Introduction

Natural Language is complex: Word level ("bring me that file"), syntactic ("I saw the boy with a telescope"), semantic ("I haven't slept for 10 days"), referential ("We gave the monkeys bananas because they were more than ready to eat."), idioms; all forms of ambiguity.

Linguistic Levels of Language Understanding:

Used to have separate models to learn lexicon, syntax, semantics, discourse (relations b/w sentences) and pragmatics (intent). Now we mainly learn it all with a single end-to-end model.

- *Lexicon* - morphological analysis:
  - Lemmatisation (reduce words to canonical valid word): ate, eating -> eat
  - Stemming (reduce to root word; not always valid word): connected, connecting, connection -> connect; worrying, worried -> worr
  - Byte-pair encoding (BPE); data driven & used by neural models
  - Part-of-speech tagging (recognising word category, e.g: verb/noun)
  - Morphological analysis (handle word variants; plural, tense, affixes)
- *Structure, Syntax* - Parsing:
  - *Grammar*: How are words put together?: e.g: S -> NounPhrase VerbPhr
    - Can build grammar trees that go all the way down representing sentences in terms of our own BNF
- The main historical motivation of NLP was machine translation.

Neural Nets: input, hidden layers, output. Feed-Forward: fully conn. Weight matrices to assign prev layer neuron contributions. *affine* layer = fc layer.

Activations: 1) Sigmoid: σ(x) = 1 / (1+e^(-x)), 2) ReLU(x) = if x <= 0: 0 else: x 3) tanh(x) = (e^x - e^(-x)) / (e^x + e^(-x)) 4) softmax(zᵢ) = e^zᵢ / Σₖ e^(zₖ)

Loss: 1) MSE: L=1/N Σᵢ₌₁ to ₙ (yᵢ-yᵢ^)^2 2) BCE: $L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^{(i)} log(\hat{y}_c^{(i)})$ 3) CategCE: $L = -\frac{1}{N}\sum_{i=1}^{N} y^{(i)} log(\hat{y}^{(i)}) + (1-y^{(i)}) log(1-\hat{y}^{(i)})$

backprop: $W = W - \alpha \frac{\partial L}{\partial W}$

Common Network Setups: 1) *Regression* (predicting score): output layer of size 1, linear activ and MSE loss 2) *Binary Classification*: output layer of size 1, sigmoid, BCE loss 3) Multi-Class Classification: output layer of size k=classes, softmax, CategCE 4) Multi-Label Classif: out: k=classes, sigmoid, BCE

Word Representation

Corpus: a collection of text data; the language dataset we are working with.
Document: One item of our corpus; documents will typically be a *sequence*.
Sequence: a string of words (e.g. a phrase, sentence, paragraph etc)
Token: A token is the atomic unit of a sequence. Nowadays subword.
Vocabulary: the set of unique tokens across our entire corpus.
How can we represent our documents to the model?

1. *One Hot Encoding*:
   - We have a dictionary that assigns id indexes to our entire vocabulary
   - To one-hot encode, for each vector i ∈ 0..N; N is seq_len, set the one index in the vector to 1 corresponding to the dictionary id index: (e.g: if "this" is 7431 in the dict, and we have 100k entries, then the one-hot encoding of "this" is the 100k sized vector with index 7431 = 1.)
   - Yields a one-not sentence representation [seq_len, vocab_size]
   - Problems: orthogonal definitions, high dimensions, fixed vocab
2. *WordNet*:
   - We map words to broader concepts using this NN. "cat", "kitten" → "feline mammal"
     - Maps some similar words together & reduces vector size
     - But relies on the completeness of a manually curated database
     - Misses out on rare and new meanings
     - Vectors are still orthogonal and equally distant from each other
     - Disambiguating word meanings is difficult
3. *Word Embeddings*: Fixed dimensional (~300-1K) dense representations learned by our neural network algorithm.
   - *Distributional Hypothesis*: "Words which are similar in meaning occur in similar contexts" - i.e: define words in terms of words they're nearby.
   - Allows us to do comparisons in nD space (euclidean dist b/w words)
   - We can rank words by categories and plot n categories vs each other
   - *Euclid Dist*: $\sqrt{\sum_{i=1}^{n}(q_i - d_i)^2}$ *Cos Dist*: $\frac{p_1 \cdot p_2}{||p_1|| \times ||p_2||}$ if the vectors are normalized its just dotprod
   - In practice word embeddings have many categories and it's hard to interpret them. Though words like "Man" and "Woman" have a cycle of dims they have similar values in - as their meanings partially overlap...
     - *Similar words have similar vectors!*
   - Word Embeddings allow for transfer learning. Find them once, and then use them in all sorts of downstream tasks.
4. *Word2Vec*: *Allows us to obtain Word Embeddings in practice:*

Target Word: The word we want to represent
Context Word: the words within a "window_size" adjacent to the target word

Word2Vec is a "family" of NN based algorithms to obtain embeddings:

- Architectures: 1) Continuous Bag of Words: simple - Predict the target word wt based on the surrounding context words; we just feed in all the surrounding words as input to our NN Model. Output: vector rep
  2) Skip-gram: We feed in our target word as input, and the model aims to predict the context words.
  - *Input*: One-hot encoding of the target word - v dims (vocab size)
  - Then, using a weight matrix W∈R^VN we obtain a hidden emb h=xW
  - Next we perform our prediction:
    - First, map the emb of the target word to the possible context words, using context weight matrix W'. Done by y=hW'.
      - Here, the N-dimensional embedding gets transformed into logits by being matrix multiplied with the context matrix.
      - Note y has length of the entire vocab
    - Next, apply softmax over our y, transforming the logits into a prob dist, allowing us to use this prediction in our loss function.
  - *Optimization*:
    - We have multiple context words but softmax only really lets us optimize against 1 word. Hence we build multiple training samples for each target word - we fix a target word t and then for each context cs word generate all training pairs (t, cs)
  - *Loss Function*: $\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j<c, j\neq 0} \log p(w_{t+j}|w_t;\theta)$

$p(w_{t+j}|w_t)$ This is just cross entropy loss for each target-context pair derivation:

$= \frac{\exp(h_{w_t} \cdot h_{w_t})}{\sum_{w'} \exp(h_{w'} \cdot h_{w_t})}$

- i.e: p(wt+|wt) is just probability of target given context
- We take the product over all those js (so probability of a word given the ENTIRE context window)
- We then do that over the entire sentence; product over ts
- Then just take negative log likelihood.
- Problem: a fwd pass requires us to sum across the entire corpus in the softmax. This is expensive...

- *Negative Sampling*:
  - To avoid the softmax cost, we can instead approximate it.
  - For each vocabulary item, we have a binary classifier: which attempts to predict whether a target word appears in the context of a context word. Loss = $\log p(D=1|w_t, w_{t+1}) + k\,\mathbb{E}_{\tilde{c}\sim P_{noise}}[\log p(D=0|w_t, \tilde{c})]$
  - For LHS we're maxing the probability of correctly predicting that a target word and a TRUE context word appear in context of each other (i.e. predicting a 1 with our binary classifier). RHS we're maxing the prob of correctly predicting that a target word and a noise word do NOT appear in context of each other (i.e. predicting a 0).
    - We have k contrastive context words from a noise distribution c~.

For example we have "The quick brown fox jumped over the lazy dog."

- To predict "quick" from "the", select noisy (contrastive) words which are NOT in the context window of "the"
  - For example, k = 1, and randomly selected noisy word = sheep
  - Compute loss function for observed and contrastive pairs.
    - The objective, at this time step becomes log p(D=1| the, quick) + log(1 - p(D=1|the, sheep))
    - Our old softmax prob now sigmoid: $p(D=1|w_t, w_{t+1}) = \frac{1}{1+\exp(-w_{t+1}^{\top} h_{w_t})}$
    - k: 5-20 works well for small datasets, 2-5 otherwise.
      - For a model with k=5, W matrix: 300x10000, we need to update weights for 1 positive word, and 5negative samples = 6 output neurons. So 6 * 300 = 1800 weight parameters need to be updated. That is only 0.06% of the 3M weights in the output layer!
- Frequency based sampling outperforms our random sampling here:
  - Instead of p(wᵢ) = 1 / |V| $p(w_i) = \frac{f(w_i)^{3/4}}{\sum_{w'} f(w')^{3/4}}$

Using our Vector Representations:

- *Word Similarity*
- Analogy Recovery: "apple' is to 'apples' as 'car' is to 'cars'": a - b = c - d => d = c - a + b; i.e: queen = king - man + woman
- Note we have very good pretrained publicly available vecs (trained on 100B words). They're easy to use and only need plaintext, and start the model off with a good prior
  - However: Require large amounts of data to train, bad quality for rare words, no coverage for unseen words, Antonyms tend to have similar distributions: e.g. "good" and "bad", does not consider morphological similarity: "car" and "cars" are independent, does not differentiate between different meanings of a word
  - ELMo looks at the entire sentence before assigning / updating embs.

Byte Pair Encoding: Solves dealing w/ OOV tokens, and large vocabs.

- Originally used for compression. TO TRAIN we do the following:
  - Start with a vocab of all individual characters (a, b, ...)
  - Split words in training corpus also into individual chars "table" → "t", "a", "b", "l", "e", "_" ("_" indicates end of word)
  - Find which 2 vocab items occur together most in training corpus e.g "ab"
  - Add that combo "ab" as a new vocabulary item (a, b, ..., ab)
  - Merge all occurrences of that combination in your corpus, "t", "a", "b", "l", "e", "_" → "t", "ab", "l", "e", "_". We repeat until the desired number of merges has been performed.
  - Training Example:
    - i. ("low": 5), ("lower": 2), ("newest": 6), ("widest": 3)
    - ii. w, "_": 5), ("l" "o" "w" "e" "r" "_": 2), ("n" "e" "w" "e" "s" "t" "_": 6), ("w" "i" "d" "e" "s" "t" "_": 3)
    - iii. "t _" occurred 6 + 3 = 9 times ("e s" or "s t" are also equal candidates)
    - iv. ("l" "o" "w" "_": 5), ("l" "o" "w" "e" "r" "_": 2), ("n" "e" "w" "e" "s" "t_": 6), ("w" "i" "d" "e" "s" "t_": 3)
    - v. Repeat from iii until we reach hyperparam number of times merged. We keep track of the merge operations in order
- "_" helps distinguish suffixes and where to insert spaces during recon
- *Inference*: we'll encounter new words we didn't see during training.
  - To handle this:
    - i. Split word into characters
    - ii. Get all symbol bigrams of the word (e.g: "tab" → "ta", "ab")
    - iii. Find a symbol pair that appeared first among the symbol merges dict (remember learned our ordered merges during training)
    - iv. Apply the merge on the word
    - v. Repeat from step 3 until no more merge operations apply
  - Might encounter Unicode we haven't seen during training. Solution: operate on byte seqs > chars. Base vocab size = 256.

TF-IDF: Common words ("a", "the", "it", "this", ...) dominate the counts but provide almost no additional information. TF-IDF solves this:

- We can weight the vectors using TF-IDF$_{w,d,D}$ = TF$_{w,d}$IDF$_{w,D}$.
  - d is the item we're creating a vec for e.g: "magazine" or "newspaper", or a full document (the subset of data we care about)
  - w - the context word that we want to weight. E.g., "reading" or "a"
  - D - A collection of documents (or sentences) that we are using to create these vectors
  - Then:
    - Term Freq (TF): Upweights words w that're more important to d
      - TF$_{w,d}$ = count(w,d) / Σw' count(w', d)
    - Inverse Document Frequency (IDF): Downweights words that appear everywhere
      - IDF$_{w,D}$ = log |D| / (|d ∈ D : w ∈ d|)

Classification

Classification: Predicting what class an observa. belongs to y^ = argmax$_y$ P(y|x)
Example Tasks: Binary Classification (hateful/not), Multi-class (impicit or explit hate/not), sentiment analysis, NL inference (logic argument solving)

Naive Bayes Classifiers: Key: posterior likelihood prior

- *Bayes Rule*: P(y|x) = (P(x|y) * P(y)) / P(x)
- Hence: y^ = argmax$_y$ P(y|x) = argmax$_y$ P(x|y)P(y)
- We make the iid assumption: $\hat{y} = argmax_y P(y) \prod_{i=1}^{I} P(x_i|y)$
- *Example: Using Naive Bayes for Sentiment Analysis Movie Reviews:*
  - Raw input is transformed into numeric representation where each input x has feature vec: [x₁, x₂, ... xᵢ]
  - After any pre-processing (space out punctuation, all lowercase, etc) we can just use bag-of-words (dict of words: counts, stores the entire vocab so most counts would thus be 0). But can condense to just a few meaningful words ("feature extraction") - e.g: good, movie, bad.
  - Then for each piece of training data:
    - Per review, store the bag of words dictionary, alongside the class label
    - Training the model is really just statistic collection. We have 5 reviews, 3 good, 2 bad. P(y) => P(+) = 3/5, P(-) = 2/5.
      - P("good"|+) = 2/4 - i.e: of the 4 words we store features for in the positive class, 2 of them were the word "good"
    - If a value is zero we run into issues, solution: "add-one smoothing": $P(x_i|y) = \frac{count(x_i,y)+1}{\sum_{x_i\in V}(count(x,y)+1)} = \frac{count(x_i,y)+1}{(\sum_{x_i\in V} count(x,y)) + |V|}$
      - Using this, P("good"|+) = (2+1)/(4+3); as |V| = 3
  - Testing: test on sample ("Not as *good* as the old *movie*, rather *bad*", -)
    - p(bad|+) & p(movie|+) = (1+1)/(4+3) as they both occur once throughout all the positive samples as a feature. (found in training)
    - p(good|-), p(movie|-), p(bad|-) = 2/6, as they all occur once out of 2 -.
    - P(+) = 3/5; P(+)P(x|+) = 3/5 * (3*2*2)/(7^3) = 0.021; P(-)P(x|-) = 0.014
      - Remember P(x|+) i.e: P("Not as... *bad*"|+) was computed via iid formula. Here we've favoured the wrong prediction
- *Improvements to Naive Bayes Classifiers*:
  - Remove all duplicates within each instance. So the extra occurrence of the word 'movie' doesn't change anything. Must re-calculate all of the conditional probabilities from the training data.
  - *Controlling for Negation:* append 'NOT_' after any logical negation until the next punctuation mark ("I didn't like the movie, ..." → I didn't NOT_like NOT_the NOT_movie, ...")
- Our problems really stem from: the conditional independence assumption, the fact that the Context of words is taken into account and that new words (not seen at training) cannot be used.

Logistic Regression:

- *Discriminative Algorithms* directly learn P(Y|X) (learn decision boundary)
- *Generative Algorithms* instead learn P(X, Y) (learn prob dists of data)
- Naive Bayes is generative, logistic regression is discriminative
- *Logistic Function*: y(x) = g(z) = 1/(1+e^(-z)) where z = w.x + b
  - P(y = 1) = 1 / (1 + e^-(w.x+b)). y^ = 1 if P(y=1|x) > 0.5, y^ = 0 otherwise.
- For Logistic Regression over classes, if we have input features x1, x2, x3 (count of *good*, count of *bad*, count of *and*), then weights are learnt per class (obviously - w1, w2, w3 in weight matrix correspond to the inputs).
  - For multiclass we use softmax over sigmoid.
- Overall, LR considers feature importance; better at dealing w/ correlated features and is better w/ large data. However, Naive Bayes quick to train.

Cross Entropy Loss: H(P,Q) = -Σᵢ P(yᵢ) log Q (yᵢ)

Neural Networks: 2 components: *linear layers*: z = w.x +b and *non linear activations*: y = g(z). FC layers have all node weights feed into the next layer.

Learnt Feature Representations: 1) Basic Model: one-hot word representatio-n, 2) Improved Model: Automatically learnt dense feature representations, or pretrained dense representations.

Document Representation: We can represent documents by fully encoding each word in our neural networks as weights. But this is bad: Model arch fixed to sentence len size, Model weights learnt for specific word positions

RNNs: Natural language data is sequenced. Hence RNNs are suited for them.

- Value of a unit depends on own previous outputs
- The last hidden state is the input to the output layer
- RNN (f) computes its next state ht+1 based on: Hidden state vector and input vector at time t: hₜ₊₁ = f(hₜ, xₜ) = tanh(Whₜ + Uxₜ)
  - The hidden state is carried along
  - The main params are matrices W ∈ R^HxH and U ∈ R^HxE. Code:

```
size_H = 100, size_E = 20; E = torch.nn.Embedding(vocab_size, size_E) U
= torch.rand(size_H, size_E, requires_grad=True)
W = torch.rand(size_H, size_H, requires_grad=True)
sent = ["this", "is", "a", "really", "good", "movie" ]
h_t = torch.zeros(size_H, 1); loss = 0 # Start as zero
for i in range(len(sent) - 1):
  x_t = E[sent[i]]; h_t = torch.tanh(W.matmul(h_t) + U.matmul(x_t))
```

- *Vanishing Gradient Problem*: Tanh derivatives are b/w 0 and 1, sigmoid b/w 0 and 0.25. Repeated backprop caused update magnitude to fall
- Gradient for earlier layers involves repeated mult of the same matrix W
  - Dominant |eigenvalue| < 1: 'vanish' else 'explode'.

CNNs: composed of a series of conv layers, pooling layers and FC layers:

- Convolution layers: Detect important patterns in the inputs
  - Filter width = embedding dimension
  - Filter height = normally 2 to 5 (bigrams to 5-grams). Stride...
  - If we have d diff parallel filters, then we have a d-dim representation
- Pooling layers: Reduce dim of features, transform them into a fixed-size
- Fully connected layers: Train weights of learned repres. for a specific task

RNNs vs CNNs:

- CNNs can perform well if the task involves key phrase recognition
- Whereas RNNs perform better when you need to understand longer range dependencies

Evaluation Metrics:

1. Accuracy = TP+TN / (TP+FP+TN+FN)
2. Precision = TP/(TP+FP)
3. Recall = TP/(TP+FN)
4. F1 = (2 x precision x recall)/(precision+recall)
   = TP / (TP + 1/2 x (FP + FN))

Micro vs Macro F1:

- Macro averaging - average of each class F1 scores; increases the emphasis on less frequent classes → better for imbalanced dataset
- Micro averaging - TPs, TNs, FNs, FPs are summed across each class. Equal to accuracy as the sum of FPs = sum of FNs = sum of off-diagonals in confusion matrix:

Micro $F_1 = \frac{\sum_i^C TP_i}{\sum_i^C TP_i + \frac{1}{2}\left(\sum_i^C FP_i + \sum_i^C FN_i\right)} = \frac{\sum_i^C TP_i}{|\text{Dataset}|} = \text{Accuracy}$

Language Modelling 1

Language modelling (LM): Assigning probabilities to sequences of words

- *Causal LM*: Predicting the next word in a sequence
  - The next word in the sentence ..
- *Masked LM*: Predicting a masked word in a sequence
  - The next .. word in the sentence is

N-gram LM Intro: Technique for (causal) LM

- We want to compute P(w|h) where w is the next word and h is the history.
- Given a corpus we can estimate P(w_n|h) as
  - P(w_n|h) = P(w_n | w_{1}, ..., (n-1)) = count(h . w)/ count (h).
  - Where h.w is the w concatenated with h and count(x) is how many times the string x occurs in the corpus.
  - Eg: P("sat" | "the cat") = count("the cat sat") / count("the cat")

N-gram LM: We approximate the full history with just the last n words.

- Full context: P(w_n | w_1, ..., w_{n-1}) ; eg: P("toilet" | "I like skibidi")
- Bigram: P(w_n |w_{n-1}) ; eg: P("toilet" | "skibidi")
- Trigram: P(w_n | w_{n-2}, w_{n-1}); eg: P("toilet" | "like skibidi")
- N-gram: P(w_n | w_{n - N + 1}, ..., w_{n-1})

We can compute the estimate probability using N-Grams as:

$$P(w_n \mid w_{n-N+1}^{n-1}) = C(w_{n-N+1}^{n-1} w_n) / C(w_{n-N+1}^{n-1})$$

Evaluating LMs: We can evaluate a sequence using the product of conditional probabilities given by a LM.

$$P(w_1, \ldots, w_n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1, w_2)\ldots P(w_n \mid w_1^{n-1}) = \prod_{k=1}^{n} P(w_k \mid w_1^{k-1})$$

Evaluating N-Grams: For an N-Gram model our probabilities are given using only the last N words. So we have for example:

- Bigram: P(w1, ... wn) = ∏_{k=1} ^{n} P(w_k | w_{k - 1})

We can insert <s> and </s> markers to the start and end of a sentence.

Perplexity: The issue with evaluating based on probability as above is that longer sequences have lower likelihoods. Rather use perplexity which is the inverse probability of the text normalized by the number of words predicted.

$$PPL(W) = P(w_1, w_2, \ldots, w_n)^{-\frac{1}{n}} = \sqrt[n]{\frac{1}{P(w_1, w_2, \ldots, w_n)}} = \sqrt[n]{\frac{1}{\prod_{k=1}^{n} P(w_k|w_1^{k-1})}}$$

- Note: We do not include <s> in our count for n but we do include </s>. This is because our model doesn't predict <s> but should predict </s> to indicate the end of a sequence.

$$= \sqrt[n]{\frac{1}{\prod_{k=1}^{n} P(w_k|w_{k-1})}}$$

Bigram LM perplexity

Using perplexity: Given a test corpus we predict the likelihood that our model would make the same prediction then compute the perplexity.

- Lower perplexity = higher likelihood = better model.

Change of base rule: log_b a = log_x a / log_x b

Cross Entropy to Perplexity: $H(T, q) = -(1/N)\sum_{i=1}^{n} \ln q(x_i)$

- Cross entropy: q(x_i) = p(x_i| x_1,...,x_i-1);
- Perplexity = e^(cross entropy) (base 2 if using log 2 in cross entropy).

Extrinsic vs Intrinsic Evaluation

- *Intrinsic Evaluation*: Eg. Perplexity.
- *Extrinsic Evalutation:* Eg. Eval on downstream tasks (maths, coding, etc...)

Language Modelling 2

Sparsity: Some tokens are very uncommon meaning that their probabilities may be 0 or very small. Testing perplexity on out of distribution text can lead to very high perplexity/ low likelihood. Following are some solutions to this

- *Add-1 Smoothing*: Steam probability mass from more frequent words by adding 1 to the count of all words.
  - Bigram: P_add-1(w_n | w_{n-1}) = (C(w_{n-1}, w_n) + 1) / (C(w_{n-1}) + V)
  - Easy to implement; But takes too much probability mass from more likely occurences, assigning too much probability to unseen events
  - Could try +k smoothing with a smaller value of k (k <1).
- *Back-off smoothing (stupid back-off)*: If we don't have trigram then use bigram then unigram (weighting as we go)

$$S(w_i|w_{i-2}\,w_{i-1}) = \begin{cases} C(w_{i-2}\,w_{i-1}\,w_i)/C(w_{i-2}\,w_{i-1}) & \text{if } C(w_{i-2}\,w_{i-1}\,w_i) > 0 \\ 0.4 \cdot S(w_i|w_{i-1}) & \text{otherwise} \end{cases}$$

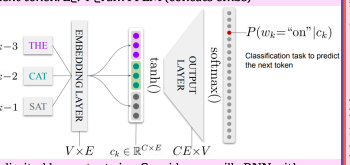
$$S(w_i|w_{i-1}) = \begin{cases} C(w_{i-1}\,w_i)/C(w_{i-1}) & \text{if } C(w_{i-1}\,w_i) > 0 \\ 0.4 \cdot S(w_i) & \text{otherwise} \end{cases} \qquad S(w_i) = C(w_i)/N$$

- *Interpolation*: Combine evidence from multiple n-grams: λ1 + λ2 + λ3 = 1

$$P_{\text{interp}}(w_i \mid w_{i-2}, w_{i-1}) = \lambda_1 P(w_i \mid w_{i-2}, w_{i-1}) + \lambda_2 P(w_i \mid w_{i-1}) + \lambda_3 P(w_i)$$

Neural LMs: Using neural networks for LM, avoids n-gram sparsity issue and lets us extract contextual word embeddings.

Feed Forward Language Model (FFLM): Pass N-1 past tokens to neural network predict next token. Eg. 4-gram FFLM (concats embs)

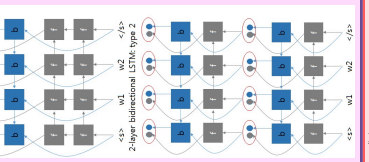


RNNs for LM: Not limited by context size. Consider a vanilla RNN with:

- Hidden-to-Hidden: W ∈ R^{HxH}
- Input-to-Hidden: U ∈ R^{HxE}
- Activation function: σ
- Hidden-to-Output: W_hy ∈ R^{|V|xH}
- Hidden-to-Output Bias: B_y ∈ R^{|V|}
- h_t+1 = σ(W h_t + U x_t)
- y_t = softmax(W_hy h_t + B_y)
- y_t gives us the probability of each token in the vocabulary being the next word. Taking the log, averaging and negating = cross entropy loss.
- At training we use Teacher Forcing (don't propogate predictions, instead GT), at inference time we don't do that.
- Many to many → output for each input word eg: multi-class label, MT, NER. Many to 1 → eg: sentiment analysis, text classification, lang identifc.

Bidirectional RNNs for classification:

- Dual RNNs: Have two RNNs one for each directions. Concatenate representations on either side for embedding of whole text.
- Bidirectional RNN: Combine representations of each token from forward and backwards RNNs.
- Multilayer RNN: Feed hidden state from earlier layer into next layer
- Bidirectional Multilayer RNN: Couple different options



Long Short-Term Memory LSTM: Advanced type of RNN which aims to mitigate vanishing gradient problem. It components are:

- Input Gate: $i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$
- Forget Gate: $f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$
- Output Gate: $o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$
- Cell input activation: $g_t = \tanh(W_{gx}x_t + W_{hg}h_{t-1} + b_g)$
- Cell State: $c_t = f_t \odot c_{t-1} + i_t \odot g_t$
- Hidden State: $h_t = o_t \odot \tanh(c_t)$

Where σ is the sigmoid function, and ⊙ is component wise multiplication.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

LSTM Benefits: LSTMs help mitigate vanishing gradient problem as gradients through cell states are hard to vanish as:

1. Additive formula means we don't have repeated multiplication of the same matrix leading to a more "well-behaved" derivative
2. The forget gate allows the model to control gradient vanishing and preservation, allowing for different values of the gate over time.

Gated Recurrent Unit: Simplified gated RNN

- Update Gate: $r_t = \sigma(W_{rx}x_t + W_{hr}h_{t-1} + b_r)$
- Reset Gate: $z_t = \sigma(W_{zx}x_t + W_{hz}h_{t-1} + b_z)$
- Candidate Activation: $g_t = \tanh(W_{gx}x_t + r_t \odot (W_{hg}h_{t-1} + b_g))$
- Hidden State: $h_t = (1 - z_t) \odot g_t + z_t \odot h_{t-1}$
- No input/output gate or cell state necessary, but has the ability to forget.
- zt acts as the update gate, determining the incorporation of the current info to the hidden states; rt is the reset gate which also impacts the maintenance of the historical information.

Machine Translation

Machine Translation: Given passage in source language, produce a translation in target language. Example of sequence to sequence problem.

Statistical Machine Translation: We want p(t|s), ie. given source sentence s we want the probability of translated sentence t.

- p(t|s) = p(s|t)p(t)/p(s) by Bayes rule
- argmax_t p(t|s) = argmax_t p(s|t)p(t): ignore p(s) as it doesn't depend on t.
- *Language Model p(t):* Can be trained on monolingual data; tells us the probability of a phrase in the target language.
- *Translation Model p(s | t):* Essentially computes how likely is the source phrase given a target translation; built on statistics of our parallel corpus using an alignment model which does phrase pair extraction..

Downsides:

- *Sentence Alignment*: Long sentences in source can be broken up into multiple sentences in target and vice versa.
- *Word Alignment*: Which words do we align in a source-target pair? Some words do not have an equivalent in a target language.
- Statistical Anomalies, idioms, out of vocabulary words.

Neural Machine Translation: Using neural networks for MT.

Encoder-Decoder: Network architecture used to solve sequence to sequence (Seq2Seq) tasks. Encoder represents source as latent encoding, decoder generates a target from this embedding.

Simple RNN: Use the last hidden state of a simple RNN as the decoder input.

Bidirectional RNN: Have two RNNs, one running forward and the other backwards on the source sentence. We can then get the latent embedding in different ways:

- First and Last: Concatenate last hidden states each RNN (ie. hidden state of last token in forward RNN and hidden state of first token in backwards RNN) - [h_i ; h'_0]
- Average: Take the mean of all hidden states in both RNN and concatenate. [mean(h_0, ..., h_i) ; mean(h'_i, ..., h'_0)]

Teacher Forcing: At inference we sample autoregressively. During training we use teacher forcing which is where we feed the ground truth token to the next decoder cell. At step t we make a prediction ŷ_t, calculate the loss against the ground truth y_t: L(ŷ_t, y_t) and then pass y_t (rather than ŷ_t) to the next decoder cell at t + 1.

- We can choose to do this at some ratio r.
- Without teacher forcing we can have accumulating errors
- With r = 100% we can have Exposure bias where our model is never conditioned on its own predictions. Unclear how much of problem this is.

Formalized Naive Encoder Decoder: With a teacher forcing ratio of 100%

- Cross Entropy Loss: $-\sum_{t=1}^{T} \log p(\hat{y}_t | (y_{c=1}, c)$
- c is the context vector given by the encoder.
- ŷ are the decoder outputs: $\hat{y}_t = \sigma(W_o D(y_{t-1}; c))$

Downside of naive Enc-Dec: Performance degrades for longer sequences.

- All sequences are represented by a d-dimensional vector, for long sequences its harder to fit all the information into this vector.
- In decoder information from context can start to fade as information from context vector may start to diminish (vanishing gradient, difficulty remembering past of sequence).

Attention: We want to be able to utilise more information from the encoder in the decoding process. For each decoding step we dynamically choose which of the encoder states are important. We consider MLP attention as a mechanism for this.

- Context Vector for t'th decoder time step is a weighted average of the hidden states of the encoder: $c_t = \sum_{i=1}^{I} \alpha_i h_i$
- Energy score come from the alignment function which is a MLP which combines information from previous decoder state s_{t-1} and an encoder hidden state:

$$e_i = a(\underbrace{s_{t-1}}_{\mathbb{R}^{D\times 1}}, \underbrace{h_i}_{\mathbb{R}^{2D\times 1}}) = \underbrace{v^T}_{\mathbb{R}^{1\times D}} \tanh(\underbrace{W}_{\mathbb{R}^{D\times 1}} \underbrace{s_{t-1}}_{\mathbb{R}^{D\times 1}} + \underbrace{U}_{\mathbb{R}^{D\times 2D}} \underbrace{h_i}_{\mathbb{R}^{2D\times 1}}) \in \mathbb{R}$$

- The alpha scores are then given by applying soft max over the energies.

Evaluating MT: Human scoring is expensive so we want automated

- *BLEU*: uses modified precision score by n-grams (👎 !synonym; !grammar)
  - p_n = (sum (max_count_n_in_any_reference))/ Total MT n-grams: eg.
    - MT_Ok: the cat the cat on the mat. p_2 = 4/6; [the cat, cat the, cat on the, the mat] each appear at most once in each reference.
    - unique_ngrams(MT_Ok, 1) = [the, cat, on, mat]
    - Reference 1: The cat is on the mat
    - Reference 2: There is a cat on the mat.
    - sum(max_countn _in_any_reference) = (2 + 1 + 1 + 1) = 5
    - total_ngrams(MT_Ok, 1) = 7
    - p_1 = 5/7
  - BLEU-4 = BP $\left(\prod_{n=1}^{4} p_n\right)^{\frac{1}{4}}$
  - BLEU-4: Brevity Penalty: BP = min(1, MT output len / Avg Ref. Len)
- Chr-F: Character n-gram F-score: balances
  - character precision: percentage of n-grams in the hypothesis which have a counterpart in the reference.
  - character recall: percentage of character n-grams in the reference which are also present in the hypothesis
- Translation Error Rate (TER): Minimum # of edits required to change a hypothesis into one of the references.
- ROUGE-n: Measures the F-Score of n-gram split references and system outputs
- ROUGE-L: F-Score of the longest common subsequence (LCS) shared between references and system outputs.
- METEOR: Unigram precision & recall with R weighted 9* higher than P
  - Chunk-based evaluation: Fewer n-gram chunks indicate a better match with the reference, while more chunks result in a penalty.
  - Lexical diversity: Accounts for stemming & synonymy, making METEOR more robust than BLEU.
- BERT-Score: Computes pairwise cosine similarity for each token in the candidate with each token in the reference sentence.

N-Gram eval methods have shortcomings where large sections may be permuted in the generated output.

Batching, Padding and Sequence Length: 2 Rules of thumb

- Group similar length sentences together in the batch
- Train your model on simpler/smaller sequence lengths first

Transformers

Transformer Encoder: classification, masked lang modelling, named entity recognition e.g BERT, roBERTa, deBERTa
Transformer encoder-decoder: translation, summarisation, free-form QA
Transformer Decoder: causal language model, text generation
Vision Transformers: visual classification, obj detection, img generation
Multimodal: img captioning/descriptions, text-to-img generation
Other Uses: predicting protein structures, basic reasoning tasks
Transformer contains stack of encoder and decoder layers

Encoder
Each encoder layer has 2 sublayers: (multi-head self-attention (MHA) → residual & norm (R&N)) → (position-wise feedforward network → R&N)
Let S = num words and D = dimensionality
First Sublayer: input (SxD) → MHA (SxD) → R&N (SxD)
Self-Attention: Mechanism that allows each token to attend to all other tokens in the seq. to compute a weigh. representation of the sequence.
Attention(Q,K,V) = softmax(QK^T/sqrt(d_h))*V
To obtain Q,K,V matrices, project encodings through weight matrices
Q = w.W^Q , K = w.W^K, V = w.W^V (all W matrices are shape Sxd_h)
MHA performs self-attention head amount of times which allows different heads to attend to parts of the sequence differently EG original Transformer used 8 heads,meaning we run self-attention 8 times over 8 different Q, K, Vs
Final representation concatenates and projects the self-attention outputs
Each head has dimensionality D/heads

$$LN(\hat{x}_n) = \gamma\hat{x}_n + \beta \quad \gamma \in \mathbb{R}^d \quad \beta \in \mathbb{R}^d$$

Normalization
xhat = (x - mean)/variance
Layer normalization normalizes the features of one sample in one batch
Two key steps: 1. xhat = normalize(x) 2. Transform xhat with learned parameters gamma, beta
Purpose of gamma and beta is to allow the neural network to learn the optimal scale and shift for each feature after normalization, so NN can adjust the range and mean of each feature to better fit the task at hand
Residual Connections help mitigate the vanishing gradient problem by providing a shortcut for information to flow to later layers of a network. The output of an earlier layer is added directly to the output of a later layer so current layer can focus on learning the difference between the two outputs, rather than learning an entirely new transformation

$$\text{FNN}(x) = \max(0, xW_1 + b)W_2 + b_2 \quad W_1 \in \mathbb{R}^{D\times d_{ff}} \quad W_2 \in \mathbb{R}^{d_{ff}\times D} \quad d_{ff} = 2048$$

Position-wise Feedforward Network
A position-wise FF network is an NLP
Position-wise just means it is applying the same transformation to every element in the sequence EG 2 layered network, with ReLU activation

Positional Encoding (PE)
Transformers are position invariant by default (diff to RNNs which are seq). PEs are a way to inject position information into the embeddings. Done by adding a pos vector to embedding of x. PE is a function of 2 inputs: word position and model dimensionality

$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \quad PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Position of the word / Index of d

Decoder
3 sublayers: (masked MHA → R&N) → (cross-attention → R&N) → (position-wise feedforward → R&N).
During *inference*, we perform auto-regressive generation: 1. Source sentence gets encoded via our encoder 2. We feed SOS token to decoder. It then predicts the first word (i.e. I) 3. We append prediction to SOS token, and then use this to predict the next word (i.e. am) 4. Continues until EOS token is predicted i.e. end of the generation.
During *training*, we feed the whole target sequence as input to the decoder and decode sentence in one go (only have to run decoder stack once)
Attention is bidirectional so we mask future tokens during training, so decoder can't access this information (this is known as masked MHA)
Mask is TxT matrix (where T is target seq length) where values in upper triangle are -infinity so illegal to access
Cross-Attention: As we decode a token, we look at all encoded tokens to find out which are more important for the current decoding step. We use the key, value tensors from the last encoder layer, and send them to all the decoder layers. So, query comes from the current decoder layer
Cross attention matrix shape = T x S. Note: K and V are from the encoder.
Transformer does 4 things every training loop: 1. Create source and target mask 2. Run encoder 3. Run decoder 4. Output logits for token prediction
Authors use weight tying in the decoder (Embedding matrix and output projection matrix are shared) and use a decaying learning rate

Pre-Training Models

Contextual Word Embeddings: Meaning of a word can depend on its context, need to take context into account when constructing word representations
ELMo - Embeddings from Language Models: Take a large corpus of plain text and train two language models:
1. One recurrent language model going forward, left-to-right,
2. A second recurrent language model going backward, right-to-left
When we need a vector for a word, combine the representations from both directions (see Machine Translation for more details)
Model Types:
- *Encoders*: Are able to access the whole sequence, using context on both sides of each token.
- *Decoders*: Are able to access context on the left. Language models, good for generating text.
- *Encoder-decoder*: Input is processed using an encoder, then output is generated using a decoder.

BERT: Bidirectional Encoder Representations from Transformers:
Takes in a sequence of tokens, gives as output a vector for each token using stacked encoder blocks. Trained using the following:
- Masked Language Modelling (MLM): Self-supervised training task that needs no labelled data. Hide k% of the input words behind a mask, train model to predict them.
  - Too little masking: too expensive to train
  - Too much masking: not enough context to make predictions
  - Actual strategy used: Pick 15% of input words as training targets - 80% of those are replaced with [MASK] token, 10% are replaced with a random word, 10% are the left as the original word
  - If we masked all chosen words, model wouldn't necessarily learn to construct good representations for non-masked words.
- Second Training Objective: Given two sentences, predict whether they appeared in the original order in the source text. doesn't provide much additional benefit, so it has not been used in newer versions.

Bert Variants: layers: hidden state dim: attention heads: parameters
1. BERT-base: 12 : 768 : 12 : 110 million
2. BERT-large: 24 : 1024 : 16 : 340 million

Trained on: BooksCorpus (800 million words) & English Wikipedia (2,500 million words). Pre-trained with 64 TPU chips for a total of 4 days
Using BERT: BERT-like models give us a representation vector for every input token by removing masked LM head. Represent a whole sequence by vector for end of sequence token.

Pre-Training Models

Multimodal models: Use multiple type of data in a model (text,images, etc)
BERT-Variants: RoBERTa, DeBERTa
- Mask contiguous tokens: SpanBERT
- Distil large model into small one (Model Distilation): DistilBERT, ALBERT
- Use sparse Attention: BigBird, LongFormer
- Application specific: ClinicalBert, MedBert, PubMedBert, BEHRT
- Use existing knowledge graphs with entity embeddings: ERNIE

Pre-training Encoder-Decoder Models: Initial input sequence can be fully attended over using the encoder, getting the maximum information. Then, a dedicated decoder is used for generating the output sequence.
Popular for machine translation: encoder and decoder can have separate vocabularies and focus on different languages.
Techniques:
- Can't use MLM as no direct correspondence between inputs and outputs
- *Prefix Language Modelling:* Source: "I like"; Target "Skibidi Toilet"
- *Sentence permutation deshuffling:* src: "party . good"; trgt: "good party."
- *Token Masking:* src: "I like <mask> toilet"; trgt: "I like skibidi toilet"
- SpanBERT-like objective/ span corruption:
  - Original Text: "Thank you for partying with me"
  - Inputs: "Thank <X> partying <Y>"
  - Targets: "<X> you for <Y> with me"

encoder-decoder models trained with span corruption found to work better than regular decoders
Instruction Training Enc-Dec:
- Encoder inputs are the "natural language" prompts for a given task
  - eg: "Please answer the following questions in 10 steps:"
- *T5:* (Text-To-Text Transfer Transformer) is trained using the span corruption unsupervised objective, along with a number of different supervised tasks.
- *FLAN-T5* (Fine-tuned LAnguage Net) is the same size as T5 but trained on much more data, more languages and 1.8K tasks phrased as instructions.

Pre-training Decoder Models: We can train on unlabelled text, optimizing p_theta(w_t|w_1:t-1). Great for tasks where the output has the same vocabulary as the pre-training data.
Alternative ways of using pre-trained decoders:
1. *Full Fine-tuning*: Supervised training for particular input-output pairs. Or we can put a new layer on top and fine-tune the model for a desired task such as use a "classification head" for a classification task.
   a. One possible task is Natural Language Inference: Label pairs of sentences as entailing/contradictory/neutral.
2. *Zero-shot*: Give the model a natural language description of the task, have it generate the answer as a continuation.
3. *One-shot*: In addition to the description of the task, give one example of solving the task. No gradient updates are performed.
4. *Few-shot*: In addition to the task description, give a few examples of the task as input. No gradient updates are performed.

Advanced Prompting: Improve task performance using prompting.
Chain-of-thought (COT): Ask model to reason about its problem solving.
- Zero shot: "Work through the problem step-by-step"
- N-Shot: "Similar the these <n> examples of step-by-step reasoning, solve the following problems"

Retrieval-Augmented Generation (RAG): Gets factual knowledge from a different system (e.g. Database or "Googling")

Limitations of instruction fine-tuning: Tasks like open-ended creative generation have no right answer, language modelling penalizes all token-level mistakes equally, but some errors are worse than others.
Reinforcement Learning from Human Feedback (RLHF): Optimise model to maximise expected rewards (as assigned by humans) - Expensive af.
- Can model human preferences as a separate NLP problem. ie Train another LM to predict the human score for a given text.
- Also Human judgments are noisy and miscalibrated! Instead of asking for direct ratings, ask for pairwise comparisons, which can be more reliable.

BERT: transformer encoder-only architecture, has multiple layers of self-attention and FF neural networks, capturing bidirectional context simultaneously. Trained using ML and next sentence prediction
GPT (generative pre-trained transformer): transformer decoder-only model, with masked self-attention layers. Trained using causal language modelling, where it predicts the next token based on previous tokens
FLAN (fine-tuned language net): based on transformer architectures like T5, using both encoder and decoder components. Fine tuned using instruction tuning. Designed for zero-shot and few-shot learning.
Parameter-efficient fine-tuning
- *Prompt Tuning*: Include additional task-specific "tokens" in the input, then fine-tune only their embeddings for that particular task, while keeping the rest of the model frozen.
- *Prefix tuning*: In addition to the input, can include these trainable task-specific "tokens" into all layers of the transformer
- *Control Prefixes*: Training different prefixes for each property/attribute you want the output to have. EG the domain or desired length of the text.
- *Adapters*: Inserting specific trainable modules into different points of the transformer, while keeping the rest of the model frozen.
- *BitFit*: Keep most of the model parameters frozen, fine-tune only the biases. Works well, considering it only affects 0.08% of parameters.
- *Low-rank adaptation*: Keep main weights frozen but fine-tune two smaller matrices A and B such that the new weights are going to be W' = W + AB

Why are Transformers popular? *Transfer learning*: Pre-training the model gives us better performance even with fewer downstream training examples
Training bigger models is giving better performance, although diminishing returns, Huge amounts of training data. Fast for their size
AI Risks: AI models can be used for malicious purposes.
Hallucinations - models can pick a wrong output word at any point, potentially leading to made-up claims. LMs can't explain themselves
Biases in the training data - models pick up any biases that appear in the data, some of them harmful
Jailbreaking - Having random users interacting with your model can be dangerous. Model can be convinced to do actions that it is designed not do.

Exam/Tutorial Qns

22. Transformers: 50,000 aligned sentence pairs in source and target language. An arb sentence in the source language has length S, and the corresponding aligned sentence in the target language has length T. Your source language has a vocabulary size of 30,000 (Vsrc), and your target language has a vocabulary size of 20,000 (Vtrg). You choose the hidden state dimensionality of your transformer to be d=500.
    i. dims of emb matrix: R^(Vsrc+vtgt)x500. self att = how much tokens relate to other tokens in the same seq. Att mat: Q^KT \elemof R^SxS
    a. 100% TF. K,V are from the encoder, Q is from the current decoder
    b. Cross attention dims: Q^KT∈R^TxS
23. Words: NLP is fun and interesting Unigram counts: 120 270 200 400 100 What would P(interestingland) change to if we included +K smoothing (with K=1)? Assume there are 100 words in the vocabulary. With +1 smoothing: P(interestingland) = (10+1)/(400+100) = 0.022

PoS (Part of Speech) Tagging

Tagset Tag (example) | ADJ (old)/ADV (slowly)/INTJ (psst)/NOUN (cat)/PROPN (UK)/VERB (run)/PUNCT (.)/SYM (%)/X (other)/PREP (on)/AUX (can)/CCONJ (and)/DET (the)/NUM (one)/PART ('s)/PRON (he)/SCONJ (if)
PoS uses Name Entity Recognition (NER), preprocessing e.g. select ADJs, sequence labelling, neural syntactic/semantic parsing, diction analysis i.e. if LLM generated, keyword/phrase recognition for indexing + searching, spam detection + filtering
Solved for mainstream languages, e.g. Spacy, NTLK $\hat{T} = argmax_T P(T|W)$
Probabilistic POS tagging
Given word sequence W = w1, ..., wn estimate tag sequence T = t1, ..., tn
Compute P(T|W), many-to-many classification
Generative (Bayes): P(T|W) = (P(W|T)*P(T))/P(W) = P(W|T)*P(T)
1. Chain rule + Markov assum (bigram): P(T) ≃ P(t1)P(t2|t1)...P(tn|t(n-1))
2. Word depends only on its tag: P(W|T) ≃ P(w1|t1)P(w2|t2)...P(wn|tn)

Combine to P(T|W) ≃ P(t1)P(w1|t1)*P(t2|t1)P(w2|t2)*...*P(tn|t(n-1))P(wn|tn)
P(ti|t(i-1)) = C(t(i-1), ti)/C(t(i-1)). P(wi|ti) = C(wi,ti) / C(ti)
Table of P(ti,t(i-1)). (p(tag) given another tag) ti is cols. Let <s> be a t(i-1).
Table of P(wi,ti). (p(word) guven a specific ta) wi is cols.
To tag, sequence words as cols, P(ti|t(i-1))*P(wi|ti) as entries. Can pick maxes

Hidden Markov Model (HMM) tagger
$\hat{T} = argmax_T P(T|W) \approx argmax_T \prod_{i=1}^{N} P(w_i|t_i)P(t_i|t_{i-1})$ (emission, transition)
- State sequence (tags) Q = q1q2...qn
- Transition probability matrix, aij is prob. state i → state j. A = a11aij...aNN. aijs over j add to 1 for each i. P(ti|tj)
- Observation sequence (words) O = o1o2...oT. All drawn from vocab. V
- Emission probabilities (likelihood of observation ot from state i) B = bi(ot). P(wi|ti)
- Start state probabilities pi = pi1,...,pin. Add to 1

Assumptions:
1. Markov (bigram approx.)- Tag only depends on past tag. 2. Independence - cur. observation only depends on cur. tag and not prev. observations

P(ti,t(i-1)) table is transition probs, P(wi,ti) table is emission probs
Viterbi - (DP: optimal substructure, overlapping subprobs) recursively find highest path probabilities (instead of taking max to find each tag given) by first building a matrix. Beam search is following k highest paths only.
vt(j) is after t observations, prob. of being in state j. Order O(|T|^2 * |W|)

$v_t(j) = max_{i=1}^{N} v_{t-1}(i)a_{ij}b_j(o_t)$ $v_t(j) = max_{i=1}^{N} v_{t-1}(i)P(q_j|q_i)P(o_t|q_j)$

P of being in every state at time t-1 is already computed, so Viterbi prob is simply the most likely of the extensions of the paths that lead to curr cell
MEMM (Max Entropy Classifier): sequence version of logistic regression classifier - a discriminative model to directly estimate posterior

Viterbi for MEMM

$\hat{T} = argmax_T P(T|W) = argmax_T \prod P(t_i|w_i, t_{i-1})$ $v_t(j) = max_{i=1}^{N} v_{t-1}(i)P(q_j|q_i, o_t)$

Constituency Parsing

Constituency Parsing: Given a sequence generate its syntactic structure
Classical Parsing: Given a grammar, a lexicon, and a new sentence generate the structure of the sentence.
Constituent: sequence of words that behaves as a unit, generally a phrase
Context Free Grammar: 4-tuple (T, N, S, R).
- T = set of terminals • N = set of non-terminals • S = start symbol
- R is set of rules (nonterminal → sequence terminal/non-terminals)

A grammar G generate a language L/ L is recognised by G.
Automtic Parsing: Use a proof system to prove parse tree from words or
- Search Problem: find all possible parse trees for a string using
  - Bottom-up: Words to grammar • Top-Down: Grammar to words

Chomsky Normal Form (CNF): A grammar is in CNF if the rules are of the form X → Y Z or X→ W. We can convert any CFG into CNF deterministically
- VP → V NP PP => (VP → V NEW, NEW → NP PP)
- INF-VP → to VP => (INF-VP → TO VP, TO → to)

CKY algorithm: Dynamic programming algorithm to parse CFG bottom up. Efficient if CFG is CNF; O(n^3 |G|); n = len string, |G| size of CNF grammar G.
- Use a two-dimensional matrix n×n, where n is length of sentence, to encode the possible parses
- Use the upper-triangular portion of the matrix
- Each cell [i, j] in matrix contains the set of non-terminals that represent all the constituents that span positions i-j of the input
- The cell that represents the entire input resides in position [0,n] of the matrix (row x column)

EG: Consider grammar: <S → AB | BC> <A → BA | a> <B → CC | b> <C → AB | a> (| denotes to different options S → AB | BC = (S→AB, S→ BC))
Checking if string "baaba" is in L(G):
CKY for parsing: 1) Add backpointers: augment entries in matrix s.t. each non-terminal is paired with pointers to the matrix entries from which it was derived 2) Allow multiple copies of the same non-terminal to be entered into the matrix to track multiple paths

	b	a	a	b	a
b	[B]	[S,A]	Φ	Φ	[S,A,C]
a		[A,C]	[B]	[B]	[S,A,C]
a			[A,C]	[S,C]	[B]
b				[B]	[S,A]
a					[A,C]

TreeBanks: Datasets of annotated trees for many sentences.
- Penn Treebank: 50K sentences, 1M words

Statistical Parsing: Vannilla parsing doesn't scale for comprehensive grammars. Solutions find most likely parse(s) with statistical parsing.
Statistical Parsing Leaning: Given a treebank we find all the rules and then assign a probability of each of them occurring: q(X → γ) = C(X → γ) / C(X) where C(X → γ) is the number of instances of rule X → γ in the data set and C(X) is the number of times non terminal X is seen. It naturally follows that X ∈ N, Σ_{X→γ ∈ R} q(X → γ) = 1 (ie q is a probability distribution)
Probabilistic CFG (PFCG): CFG extended with a function q which gives the probability of each rule as above.
Probability of tree t: P(t) = Π_{i=1}^{n} q(X_i → γ_i)
Yield: The string of words s is called the yield of any parse tree over s
Best Tree: Out of all parse trees that yield s, pick the parse tree t that is most probable given s: t(s) = argmax_{t ∈ τ(s)} P(t | s)
- argmax P(t|s) = argmax P(s|t)P(t)/P(s) = argmax P(t) as P(s|t) = 1 since a parse tree includes all words in a sentence and p(s) doesn't depend on t.

CNF for PCFG: Modify probabilities s.t the probability of each parse remains the same: (VP → VT NP PP: 0.2) => (VP → NEW PP: 0.2, NEW → VT NP: 1.0)
CKY for PCFG:
- n: number of words in sentence
- w_{i...j} = w_i, ..., w_j = word sequence from i to j
- X_{i...j} = nontermical X dominating w_i, ..., w_j
- Dynamic Table: π[i, j, X] = maximum probability of a constituent with non terminal X spanning words w_{i...j}. Goal is: (max_{t ∈ τ(s)} p(t)) = π[1, n, S]
- Algorithm
  - Base Case: for all i = 1...n, X ∈ N: π[i, i, X] = q(X→ w_i) where q(X→w_i) = 0 if X→ w_i is not in grammar
  - Recursive Case: for all i= i...n-1, j = (i+1)...n, X ∈ N

$$\pi[i,j,X] = \max_{X\to YZ \in R,\ s\in\{i,\ldots,(j-1)\}} (q(X \to YZ) \times \pi[i,s,Y] \times \pi[s+1,j,Z])$$

Constituency Parsing

Evaluating Parser: ParseEval metrics used to evaluate structure
- How much of constituents in the hypothesis parse tree look like the constituents in a gold-reference parse tree.
- A constituent in hyp parse is labelled "correct" if there is a constituent in the ref parse with the same yield and LHS symbol only considering rules from non-terminal to non-terminal

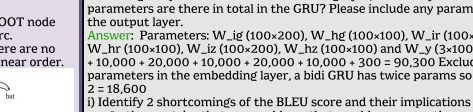
Issues with PCFG:
- Poor independence assumption: Probability estimates of rules computed independently of surrounding context. We can split non-terminals based on context and annotating each tree node with its parent.
- Lack of lexical conditioning: CFG rules don't model syntactic facts about specific words, leading to problems with subcategorization ambiguities, preposition attachment, and coordinate structure ambiguities. Can be fixed by using Lexicalised PCFGs (eg S(saw) → NP(man) VP(saw))

Dependency Parsing

Dependency Parsing: Another form of extracting syntactic structure.
- Connect words in sentence to indicate dependencies. Directional arrow from heads to dependents. Can be typed (arrows are labelled).
- Advantages: Ability to deal with morphologically rich langaugages, head-dependent relationships provide approximation to semantic relationship betweem predicates and arguments, can be used to analyse complexity of a language.

Dependency structure: A directed graph G=(V,A)
- V = set of vertices (words, punctuation, ROOT)
- A = set of ordered pairs of vertices (i.e. arcs)

Dependency Tree: Acyclic dependency structure with a single ROOT node with no incomings arcs. Each vertex has exactly one incoming arc.
Projective vs Non-Projective: A dependency structure where there are no crossing dependency arcs when the words are laid out in their linear order.



Automated Techniques: Dynamic programming is cubic time/not accurate
- Shift Reduce (transition-based)
  - Predict from left-to-right, Fast (linear) but slighhly less accurate, MaltParser
- Spanning Tree (graph-based, constraint satisfaction)
  - Calculate full tree at once, slightly more accurate, slower, MSTParser

MaltParser: Greedy choice of attachment for each word in order, guided by ML classifier, operates as a stack machine, formally 3 data structure
- Stack σ: starts with ROOT
- Buffer β: starts with all words in sentence
- Arc Set A: starts empty

With operations: shift/ left arc/ right arc. Optionally, set of dep. labels for left and right arc actions (~40).

```
Start:  σ = [ROOT], β = w1,...,wn, A = ∅
1. Shift:        σ, (wi | β), A   ⇒ (σ | wi), β, A
2. Left-Arcr:  (σ | wi | wj), β, A ⇒ (σ | wj), β, A ∪ {r(wi, wi)}
3. Right-Arcr:(σ | wi | wj), β, A ⇒ (σ | wi), β, A ∪ {r(wi, wj)}
Finish:  σ = [w], β = ∅
```

Action Decisions: Actions are predicted using a discriminative classifier over each move. Traditionally uses lots of engineered features such as: top of stack word, its POS; first in buffer word, its POS; etc.
Evaluation: Treebanks exist for dependency parsing as well. Can use standard metrics between reference and hypothesis (labelled) arcs.
Neural Parsing: Do dependency parsing as a SEQ2SEQ task where the output sequences are a linearized tree (using brackets). Can use many architectures mentioned previously (RNNs, LSTMs, transformers). Train with cross-entropy. Evaluate as translation (BLEU) or parsing tasks (parseval)
Advanced Neural Parsing: Uses graph-based models.

Exam/Tutorial Qns

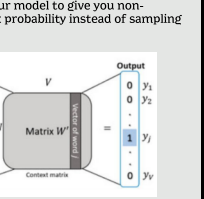
13) The dimensions for word embeddings and GRU states are E and H, respectively. The vocabulary size is V. Provide the size (R?x?) of Wir, Whr and br in Reset Gate: x_t=(E, 1) h_t=(H, 1) Wir=(H, E) Whr=(H, H) br=(H, 1)
14) Dont optimize W.r.t BLEU: because not differentable & extrinsic vs intrinsic.
15) NMT loss is the sum of log-probabilities for every token prediction. How many log-probability terms would we have for the above sample batch with three training examples? batch size * (sum of all target lengths)
16. Computing Attention Matrix: 1) Q = x . W_q, K = x . W_k 2) compute QK^T
3) divide by root num Qs 4) softmax QK^T (e^val / (sum of all row e^vals)
5) V = X.W_v 6) multiply 4) and 5) for the final answer
17) The task of named entity recognition (NER) is defined as seeking "to ... names, organisations and locations". Assume we have 4 labels: N[ame], O[rganization], L[ocation] and E[lse], where E is the label for tokens which are not named entities. Describe how you would use BERT to perform NER using the above 4 possible labels. As part of your answer, explain the type of classification problem this is, the loss function you would use, and the inputs and outputs shapes. d = dimension of word embs from BERT ebc Input (N, d) Output (N, 4) Need to attach linear layers (up to max padding x d --> max padding x 4) on top of the BERT ebc, and softmax across 4 diff classes. Loss - cross entropy loss. multi-class classification.
18) Transformers: As we increase the dim d of the input/model, describe how model params changes 3d^2 for self attention, 2*delta*d_ff ii) Does the number of params in self atten and position-wise feed-forward layers depend on the input sequence length? weight dimensions do not depend on the Sequence length. [D x dh]. Position-wise weight matrix is not dependent on sequence length [D x dff]
19) Which word does the model predict as the most likely context word for "rabbit"? Demonstrate the choice with calculations. Multiply context emb matrix with the vector for rabbit i) softmax: take the curr predi and then it's minus log odds of that 20) Calculate the loss of the model when using negative sampling, with "writes" as the only negative sample. P(D=1 | "eats", "rabbit") + P(D=0|"writes" "rabbit")
20) With a bigram approximation, give the equations for the transition and emission probabilities of an HMM predicting words W given tags T?
transition P(W) = P(w1)P(w2|w1)...P(wn|wn-1); emission = P(T|W) = P(t1|w1)P(t2|w2)...P(tn|wn) if trigram how change? P(W) = P(w1)P(w2|w1)P(w3|w2,w1)...P(wn|wn-1, wn-2) P(T|W) = P(t1|w1)... P(tn|wn) viterbi bigram runtime O(|W|^2 T) deal with prev unseen for PoS: Add an unseen tag that can be used (will also need to add examples to the training set) Setting the tag to the most common tag in the dataset (e.g. noun)
21) Assume we have a unidir (single layered) LSTM, with 100-dim hidden states and 200-dimensional word embeddings. Excluding the emb layer and any output layer, how many parameters do we have in the model? Assume there are no bias terms in the LSTM. (100x200+100x100)x4 = 120000
22. Drawbacks of bleu, what metrics overcome these You need to have enough reference sentences; It only reports the precision score (not recall or f1); It has no vision for further semantic. Meteor, Rouge-N, both measure both recall and precision (with Meteor giving a high weight to recall). And Meteor considers lexical diversity which Bleu doesn't (BERTscore maybe?)

Exam/Tutorial Qns

1) Given the probability table on the right. calculate the probability of the sentence "hamish is the sweetest cat", using a bi-gram model. You can use the information that P(hamish| < s >) = 0.1, and P(< /s > |cat) = 0.1.

	hamish	is	the	sweetest	cat
hamish	0.01	0.4	0.2	0	0
is	0.02	0	0.4	0.08	0.01
the	0.01	0	0	0.45	0.3
sweetest	0.01	0	0.01	0	0.5
cat	0	0.15	0.03	0	0

Answer: Probability = P(hamish| < s >)×P(is|hamish)×P(the|is)×P(sweetest|the)× P(cat|sweetest) × P(< /s > |cat) = 0.1 × 0.4 × 0.4 × 0.45 × 0.5 × 0.1 = = 3.6 * 10^-4
Perplexity = (1/prob)^1/(num words) = (1/3.6 * 10^-4)^1/6 = 3.75 (incl. eos token)
2) In general, provide an expression for the cross-entropy loss (using log base 2) in terms of perplexity. Answer: ppl = 2^H where H = log_2(ppl) where ppl describes perplexity, and H is the cross-entropy loss (bits per character)
3) Consider a feed-forward language model using 4 words of context, 200-dimensional word embeddings and a vocabulary of 20,000 words. Assume no bias terms in the model. How many parameters are there in the model? Include parameters in the embedding layer. Answer: Embedding layer (20,000 x 200), Output layer (4 x 200 x 20,000) Total: 4,000,000 + 16,000,000 = 20,000,000
4) Consider a GRU with: • 200 dimensional word embeddings • 100 dimensional hidden states • 3 output logits (for a multi-class classification task) • No bias terms in the network • GRU is bidirectional (and single layer) Excluding the parameters in the embedding layer, how many model parameters are there in total in the GRU? Please include any parameters in the output layer.
Answer: Parameters: W_ig (100×200), W_hg (100×100), W_ir (100×200), W_hr (100×100), W_iz (100×200), W_hz (100×100) and W_y (3×100). 20,000 + 10,000 + 20,000 + 10,000 + 20,000 + 10,000 + 300 = 90,300 Excluding the parameters in the embedding layer, a bidi GRU has twice params so 90,300 * 2 = 18,600
i) Identify 2 shortcomings of the BLEU score and their implications ii) What are 2 other metrics that you could use that would overcome these shortcomings? How would they help? Answer: i) Brevity Penalty - Penalizes translations shorter than ref translations. Can unfairly lower score of shorter but more accurate translations. Lack of Recall - Primarily based on precision, doesn't take into account recall so doesn't penalize cases where important words are missed. ii) ROUGE - Evaluates both precision and recall. METEOR - also considers lexical diversity: stemming & synonymy matching
5) When using model for inference, you notice that whenever you give it the same prompt, it is always giving you the same response. Why is this happening? How might you encourage your model to give you non-deterministic outputs? Answer: Uses max probability instead of sampling from probability distribution
Word2Vec Skipgram Architecture:



6) Consider the training corpus in Table 1. Create a Binary Naive Bayes model based only on the features 'good', 'great', 'bad' and 'awful'. Use this model to predict the class of the following review: "tom cruise did it again , so great it's a masterpiece , don't listen to the bad reviews , just enjoy this great film". Do not include +1 smoothing.
Answer: P(+) = 4/7 , P(-) = 3/7
P(great|+) = 2/4 , P(great|-) = 0/3
P(good|+) = 1/4 , P(good|-) = 1/3
P(bad|+) = 1/4 , P(bad|-) = 1/3
P(awful|+) = 0/4 , P(awful|-) = 1/3
We find P(sentence, +) = P(sentence|+) × P(+) = P(great|+) × P(bad|+) × P(+) = 2/4 × 1/4 × 4/7
Whereas P(sentence, -) = P(sentence|-) × P(-) = P(great|-) × P(bad|-) × P(-) = 0/3 × 1/3 × 3/7
As 2/4 × 1/4 × 4/7 > 0, the model predicts the example to have the positive class

Training corpus (after some pre-processing)	Class
almost as good as the first top gun	+
fun throughout , definitely recommend this great film	+
awful , tom cruise had no depth once again . why is his acting so bad	-
better than i expected , not bad at all	+
lived up to the name top gun , what a great film	+
wish i had just rewatched the original , this one was nowhere near as good	-
tom cruise should do the stunts and leave the acting to someone else	-

Table 1: Film reviews, categorised as being either positive (+) or negative (-)

7) How many parameters do we have for a 3-class logistic regression model with 10 different Bag of Words input features? Answer: For our weights W we have: 3 x 10 = 30 parameters. We also have 3 bias term parameters. So in total, we have 33
8) State the dimensions for each matrix and vector in the LSTM equations , where we have E dimensional word embeddings and H dimensional hidden states. Answer (bias terms removed from equations): The dimensions of each matrix are: W_ii (HxE), W_hi (HxH), W_if (HxE), W_hf (HxH), W_io (HxE), W_ho (HxH), W_ig (HxE) and W_hg (HxH). The dimensions of each vector are: x_t (Ex1), h_t (Hx1), c_t (Hx1), f_t (Hx1), g_t (Hx1), o_t (Hx1) and i_t (Hx1)
9) Consider training the skip-gram model of word2vec w/settings: BPE vocabulary = 50,000, word emb dim= 300, context window = 5.
i) In the softmax variant, for a single training sample, how many neurons need to be updated in the input weight matrix? *300 neurons in hidden layer, so 300 neurons in input weight mat get updated*
ii) In neg sampling variant, hyperparam k (number of negative samples) is generally decided as a function of the size of the training corpus for word2vec. Explain why larger corpora require smaller k. *we need to update weight for (k+1) neurons (k for negative and 1 for positive) - here (k+1)*300 weight values. With a large corpora, we need more hidden unit to represent the corpus thus using a smaller k could help reduce the computational cost by updating limited number of weights*
10) how to avoid gibberish seqs in n-gram? *use larger n and/or eos token*
ii) how to generate diff follow up seqs? Beam search, at each timestep, eval curr prob for seq and proceed w/ action on the k largest sentences till end. Or randomly pick from the top k. 11) FFN > N Gram for large corpus; FFN > N Gram: Better Generalization, larger context window
12) assume that the vocabulary size is V = 32, 000 and the dimension for embs is E = 600. Construct a FFLM that has a context window of C = 3 words. We approx the context X<t by concating the prev word embs. assume net has only one emb layer and one output layer, what is the total number of learnable ps? look at FFN diag in LM 2 section. emb Layer: V*E=32000*600 Output Layer: CE*V=3*600*32000 Total: 76800000 parameters how to increase C without increasing params too much emb to lower dims w/ enc; pca; share weights

Natural Language Inference is a classification task with 3 classes.