-trained separate process. Each word can be represented by k-dimensional vector where those factors are commonly pre-trained and available as a look up table for you.

- **DATA PREPARATION FOR NLP.**
  - Starts with TOKENIZING TEXT
  - Then running applicable preprocessing techniques on tokenized text.
  - To build tokens:

STEP 1: Scan text in each example, parse out each individual string into a token & create a mapping from token to feature ID.

↳ This can be applied to individual words, characters, combinations, or either.

Doc	Text
1	The cow jumped over the moon.
2	Somewhere over the rainbow.
3	Over the moon.

Build Vocabulary

Token	FID
cow	0
jumped	1
over	2
moon	3
somewhere	4
rainbow	5

We leave out some words because they take up memory without adding predictive value

STEP 2: After building token-to-feature ID dictionary and applying preprocessing techniques, we would build a data matrix S.A. what we have here:

Doc	Text
1	The cow jumped over the moon.
2	Somewhere over the rainbow.
3	Over the moon.

FID	0	1	2	3	4	5
Token	cow	jumped	over	moon	somewhere	rainbow

Doc	X0 (cow)	X1 (jumped)	X2 (over)	X3 (moon)	X4 (somewhere)	X5 (rainbow)
1	1	1	1	1	0	0
2	0	0	1	0	1	1
3	0	0	1	1	0	0

SIMPLIEST APPROACH:
Assign a binary value on whether or not the specific word is in the examples.

This is how we get numeric features.

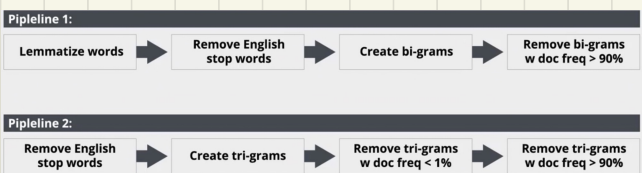
At this stage the text should be suitable for any ML training algorithm

↳ COMMON PREPROCESSING STEPS: These will essentially determine what should be an individual string token & thus should be included in the final feature set.
[All happen at the stage where we're parsing the text & creating the text-to-feature mapping]

1. LEMMATIZATION: We take each word and converting it to some canonical form.
   ↳ Most common used when you've dealing with different verb tenses, noun versions of a verb, possessive and plural forms of a noun.
   ↳ MOTIVATION: to reduce our memory footprint by reducing the feature set size without hurting model quality.
   Ex) am, is, are ⟹ be
   Ex) cats, cats', cats ⟹ cats
   Ex) playing, player, played ⟹ play

2. MAKE N-GRAMS: N-grams are just combinations of individual word tokens as shown in this example.
   ↳ Most common choice of n would be 2 (bi-grams) or 3 (tri-grams)
   ↳ When using n-grams we typically add them to our feature set along with the OG words
   ↳ MOTIVATION: Capture more nuanced meanings and expressions that require multiple words to understand
   Ex) "This product is terrible, definitely not great"
   BI-GRAMS: "this product", "product is", "is terrible", "terrible definitely", "definitely not", "not great"
   TRI-GRAMS: "this product is", "product is terrible", "is terrible definitely", "terrible definitely not", "definitely not great"
   (would map to a new feature)

3. REMOVE STOP WORDS: Token that appears very frequently in different examples of text but also adds very little predictive value.
   ↳ MOTIVATION: We want to remove them to reduce data size and to speed up computation.
   ↳ HOW TO IDENTIFY A STOP-WORD:
   1. Word belongs to a specified language-specific set
   2. Word has document frequency > K or document frequency < J
      (Remove if it appears too frequently / Remove if it appears too infrequently)
   Ex) Conjunctions "and" and "or" or the specific speech "the" and "a".



SIMPLIEST APPROACH:
Try different pipelines and choose the one that yields the best performance.

GENERAL GOAL:
Build a feature set that captures as much predictive value as possible but isn't so large that it becomes computationally infeasible to run or becomes too likely to overfit.

FEATURE ⟹ MODEL STEP

We'll assume that the text has already been appropriately parsed and any preprocessing has been done.

VECTORIZATION: Tells us the numeric value the feature should take on.

↳ 3 COMMON FORMS OF VECTORIZATION:

1) BINARY: Use the presence of the token in a document as the numeric feature value.
2) COUNT: Use the count of the token in the document.
3) TERM FREQUENCY INVERSE DOCUMENT FREQUENCY (TF-IDF): Builds on the count method. Use the term frequency within a doc divided by the document frequency
   - Start by computing the frequency in which the appears across all documents
   - Motivated by heuristics BUT it's very predictive and useful method
   - If the token appears a lot in a given document ⟹ importance to the document goes up
     If the token appears a lot of other documents ⟹ importance to the document goes down
   - EXAMPLE OF TF-IDF in Scikit-learn:

```
X = df['text_feature']
y = df['label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.20, random_state=1234)


from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(
                                   ngram_range = (1,2),
                                   stop_words='english',
                                   min_df = 10
)
tfidf_vectorizer.fit(X_train)

X_train_tfidf = tfidf_vectorizer.transform(X_train)

X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

Tokenization and pre-processing all specified in the Vectorizer

"Fitting" builds the word to feature dictionary (fit the vectorizer)

"Transforming" applies the TF-IDF logic and converts text to numeric data matrix

PROCESS is SIMILAR to what we usually see in model building

SCIKIT-LEARN PIPELINES

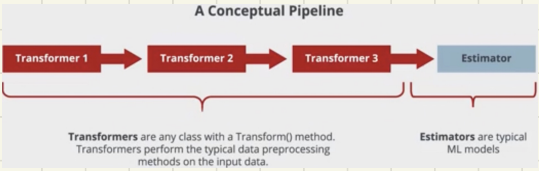
Building a text classifier involves lots of steps. Before any model selection we need to perform a series of preprocessing steps to convert the text into numeric data that's compatible with most ML algorithms.

With TEXT DATA only the sequence of preprocessing to modeling can be AUTOMATED.

↳ AUTOMATION helps us explore a range of design options & choose what's best for the application.

Scikit-learn provides packages that enables us to combine preprocessing & model selection into 1 convenient set of steps:

↳ Package pipeline: a series of transformers followed by an escalator

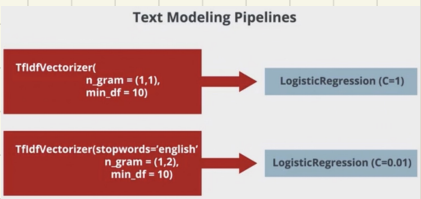


It's a class that transforms data usually from a more raw form to a form more suitable for some modeling application.

ADVANTAGES:

- Pipeline approach is very useful when we want to combine the preprocessing and model selection steps into streamlined and automated process
- Using a pipeline ensures that the preprocessing we do for training data is identical to the preprocessing we do for training data is identical to the preprocessing we do to our test data.

EXAMPLE PIPELINES THAT WOULD BE COMMON IN TEXT BASED MODELING:



→ Where using 1 transformer b/c lot of the text processing technique are performed by the same transformer.

→ Call the TF-IDF vector in scikit-learn, we convert the raw text into a data matrix where the values are determined by the TF-IDF process.

→ Each uses the same type of transformer & estimator but the input parameters are different.

→ While finding the optimal configuration of preprocessing & model hyperparameters can be a time consuming process, the pipeline approach can streamline it.

FIRST IS TO SET UP THE PIPELINE:

```
from sklearn.pipeline import Pipeline

text_pipe = Pipeline(
                [
                    ('vectorizer', TfidfVectorizer(min_df=10)),
                    ('model', LogisticRegression(C=1))
                ]
)

text_pipe.fit(X_train, y_train)

test_predictions = text_pipe.predict_proba(X_test)
```

→ Each individual item in the list is either a TRANSFORMER or an ESTIMATOR

→ Last step will ALWAYS need to be an ESTIMATOR

→ We include the hyperparameters of each step when we build a pipeline

Once set up we treat the pipeline as a STANDARD MODEL OBJECT.

Pipeline is specified as a list of steps. Each list item is a tuple with (description, class).

input is the RAW TEXT DATA → The raw text is converted to #s using TF-IDF vectorizer & then we build a logistic regression on top of that.

We can treat the Pipeline like any other model object.

make predictions against the raw text

With the pipeline approach we can apply multiple types of preprocessing to our data, fit a model, and then predict with new data in just 3 steps.

COMPLETE MODEL BUILDING PROCESS: pipeline approach + model selection

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split

############################## Split data ##########################################

X = df['text_feature']
y = df['label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.20, random_state=1234)

############################## Set up the Pipeline and Parameter Grid ##############
text_pipe = Pipeline(
                [
                    ('vectorizer', TfidfVectorizer()),
                    ('model', LogisticRegression())
                ]
)

param_grid = {'model__C': [0.1, 10],
              'vectorizer__min_df':[10, 100],
             'vectorizer__ngram_range':[(1,1),(1,2)]}

############################## Set up and run the grid search ######################
grid_search = GridSearchCV(text_pipe, param_grid, cv = 5, scoring = 'roc_auc')
grid_search.fit(X_train, y_train)

############################## Get predictions on new data #########################
best_model = grid_search.best_estimator_
predictions = best_model.predict_proba(X_test)
```

3 hyperparameters I we'll be exploring & different configurations.

Initiate CV process using the fit method.

We can then access the best performing model & use that to make predictions or do evaluation.

Q: WHICH OF THE FOLLOWING REPRESENTS A VALID PIPELINE FOR NLP APPLICATIONS?

A. count vectorizer → model training → stop word removal → Model validation

B. stop word removal → model training → count vectorizer → Model validation

C. count vectorizer → stop word removal → model training → Model validation

D. stop word removal → count vectorizer → model training → Model validation ← first vectorize words then remove those with low frequencies

WORD EMBEDDINGS APPROACH

- Converting text into features
- Output of an embedding:

Matrix of 4 words & their associated embeddings

Word	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15	X16	X17	X18	X19
cat	0.94	0.76	-0.66	-0.10	-0.63	0.88	0.56	-0.40	0.21	0.98	-0.05	-0.22	0.12	-0.38	-0.41	0.99	-0.86	0.64	0.81	0.36
jumped	-0.56	0.64	-0.75	-0.31	0.08	-0.59	-0.47	0.32	-0.23	-0.08	0.73	0.53	0.39	0.36	-0.62	0.84	0.10	-0.96	0.91	-0.12
over	0.78	0.22	0.98	-0.46	-0.59	-0.06	0.98	0.56	-0.28	0.31	1.00	-0.84	0.23	-0.74	0.09	0.99	-0.91	0.23	0.47	0.89
moon	0.23	-0.48	0.77	0.02	-0.93	-0.31	-0.21	0.93	-0.70	0.31	1.00	-0.05	-0.29	0.16	0.18	-0.48	-0.55	0.94	-0.60	0.83

- **WORD EMBEDDING:** K-dimensional vector of values that are typically in the range of [-1,1].
  - Choice of K is up to you
  - Typical word vectors range from 50 to 300
  - HIGHER THE K VALUE ⟹ BETTER QUALITY EMBEDDINGS BUT requires greater computational resources
  - LARGER THE VECTOR SIZE ⟹ MORE MODELING FLEXIBILITY BUT to effectively use it you need larger data sets.
  - The VALUES in the EMBEDDING are UNIT LISTS
    The MEANING of the values lies in how they identify similarities between the words

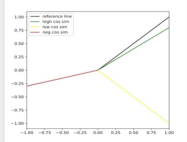
Ex)

→ We can use a SIMPLE FUNCTION to quantify similarity between 2 word vectors

FUNCTION TO COMPUTE SIMILARITY between 2 VECTORS: **COSINE SIMILARITY**

$$\text{Cosine Similarity} = \frac{V_1 * V_2}{|V_1||V_2|}$$

```
def cosine_similarity(v1, v2):
    '''
    Inputs v1 & v2 are Numpy arrays
    '''
    dot_product = sum(v1 * v2)
    norm_v1 = np.sqrt(sum(v1 * v1))
    norm_v2 = np.sqrt(sum(v2 * v2))
    return dot_product / (norm_v1 * norm_v2)
```



- Range of [-1,1]
- Words with similar semantic meanings ⟹ word embedding with HIGH cosine similarity.

Ex)

- **Word2vec:**
  - Popular word embedding methodology

WORD EMBEDDINGS: mappings from words to vectors [Ex) A particular word embedding may map the word "flower" to a vector of £1,2,3,...,4,63]

- **PURPOSE:** To embed semantic meanings of words into vectors of numbers.
  - These vectors are then used as input a ML model
- **WORD EMBEDDING vs. VECTORIZATION:**
  - **SIMILARITY:** both convert text into numerical representations
  - **DIFFERENCE:** word embedding seeks to capture the semantics of words
    Vectors simply convert words into #s based on some predetermined rules

  Ex) The word "doctor" is much more similar to the word "patient" than the word "cat"

- To measure the similarity between 2 words: **COSINE SIMILARITY**

$$\text{Cosine Similarity} = \frac{V_1 * V_2}{|V_1||V_2|}$$

- 2 vectors that are aligned in the SAME direction has a value of 1.
- 2 vectors that are aligned in the OPPOSITE direction has a value of -1.
- 2 vectors that are ORTHOGONAL to each other has a value of 0.

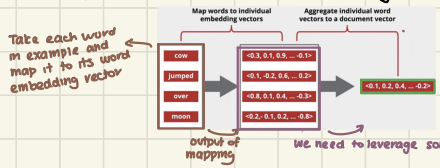
- **MOTIVATIONS:**
  1. Massively reduce feature count ⟹ helps reduce the computational & storage burden with the underlying data.
  2. Massively reduce data sparsity
  3. Pools similar words based on similar semantic meaning

removes common issue in standard text factorization techniques.

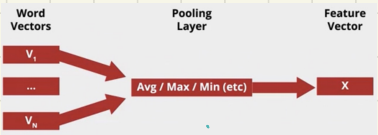
A lot of words are rare which makes learning patterns from them difficult. These words don't show up enough in examples for us to train a low variance signal on them. This is what data sparsity means

- Word embedding helps us pull similar words which helps us reduce DATA SPARSITY while not throwing out any data.

- **USING WORD EMBEDDINGS:**
  - To use word embedding on our model training... Do some preprocessing



  - **MOST COMMON AGGREGATION METHOD:** Take elementwise average of each word vector dimension.

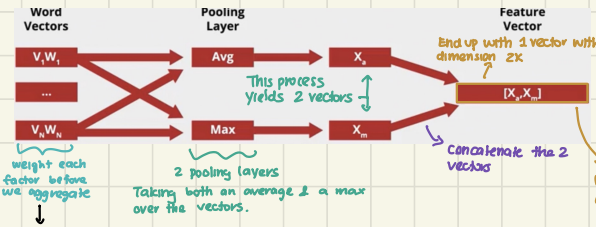


We can generalize this aggregation idea into a **POOLING LAYER**.
↳ Aggregates our multiple vectors into 1.

Other kinds of aggregation techniques:
- MIN AGGREGATION TECHNIQUE
- MAX AGGREGATION TECHNIQUE

} more likely to capture important words that have more extreme values in any one dimension

EXPANDING ON THE IDEA: 2 pooling layers as well as vector weights



It's likely that certain words are more influential in a given example than others.

Using an average pooling layer might reduce the weight of important words in a particular example.

We overcome this limitation by using 2 pieces of functionality added within this diagram. First weight each factor before we aggregate them.

BOTH max & average values capture the average meaning of the words associated with larger word embedding values.

Good weighting technique: apply TF-IDF weights
↳ First compute the TF-IDF weight for each word in each document & multiply the associated weights with the associated word vectors.
This would reduce the impact the vectors associated with words that are more common across documents.

- **CONSIDERING WHICH EMBEDDING TOOLS: DESIGN DECISION**
  1. Should I use embedding or not?
  2. What dimensionality should I use?
     ↳ KEY COMPONENT is to see the size of the embeddings. Common options are 50, 100, 300
  3. Which pooling function(s) should I use?
  4. Should I weigh the individual word vectors?

Ultimately you need to consider a few reasonable options & run empirical tests & validation to choose which is optimal for your problem.

BEST APPROACH: start simple & iterate from there
↳ You can test one embedding method with specified dimensions & compare against non-embedding method.
If you find one option better, you can iterate on that particular option until you reach a point of diminishing returns.

NEURAL NETWORKS: class of supervised learning algorithms that can find complex patterns & relationships in data & can therefore solve very complex problems that other models cannot solve.

↳ Can be used in both CLASSIFICATION & REGRESSION problems.

↳ Use of vectorizers & standard preprocessing techniques covers several classification tasks S.A. SENTIMENT ANALYSIS & TOPIC CLASSIFICATION.
Word embeddings are more of a modern technique in the sense that they're part of the suite methods driven by neural networks but they can still be used to solve classic & simpler NLP tasks.

↳ ADVANCED NLP APPLICATIONS:
- MACHINE TRANSLATION = translates text from one language to another
- TEXT GENERATION/CHATBOTS = generate sensible dialogue
- QUESTION & ANSWERING = generate sensible answers given a question
- TEXT SUMMARIZATION = extract small snippets of text that summarizes a larger body of text

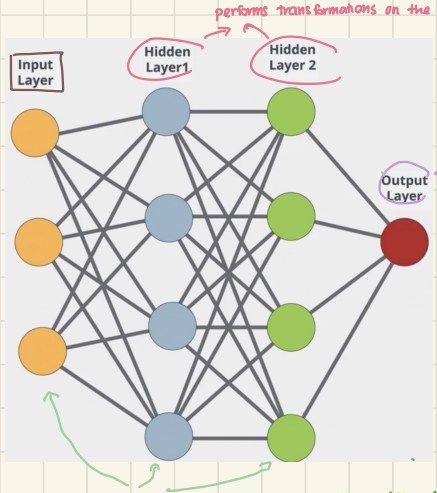
} Goes beyond standard for classification. Instead of generating phrases or sentences that replicates human use of language.

↳ All NLP tasks use ML models known as NEURAL NETWORKS

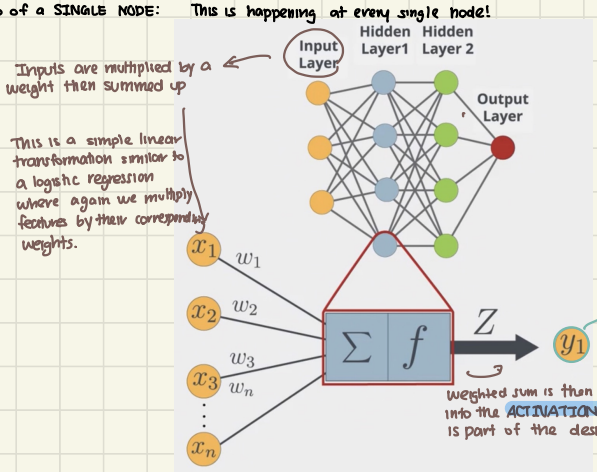
↳ NEURAL NETWORK:
- When relationships between features and a label are LINEAR we use SIMPLE LINEAR MODEL to model those relationships
- When relationships become more complex or NON-LINEAR we have to find another model to make predictions.
  ↳ NEURAL NETWORKS is a popular class of models that can help us identify non-linear relationships
- NEURAL NETWORKS are supervised learning models that can learn very complex patterns & relationships among features & labels. ∴ they can solve very complex patterns, real-world problems.
- NEURAL NETWORKS is a composition of simple linear & non-linear transformations of input data.

- **DIFFERENCES BETWEEN LINEAR MODEL & NEURAL NETWORKS:**
  - linear model like logistic regression learns one weighted function or line that can be used as a model to make the predictions
  - Neural network extends this using many linear functions coupled with linear transformations of those function so that the final combination of them is a non-linear relationship
    - More flexible b/c it allows data to be modeled with more complex weighted functions ∴ giving neural network the ability to solve very complex problems dealing with non-linear relationships between features & labels.

- **STRUCTURE OF NEURAL NETWORKS:**



Close up of a SINGLE NODE: This is happening at every single node!



ACTIVATION FUNCTION: Helps us capture non-linear relationships in our data & is what distinguishes a neural network from something like a logistic regression.

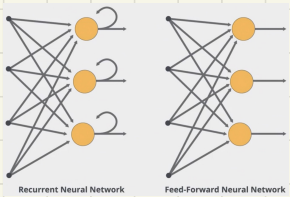
- A neural net with many layers & nodes per layer is performing millions of transformations on the data that enables it to fit any arbitrary curve between features and a label

- The neural network you just saw is called **FEED FOWARD NEURAL NETWORK**
  - NOT ideally suited for SEQUENTIAL DATA (i.e. Text data)
  - FEED FOWARD NEURAL NETWORK would take in usually 1 word or sentence at a time in a text sequence but would lose context of what was previously input.

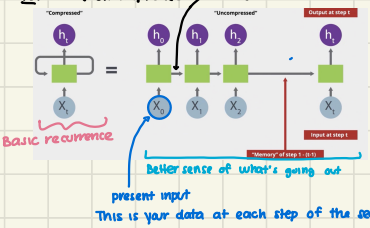
    We would say here that they have NO MEMORY from previous steps of the sequence.
    Since they have no memory they lose important signals from the sequence & are generally BAD at predicting what's coming next.

    To accommodate this issue, special neural network called **RECURRENT NEURAL NETWORKS (RNN)**

- **RECURRENT NEURAL NETWORKS (RNN):**
  - Designed so that the output of a given node flows back into the same node
  - The information cycles through a loop. When it makes its decision, it considers the current input & also what it has learned from the inputs it has previously received.



RNN compressed vs. RNN uncompressed



memory (represented by arrows btw squares) You can think of the memory as an embedding of all the previous information has been passed in the network.

RNN will be processing a sequence where each step of the sequence is indexed by the symbol T

The network here has the ability of to store in its memory representation a lot of the key language context needed to draw specific conclusions about the modeling task.

At each step we have the output represented by the PURPLE NODES

There's several variations of RNN these variations include:

- Choices for the architecture of the cells
  - ENCODER-DECODER NETWORKS: stacks 2 separate RNNs next to each other
    - It encodes the input sequence first into a low dimensional space & then the decoder part decodes the lower dimensional representation into an output
    - Often used in translation:
      encoder creates an embedding representation of a piece of text from 1 language & then decoder converts the encoding to plain text in another language.
- Direction of the flow of information
- Layering of the cells

LINEAR & NON-LINEAR TRANSFORMATIONS

- Neural Networks follows the patterns of linear models in that it computes a weighted sum of inputs and these weights are what's learned by the model.
- In Neural Networks every node in a hidden layer receives input from all NODES/NEURONS in the previous layer.
  - Each NEURON applies the ACTIVATION FUNCTION to the sum of weighted inputs coming from the previous layer's neurons.
  - You can choose an ACTIVATION FUNCTION based on your problem & what you're trying to predict.
  - COMMON ACTIVATION FUNCTION: ReLU (RECTIFIED LINEAR UNIT)
    - Defined as: $f(z) = max(0, z)$
    - Input $< 0 \implies$ output $= 0$
      Input $> 0 \implies$ output $= z$
    - If weighted sum it receives from the linear function is less than 0, it can "turn off" & pass along a 0, otherwise it'll simply pass along the same weighted sum it receives.

- Neural Networks is just a series of linear and non-linear transformations. It involves 2 things: LINEAR FUNCTION + NON-LINEAR ACTIVATION FUNCTION
  - Linear function is applied to the next input layer
  - Corresponds to what you've implemented in other linear model: a matrix vector multiplication that involves multiplying input features by their weights & adding up the results.
  - Output is AMPLIFIED/DAMPED by a NONLINEAR ACTIVATION FUNCTION.
    - This is where firing of a neuron in a neural network comes from.
    - The activation function takes the output of the linear function (the SUMMATION VALUE) and essentially determines how little or how much the linear function should contribute to the next level & ∴ "transforms" the weighted sum to 0.

      The output of the function is the output of the neuron; in the case we described the neuron "turns off" & passes on 0 to the next layer.

This linear + non-linear transformation process continues for every hidden layer.

All neurons in 1 hidden layer output a value. These collective values becomes input to every neuron in the next layer, resulting in more linear transformations which is then followed by another application of non-linear transitions & so on.

The nonlinearity of the transition functions — that allows neural networks to model very complicated relationships.

- Linear models use GRADIENT DESCENT + LOSS FUNCTIONS in their training to determine best parameters
  Neural networks use GRADIENT DESCENT + LOSS FUNCTIONS for optimization during training.
  REGULARIZATION can be applied to prevent overfitting

PYTHON PACKAGES FOR NEURAL NETWORKS

1. Tensor Flow
2. Caffe
3. Py Torch
4. Keras

TIME SERIES FORECASTING: Family of algorithms.

- Recommendation system based on something similar you bought based on your historical data. Trying to predict what your trying to buy next.
- ALGORITHM: LSTM (Long short-term memory)
  GRUs

REVIEW RECURRENT NEURAL NETWORKS

- **FEEDFOWARD NETWORKS** are not ideal for text data because text data are sequential & context dependent.
  ∴ we need more complex neural networks to encode this sequential nature of text.
- Special types of deep learning neural networks have been developed to deal with text data. These models consist of:
  1. **ENCODER:** A neural network that takes in a sequence of words (represented using word embeddings) and outputs a vector or a code that can be viewed as a summary of the input sequence
  2. **DECODER:** A neural network that takes in the output of an encoder & turns it into a scalar or sequence of outputs. (These can be words represented by word embeddings or other things, depending on the application.)
- **RECURRENT NEURAL NETWORKS:**

The goal of a sentiment analysis task is to tell whether the sentiment behind a text is positive or negative. In such cases, the ordering of the input text is important. For instance, the sentence "John enjoys eating the shark" and the sentence "The shark enjoys eating John" are very different (the second one probably has a more negative sentiment than the first). We need more complex neural networks to encode this sequential nature of text.

This is where recurrent neural networks (RNNs) come into play. Concretely, the simplest recurrent neural network consists of two learned parameters, $\mathbf{U}$ and $\mathbf{W}$. Suppose you have a sequence of inputs $s_1, s_2, s_3, \ldots, s_n$ (this can be a sentence where each s_i is the embedding of a word in the sentence) and an initial hidden state $\mathbf{h}_0$.

Computing the Output of a Recurrent Neural Network

A recurrent neural network essentially does the following for each input s_i in the sequence:

1. We process the previous hidden state $\mathbf{h}_{i-1}$. This is done with matrix multiplication to yield $\mathbf{W}\mathbf{h}_{i-1}$. This represents the "history" component of the sentence before the word s_i.
2. Process the current input s_i by using matrix multiplication to yield $\mathbf{U}s_i$. This represents the information from the current input s_i.
3. Add up the the output of the previous two steps to yield $\mathbf{W}\mathbf{h}_{i-1} + \mathbf{U}s_i$.
4. Apply a nonlinear activation function to the output of (3) to yield the hidden state $\mathbf{h}_i = \sigma(\mathbf{W}\mathbf{h}_{i-1} + \mathbf{U}s_i)$.
5. (Optional) Depending on the application, we can further process the hidden state variable o_i.

The final hidden state can be viewed as the vector representation of the whole sequence.

Intuitively, think of the hidden state as a temporary representation of all the previous words and this hidden state will determine how to process the current input. Having this hidden state allows the neural network to "remember" the sequence and encode the sequential information in the sequence.

Training Recurrent Neural Networks

The output of the RNN can be used in a variety of ways, which impacts the way it should be trained.

For instance, in machine translation, we want to have a model that can take in an English sentence and output a German sentence. We can use one RNN as our encoder to encode the English sentence into a vector representation $\mathbf{h}$, then have another RNN as our decoder that decodes $\mathbf{h}$ into a German sentence. The encoder is straightforward: we pass in a sequence of embedded English words and follow the above steps.

However, for the decoder, we pass in a vector (the encoded English sentence) instead of a sequence of words. How can we use an RNN if the input is a vector and not a sequence of word embeddings? The key is to use $\mathbf{h}$ as the initial hidden state and input a starting word for the sentence. This can be as simple as a non-word, or a "token" that indicates the start of a sentence. We then pass the hidden state variable through some function that picks the best German word. For example, given the hidden state $\mathbf{h}$, you predict word i with word vector $\mathbf{w}_i$ with probability $P(i|\mathbf{h}) = \frac{\exp(\mathbf{w}_i^\top \mathbf{h})}{\sum_i \exp(\mathbf{w}_i^\top \mathbf{h})}$. Sometimes people pick the most likely word; sometimes they sample one randomly from this word distribution.

The word vector of the generated German word is the next input passed into the RNN. The whole process can be repeated until some stopping criterion. We can then compute the loss between the ground truth translation and the prediction made by the decoder. With this loss, we can continue the training process to update the parameters of the encoder and decoder.

TERMINOLOGIES

NLP: To standardize & analyze text at scale. This includes

It also includes:

- converting text to numeric representations so we can perform statistical modeling on text more easily.
- computing a degree of association among documents to measure their similarity or dissimilarity.
- drawing structural & semantic meaning from text and organizing documents to search them more efficiently
- cleaning, preprocessing text to reduce typos, removing unimportant words & phrases, parsing text into meaningful words and sentences.

CORPUS: large set of text documents

DOCUMENT: long sequence of characters

WHITESPACE: {\t, \r, \n, \v, \f, " "}

CHARACTER TOKEN: |N|L|P| |i|s| |f|u|n|.| |I| |l|i|k|e| |i|t| |a| |t|o|n|.|

WORD TOKEN: |NLP| |is| |fun|.| |I| |like| |it| |a| |ton|.|

SENTENCE TOKEN: |NLP is fun.| |I like it a ton.|

CHEAT SHEET

Neural Network Cheat Sheet

Algorithm Name	Neural networks
Description	Neural networks are versatile models. They use piecewise linear functions to approximate decision boundaries (classification) or the label function (regression).
Applicability	Any supervised learning problem (classification or regression)
Assumptions	Input features are homogeneous. Label function is smooth.
Underlying Mathematical Principles	Activation function Loss function Forward propagation Backward propagation Stochastic gradient descent
Activation Functions	ReLU = $\sigma(z) = \max(z, 0)$ Sigmoid = $\sigma(z) = \frac{1}{1+e^{-z}}$ tanh = $\sigma(z) = \tanh(z)$
Loss Functions	**Regression:** Squared loss **Classification:** Cross entropy loss
Additional Details	• Training is usually done using SGD — a variant of GD that uses gradient computed from a small batch of training examples (sampled in each iteration). • Gradient is computed efficiently using backwards propagation. • Weight decay and dropout are used to regularize the model.

CHEAT SHEET

Recurrent Neural Networks

Algorithm Name	Recurrent neural networks
Description	A special type of neural network that is well suited for sequence data
Applicability	Any supervised learning problem (classification or regression) when the input data are sequences
Assumptions	The data is a discrete sequence
Underlying Mathematical Principles	Hidden states
Additional Details	A recurrent neural network keeps track of a hidden state that can be viewed as a summary of the input sequence. In essence, an RNN processes the input sequence in a stage-wise manner, i.e., it takes in a random starting hidden state and combines it with the first item in the sequence to form a new hidden state, and this new hidden state is combined with the next item in the sequence to produce another new hidden state. Sometimes the RNN also produces an output at each state, which is typically a linear transformation of the hidden state at this position. This process is repeated until every item in the sequence is processed. Depending on the task, the final hidden state will be used differently. If we want to classify a sequence (sentiment classification), then the final hidden state will be passed to a classifier. If we want to produce another sequence (machine translation), then the hidden state can be used as the initial hidden state of another RNN.