Stanford CS224N: NLP with Deep Learning

Table of Contents

Lecture 1: Introduction and Word Vectors

Human Language and word meaning

Problem: One-hot encoding of words makes vectors orthogonal (no natural notion of similarity). It is also high-dimensional (size of a dictionary).

> Distributional semantics: A word's meaning is given by the words that frequently appear close-by (we refer to it as a *context*)

Solution: We use distributed representation to build a word vector. It now is a non-sparse numeric vector with dimensionality way less than the dimension of a dictionary. A way to achieve it is *Word2vec (Mikolov et al. 2013)*

> Note: Distributional vectors are also referred to as word embeddings. Similar words are expected to have close vectors.

Word2vec

Idea: For a given corpus of text, given center word c and its context (outside) words o, use similarity of word vectors for c and o to calculate $p(o|c)$ (or $p(c|o)$). Do it iteratively for the entire corpus, for each word c in the corpus.

Objective function

Likelihood:

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} p(w_{t+j}|w_t, \theta)$$

Where $t \in 1, \ldots, T$ is a position in the text corpus, m is a (half of) window/context size, \theta are the parameters of the model, which are the vector representations of the words (its components).

The objective function $J(\theta)$ that we want to optimise is the negative log likelihood:

$$J(\theta) = -\frac{1}{T} logL(\theta)$$

where negative sign turns the problem into minimization and $1/T$ scales the objective to be independent of the size of the corpus.

In order to calculate $p(w_{t+j}|w_t, \theta)$ we use two vectors per word w:

- v_w when w is a center word
- u_w when w is a context word

> Remark: Remember that each word is associated with two vectors, so $\theta \in R^{2dim(V)}$

Then, for a center word c and its context o:

$$p(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w \in V} exp(u_w^T v_c)}$$

where V is a vocabulary.
Nominator is a similarity between the context word vector and the center word vector and the denominator normalizes the similarity over the entire vocabulary.

Note: We're using **softmax** here because it guarantees that the output is the probability distribution.

We then optimize the objective function by finding $\frac{\partial}{\partial v_c} J(\theta)$ and doing gradient descent on it.

Vector composition

Word2vec exhibits compositionality, i.e., adding two word vectors results in a vector that is a semantic composite of the individual words, e.g., $king - man + woman = queen$

Lecture 2: Word Vectors and Word Senses

Gradient Descent

Recall the equation for gradient descent:

$$\theta^{t+1} = \theta^t - \alpha \nabla_\theta J(\theta)$$

where θ^t denotes the vector of parameters at iteration t.

Stochastic Gradient Descent

Problem: $J(\theta)$ is a function of all windows in the corpus, which makes it very expensive to compute
Solution: SGD - sample windows one at a time and perform parameter update for each of the windows one at a time (assume that a sampled $J(\theta)$ is an estimate of the actual $J(\theta)$). Or use mini-batch GD - compute $J(\theta)$ based on a couple of windows and then perform an update.

Word2vec variants

There are two variants of Word2vec:

- Skip-grams (SG) model - predict context (outside) words (position independent) given center word
- Continuous Bag of Words (CBOW) - predict center word from (bag of) context words

Skip-gram model with negative sampling (HW2)

Problem: In the equation presented previously: $p(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w \in V} exp(u_w^T v_c)}$ the denominator is calculated for every word in the vocabulary, **which is expensive**.

Solution: Use negative sampling, i.e. sample K negative samples (using word probabilities) according to the probability distribution. The original paper uses $P(w) = U(w)^{3/4}/Z$, where U is a unigram distribution (i.e. count of each word in the original corpus), and $3/4$ decreases the "weight" of words that appear ofthen (*and, the, of* etc.) and Z is the normalization term.

Prior to Word2vec

Prior to Word2vec people would build a **coocurrence matrix** X, which, for a given window size m, would count, for each word i in the corpus, the coocurrence of each other word. For example, if X_{ij}

will be the count of how many times word j was in the context (window) of word i (note X is symmetric).

Then, X would be factorized into $U\Sigma V^T$ with U, V^T being orthonormal (SVD), in order to reduce the dimensionality (by throwing away the smallest singular values).

This led to the dichotomy of two methods: count based vs. direct prediction (e.g. skip-gram/CBOW)

Note on SVD

Factorizing X into $U\Sigma V^T$ and getting rid the smallest singular values (we remove vectors of U right-to-left and rows of V^T bottom-up), retaining only k of them leaves us with $\hat{X}$, which is the best $rank(k)$ approximation of X w.r.t. least squares.

GloVe - Global Vectors for Word Representation

GloVe is meant to unify the *count methods* (i.e. based on some count matrices) and *prediction methods* (i.e. based on loss-based estimation).

Insight: Ratios of co-ocurrence probabilities can encode meaning components. e.g. given $P(x|ice)$ and $P(x|steam)$ (*ice* and *steam* being some example words from V) for all $x \in V$ (normalized co-ocurrence matrix), $\frac{P(x|ice)}{P(x|steam)}$ can have meaning.

Second insight: We can capture ratios of co-ocurrence probabilities as linear meaning components in a word vector space by using a log-bilinear model:

$$w_x \cdot w_y = logp(x|y)$$

so for $w_y = w_a - w_b$ we have:

$$w_x \cdot (w_a - w_b) = log\frac{p(x|a)}{p(x|b)}$$

which means that a vector difference ($w_a - w_b$) this turns into a ratio of co-occurrence probabilities.

> Remark: Word2vec and GloVe are examples of context-free models because they generate a single word embedding for each word in the vocabulary. In contrast, e.g. BERT takes into account the context for each occurrence of a given word. For instance, whereas the vector for "running" will have the same word2vec vector representation for both of its occurrences in the sentences "He is running a company" and "He is running a marathon", BERT will provide a contextualized embedding that will be different according to the sentence.

How to evaluate word vectors?

There are two methods:

1. Intrinsic - evaluation on a specific/intermediate task (e.g. how synonyms are close together, are parts of speech correctly classified)
2. Extrinsic - Evaluation on a real task

> Remark: Syntactic analogies are about syntax, e.g. *good - better - the best*, whereas Semantic analogies are about meaning, e.g *king - man*

Linear Algebraic Structure of Word Senses with Applications to Polysemy

A given word might have many meanings, with each meaning being de-facto a separate word (*sense* of a word). What we usually find (i.e. the vector that we find from Word2vec) is the superposition (linear combination) of those senses, weighted by their frequency in a corpus. Because of the sparse coding we can actually separate out the components given the resulting (after superposition) vector.

Lecture 3: Neural Networks

Classification

Given training data $\{x_i, y_i\}_{i=1}^N$, $x \in R^d$, we want to classify each x into of of C classes. The traditional approach is to assume x_i are fixed and train a softmax/logistic regression weights $W \in R^{C \times d}$ to determine a decision boundary (hyperplane).

Prediction: for each test x, predict:

$$p(y = c|x) = \frac{exp(W_c x)}{\sum_{c=1}^{C} exp(W_c x)}$$

where W_c is the row of the weight matrix W corresponding to the class c

Cross entropy

Let p be a true probability distribution (for a given sample that distribution is 1 for a true class and 0 elsewhere) and our computed model probability be q. The cross entropy is:

$$H(p, q) = -\sum_{c=1}^{C} p(c) log q(c)$$

where C is the number of all classes.

> Remark: Since p is one-hot encoded, the only term of the cross entropy is the negative log probability of the true class.

> Remark 2: In semi-supervised learning we are often given a probability distribution over labels for a given sample

So cross-entropy loss is about minimising our objective function loss by minimising difference between true probability distribution (training labels) and our probability distribution from a trained model.

Finally, our cross-entropy objective function (for the entire dataset with N samples) is:

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} log\left(\frac{exp(W_c x)}{\sum_{c=1}^{C} exp(W_c x)}\right)$$

In general ML, θ usually is a column of weights W, i.e. $\theta \in R^{C \times d}$

Classification with word vectors / representation learning

In NLP we usually learn both: weights W and word vectors x. Learning W means learning conventional parameters and learning x is called *representation learning* (because x is a word representation).

So when optimising our objective we end up with the following gradient vector:

$$\nabla J(\theta) = \begin{bmatrix} \nabla W_1 \\ \vdots \\ \nabla W_C \\ \nabla x_{aardvark} \\ \vdots \\ \nabla x_{zerba} \end{bmatrix} \in R^{C \times d + V \times d}$$

Named Entity Recognition (ENR)

> ENR is about **finding** and **classifying** names in text. Classifying is about finding a class for a given *name*, e.g. *location, organisation, person*.

Task: Classify a word in its context window of neighbouring words
Solution: (Collobert & Weston, 2008/2011) - we train a model that returns a score s for a given window of words. Score is high if a middle word in a window is a given entity class (e.g. *location*). A given window is a contatenation of constituent word vectors.

Final model (for a single entity class) is:

$$s = U^T f(Wx + b)$$

where U, W, b are trained. $x \in R^{wd}$ is the concatenated vector of words in a window of size w (each word has dimensionality d), f is a nonlinear activation function, $W \in R^{d \times wd}$ is a weight matrix (each row in a matrix corresponds to one window word of x), U is a vector $U \in R^{d \times 1}$.

> Remark: If we want to be able to recognize more entity classes we need to learn c such models (and then pass the output scores through softmax)

Gradients / Multi-variable calculus

Given f with 1 output and n inputs: $f(x_1, \dots, x_n)$ its derivative (row vector) is:

$$\frac{\partial f}{\partial x} = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]$$

Given f with m outputs and n inputs $f(x) = [f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)]$ its gradient is a **Jacobian matrix** $J \in R^{m \times n}$, which generalizes the notion of the gradient:

$$\left(\frac{\partial f}{\partial x}\right)_{ij} = \frac{\partial f_i}{\partial x_j}$$

i.e. each row of J corresponds to some ith output component of f and each column is w.r.t each input x_j

Chain Rule for multivariable calculus

In multivariable calculus the chain rule is simply done by multiplying Jacobians. For a given $h = f(z), z = Wx + b$ (with z being a vector) we have:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial z}\frac{\partial z}{\partial x}$$

> Remark: The activation function is a function $h : R^n \to R^n$ so its Jacobian is $J \in R^{n \times n}$. However, the Jacobian is always diagonal as $\frac{\partial h_i}{\partial z_j} = 0$ for any $i \neq j$ (i.e. the activation does not combine multiple components to do the transofrmation of a single component). Thus, if f is an activation function we have:
>
> $$\frac{\partial}{\partial z}(f(z)) = diag(f'(z))$$

Other Jacobians

$$\frac{\partial}{\partial x}(Wx + b) = W$$
$$\frac{\partial}{\partial b}(Wx + b) = I$$
$$\frac{\partial}{\partial u}(u^T h) = h^T$$

Re-using computation

If we tried to compute $\frac{\partial s}{\partial W}$ and then $\frac{\partial s}{\partial b}$ we'd essentially need to compute $\frac{\partial s}{\partial h}\frac{\partial h}{\partial z}$ twice. To save computation we, in backpropagation, compute it once and then reuse. We call it then δ - a local error signal, which represents, for a given layer in a NN, whatever is happening above it.

Multivariable-calculus conventions

In ML we follow so called *shape convention*, i.e. we make sure that the Jacobian has a shape that allows us to the update rule (GD). This is in contrast to e.g. numerator convention or denumerator convention.

Turns out that, in order to follow the shape convention we have:

$$\frac{\partial s}{\partial W} = \delta^T x^T$$

> Remark: There is a disagreement between Jacobian form (which makes the chain rule easy) and the shape convention, which makes implementing SGD easy. We expect answers to follow the shape convention, but we compute them using Jacobian form.

Lecture 4: Backpropagation

> Notation remark: In a weight matrix W_{ij}, i corresponds to the index of the output and j corresponds to the index of the input.

Computation graph is a way to represent neural network euqations in form of a graph, where source nodes indicate inputs and interior nodes indicate operations.

Each node performs some transformation, let's say we have node f: $f = h(z)$ (e.g. nonlinearity). Then h is being transformed by some other nodes to get final s (score). The chain rule for this node is $\frac{\partial s}{\partial z} = \frac{\partial s}{\partial h}\frac{\partial h}{\partial z}$, where $\frac{\partial h}{\partial z}$ is called a **local gradient**, $\frac{\partial s}{\partial h}$ is an **upstream gradient** and $\frac{\partial s}{\partial z}$ is a **downstream gradient**.

> Remark: If a node (say f) has different output branches (i.e. a result from it goes to more than one other node), in the backpropagation case, in order to calculate the gradient at that node we sum the upstream gradients, i.e.
>
> $$\frac{\partial s}{\partial x} = \frac{\partial s}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial s}{\partial b}\frac{\partial b}{\partial x}$$

where: x represents a value coming from a given node, s is the ultimate output (score) of the network and $\frac{\partial s}{\partial x}$ represents how much would the output of the network change given a small change in x for that one node. a, b are the operations/values coming out of the immediate upstream nodes for node f (i.e. the ones that take as input the output of that node, x).

> Remark: If done correctly, Forward pass and Backward pass have the same computational complexity

Automatic Differentiation

- Gradient computation can be automatically inferred from the symbolic expression of the forward propagation

- Each node type needs to know how to compute its output and how to compute the gradient w.r.t. its inputs given gradient w.r.t. its output

> Theano used to do symbolic differentiation. Modern DL frameworks like Tensorflow, PyTorch require the client to implement both `forward` and `backward` routines.

Nonlinearities

Logistic/sigmoid functions are, in practice, not used often as a nonlinearity but only in the output layer to produce a probability distribution.

Other nonlinear functions include:

- $tanh$, which is essentially a scaled version of the sigmoid function $tanh(x) = 2\sigma(2x) - 1 \in]-1, 1[$
- $hardtanh(x)$
- ReLU (rectified linear unit): $ReLu(x) = max(x, 0)$
- Leaky ReLU / Parametric ReLu - $y = ax$ for $x < 0$ for some small a. Leaky ReLu uses $a = 0.01$

Parameter Initizalization

Problem: Symmetry prevents learning/specialization (of neurons), so initializing all weights to 0 does not work
Solution: Initialize hidden layer biases to 0 and output biases to optimal value if weights were 0 (e.g. mean target or inverse sigmoid of mean target). Initialize all other solutions to $Uniform(-r, r)$ for some small r.
Solution 2: Use Xavier initialization which uses different variance (inversely proportional) for the distribution of W depending on the sizes of previous layer size (fan-in) n_{in} and next layer size (fan-out) n_{out}:

$$Var(W_i) = \frac{2}{n_{in} + n_{out}}$$

The rational behind the Xavier initialization is that we will be using a given weight proportionally many times to the size of neighbouring layers.

Lecture 5: Dependency Parsing

There are two views of linguistic structure:

- Context-free grammars (CFGs)
- Dependecy structure

Context-free grammars (CFG)

Phrase structure organizes words into nested constituents:

- NP (Noun Phrase) = Determiner (a.k.a Article) + (Adjective) + Noun
- PP (Prepositional Phrase) = Preposition + NP
- VP (Verb Phrase) = Verb + PP

These are examples of context-free grammar rules used to describe the structure of the language.

Dependency structure and Dependency Grammar

Dependency structure shows which words depend on (modify or are arguments of) which other words.

> Dependency syntax postulates that syntactic structure consists of relation between lexical items, normally binary assymetric relations ("arrows") called dependencies. In another words we take a sentence and build a tree with each of the words using "arrows" to denote the dependencies between them.

Ambiguities

Prepositional phrase attachment ambiguity is an ambiguity resulting from two meanings of a sentence, e.g. "Scientists count whales from space" - it's ambiguous where the prepositional phase "from space" attaches - to "scientists" or to "whales"

Coordination scope ambiguity - e.g. "Shuttle veteran and longtime NASA executive Fred Gregory appointed to board" - it's ambiguous how many people got appointed to board (1 or 2?).

Verb Phrase (VP) attachment ambiguity - e.g. "Mutilated body washes up on Rio beach to be used for Olympics beach volleyball"

Dependency parsing

Dependency parsing is about finding the tree (resolving dependencies of words in a sentence). A sentence is parsed by choosing for each word what other word (including root) is it a dependent of.

Methods of Dependency parsing:

- Dynamic Programming
- Graph algorithms (creating an MST for a sentence)
- Constraint Satisfaction (remove edges that do not satisfy hard constraints)
- "Transition-based parsing" or "deterministic dependency parsing" (greedy choice of attachments guided by good ML classifiers)

Recently the "Transition-based parsing" or "deterministic dependency parsing" have been used by training ML models to do the parsing through classication

Lecture 6: Language Models and RNNs

Language model is a model that, given a sequence of words $x^{(1)}, \ldots, x^{(t)}$ computes a probability distribution over the next word, $x^{(t+1)}$:

$$p(x^{(t+1)}|x^{(1)}, \ldots, x^{(t)})$$

where $x^{(t+1)} \in V$

N-gram Lanaguage model

Task: Learn a language model
Solution (pre deep-learning): learn an n-gram language model (by counting n-grams in a large corpus of text)

> Def: n-gram is a chunk of n consecutive words (the order matters)

$$p(x^{(t+1)}|x^{(1)}, \ldots, x^{(t)}) = \frac{p(x^{(t+1)}, x^{(t)}, \ldots, x^{(t-n+2)})}{p(x^{(t)}, \ldots, x^{(t-n+2)})}$$

where nominator is a probability of an n-gram and the denominator is a probability of a n-1 gram (note the expressions on the RHS don't have condition in them).

Problems:

- In n-gram we assume that the probability of the next word only depends on previous $n-1$ words, which potentially throws away context information
- Sparsity - model will assign a zero probability to n-grams that did not exist in the corpus of text, even though they might be valid
- Another sparsity problem - if the denominator in the probability expression is zero, we need to *back-off* to a n-2 gram (ignore the $n-1$ th word and then try and compute the probability). Similarly, if n-2 gram has the same problem we need to back-off to n-3 gram model etc.
- Sparsity increases with n

Fixed-window neural language model

An alternative to n-gram language model we can build a neural model, which first, finds word embeddings for the constituent words, then concatenates them and then traing an NN based on those to output a probability distrubution over the next word.

The main problem with this approach is the fixed-window size, which **is the motivation behind the RNNs**.

Recurrent Neural Networks

RNN introduces a concept of a hidden state $h^{(t)} \in R^d$ where d is the dimensionality of the input: $e \in R^d$ (i.e. the embedding). Note the original one-hot encoded has the dimensionality of the entire

vocabulary: $x \in R^V$.

Now the network has two parameters: W_e and W_h. W_e is applied to e and W_h to h:

$$h^{(t)} = \sigma(W_h h^{(t-1)} + W_e e^{(t)} + b_1)$$

The output distribution can be received by passing h through softmax at any point in time (we can e.g. pass a couple of words and only after that compute the output).

> Remark: At each point in time hidden state is the combination of the current input and the hidden state from the previous iteration, so the network feeds into itself

> Remark 2: RNN applies the same weights on each step

> Remark 3: When we feed an input (e.g. a sentence) into an RNN, at a given timestamp, the hidden cell state can be viewed as a *representation* of that input at that particular timestamp.

RNNs can be used for POS (part of speech) tagging, named entity recognition or sentiment classification. You can also combine the hidden states of a input (e.g. an average) and this combination might actually have some meaning (e.g. if an input is a question, the combination of hidden states might be an encoding of that question), which could be used in question answering. So RNN can be treated as an encoder.

Advantages of RNNs

- Any length of input allowed
- (In theory) Hidden state has access to information from many steps back
- Model size does not increase for longer input

Disadvantages of RNNs

- Hidden states computation is sequential (cannot be parallelized)
- (In practice) hard to capture information from many steps back

Training RNNs

In order to train an RNN we:

1. Feed a corpus of text into RNN and compute $\hat{y}^{(t)}$ for every step t
2. Compute a loss function at time t using predicted next word and the true next word using cross entropy - we get $J^{(t)}(\theta)$
3. Calculate average loss $J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)$
4. Backpropagate using *Backpropagation through time* algorithm

> Remark: During training, we feed the gold (aka reference/true) target sentence into the decoder, regardless of what the decoder predicts. This training method is called **Teacher forcing**.

> Reminder: Decoder works by predicting the next word given a current input, which is initialized as $< start >$ token. It can, at each step, assume the current input word is an $argmax$ over all most likely words produced in the previous timestep (greedy decoding), or it can consider k of most probable partial sequences (hypotheses) at each timestep (beam search decoding).

Evaluating language models

The standard evaluation metric for language model is called **perplexity**:

$$perplexity = \prod_{t=1}^{T} \left(\frac{1}{p_{LM}(x^{(t+1)}|x^{(1)}, \ldots, x^{(t)})} \right)^{1/T}$$

which is equal to the exponential of the cross-entropy loss, i.e. $exp(J(\theta))$. *LM* stands for a Language Model.

> Note: Lower perplexity is better

Conditional language model

Conditional language model is a model to which we provide an input, e.g. voice recording, and then we use it as a normal language model. Its behaviour, however, is different for each of the inputs provided as a *condition*.

Lecture 7: Vanishing Gradients, Fancy RNNs

Summary

Vanishing Gradient problem motivates the use of GRU (Gated Recurrent Unit) and LSTM (Long-term short memory).

Vanishing Gradient

The intuition behind the Vanishing gradient is that the gradient of the loss w.r.t. hidden layer gets exponentially smaller as we "unroll" backpropagation to the past (due to chain rule). This will happen if the largest eigenvalue of W_h is less than 1. On the other hand, if the largest eigenvalue is greater than 1, we have a problem with **exploding gradients**.

Vanishing gradient problem causes RNN-LMs (Language Models) to learn sequential recency rather than syntactic recency. For example, in a sentence "The writer of the books _", RNN would rather choose "are" rather than "is" (if given those two options) because "are" is sequentially recent to "the books", as opposed to "is", which syntactically recent to "the writer".

The problem with RNN is that $h^{(t)}$ gets constantly rewritten in each iteration. The idea is to add a new state "memory", which is a motivation behind LSTMs.

LSTMs

Long-short term memory networks introduce the concept of cell state $c^{(t)}$ which has the same dimensionality (say n) as $h^{(t)}$ but is meant to store long-term information.

- LSTM can erase, write and read information from the cell
- the selection of which information is erased/written/read is controlled by three corresponding gates (forget gate $f^{(t)}$, input gate $i^{(t)}$ and output gate $o^{(t)}$), all being vectors with the same size n
- each component of the gate vector $\in]0, 1[$ where 0 denotes closed gate and 1 denotes "open" gate
- gate vectors change dynamically at each step

Gates

- Forget gate $f^{(t)}$ controls what is kept vs. what is forgotten. It is used with $i^{(t)}$ to build new cell state $c^{(t)}$ at each iteration based on $\tilde{c}^{(t)}$ and $c^{(t-1)}$ where $\tilde{c}^{(t)}$ is a proposed new cell content
- Input gate $i^{(t)}$ is combined with $f^{(t)}$ to build $c^{(t)}$ (see above)
- Output gate $o^{(t)}$ controls what is being output to the hidden state (i.e $h^{(t)}$ depends on $o^{(t)}$ and $c^{(t)}$)

> Note that the actual output form the LSTM at each timestep is the hidden state $h^{(t)}$ after softmax.

The forget gate allows to easier perserve information over many timesteps if it's set to remember everything.

> Remark: As of 2019, Transformers have become dominant in certain tasks as compared to LSTMs

The secret behind the success of LSTMS is that when we calculate $c^{(t)}$ we, rather than multplying, we add some nonlinearity to $c^{(t-1)}$. There is a direct, linear connection between $c^{(t-1)}$ and $c^{(t)}$ which makes backpropagation work well (no vanishing gradients).

Exploding gradient

The problem with exploding gradient is that the update term in SGD becomes too big, so an update makes to big of a "step" at each iteration (during training it can manifest itself as `NaN`s or `Inf`s).

Possible solutions to the exploding gradient problem include:

- Gradient clipping (clip gradient at some predefined threshold)

Gated Recurrent Units (GRU)

GRUs (2014) were proposed as a simpler alternative to LSTMs. In each timestep t we have input $x^{(t)}$ and hidden state $h^{(t)}$ but no cell state.

Gates

- Update gate $u^{(t)}$ - controls what is being output to the hidden state
- Reset gate $r^{(t)}$ - controls what part of previous hidden state are used to compute new content

We use $r^{(t)}$ to compute *new hidden state content* $\tilde{h}^{(t)}$ (candidate for a hidden state) and then use it with $u^{(t)}$ to define $h^{(t)}$.

GRU deals with vanishing gradient problem by having the update gate $u^{(t)}$. In the extreme case, if it is a zero vector then $h^{(t)}$ does not change in a given iteration.

GRU vs LSTM

- GRU is faster to compute and has less parameters
- No conclusive evidence that one consitently performs better than the other

Fancy RNNs

Residual connections (ResNet)

A possible solution to the vanishing/exploding gradient problem is adding a residual (skip) connections, i.e. connections between non neighbouring layers. This allows the gradient to have shorter path to deep layers during propagation.

DenseNet is an extreme example where every layer is connected to every another layer.

Bidirectional RNN

When doing a sentiment analysis of the sentence *"the movie was terribly exciting"*, the hidden state of the RNN after feeding in the word "terribly" is called its *contextual representation*. This contextual representation however lacks the context of the sentence to the right of the word "terribly".

Bidirectional RNN is about stacking two RNNs - forward and backward (reverse order of feeding in sentences). The hidden state at each timestep is a contatenation of hidden states of both stacks.

Lecture 8: Translation, Seq2Seq, Attention

Overview

Sequence-to-sequence is a neural architecture, which solves the problem of Machine Translation. Attention is a new neural technique used to improve Seq2Seq.

Machine Translation

Machine Translation is the task of translating a sentence x from one language (source language) to a sentence y in another language (target language)

Statistical Machine Translation (1990s - 2010s)

Statistical Machine Translation is about learning a probabilistic model p that aims to find y such that it maximizes the $p(y|x)$:

$$argmax_y \ p(y|x)$$

where x is a sentence in a source language and y is a translated sentence (in a target language). Using Bayes theory we can convert it into:

$$argmax_y \ p(y|x) = argmax_y \ p(x|y)p(y)$$

where $p(x|y)$ is referred to a **Translation Model** (how words and phrases should be translated - learnt from parallel data) and $p(y)$ is referred to a **Language Model** (understanding target language - learnt from monolingual data).

> Note: *Parallel data* is a corpus with pairs of sentences in both languages

The premise behind splitting into two probability distributions is a "division of labour", which is easier to learn.

Alignment

In reality, instead of learning $p(x|y)$ we try to learn $p(x, a|y)$ as our Translation Model, where a is called an **alignment**, and it models how words in x map to words in y for a given sentence x and its translation y (e.g. sometimes one word in source language maps to many words in target language e.g. "entarte" in fr. maps to "hit with a pie" in eng). The alignments can be one-to-one, one-to-many, many-to-one, many-to-many.

We use some heuristic search algorithm to search for the best translation (y that maximizes the $argmax$), discarding too low probability hypotheses.

Neural Machine Translation

In 2014, Neural Machine Translation (NMT) was discovered. NMT allows to do machine translation using a single neural network. The architecture of that NN is called **sequence-to-sequence** (Seq2seq) and involves two RNNs.

> Remark: Google Translate switched from SMT to NMT in 2016

Seq2seq

Inference

Seq2seq uses two RNNs: encoder RNN and decoder RNN. The way it works is: we feed an input sentence - list of words (to be specific - their embeddings) x to the encoder (one by one) and, at the end, we save the last hidden state. Then, we use that state as an initial hidden cell state in the decoder RNN and we feed it with a token $< start >$ as the first input. Then, we retrieve the $argmax$ of its

hidden state output (hidden state output is a probability distribution over all words in a target language dictionary) - a single word - and feed it back as input until the decoder RNN returns (in its hidden cell state) the $< end >$ token. The sequence of hidden cell states (after $argmax$) of the decoder RNN's forms a translated sequence y.

In short:

- Encoder RNN produces an encoding of the source sentence
- Decoder RNN is a Language Model that generates a target sentence, conditioned on encoding (from the encoder)

> Remark: You need a separate word embedding for each language

Seq2seq is a **conditional language model**. It's a language model because the decoder is predicting the next word of a target sequence y, and its a conditional model because the predictions are conditioned on the source sentence x.

> Remark: Unlike SMT we directly learn $p(y|x)$ in Seq2seq

Training Seq2seq

At training time, when training the decoder, it is being fed with the true target output at each step, and not based on its own output from the previous step. The loss is computed for each true word (target) and the corresponding decoder prediction (hidden cell state) and averaged out.

Seq2seq use-cases

- Summarization (long text $\rightarrow$ short text)
- Dialogue (previous utterances $\rightarrow$ next utterance)
- Parsing (input text $\rightarrow$ output parse as sequence)
- Code generation (natural language $\rightarrow$ Python code)

Beam search decoding

Problem: The problem with using $argmax$ at each step when decoding the translation is that the sum of $argmax$es is not necessarily an $argmax$ of the entire output (i.e. greedy might give a suboptimal translation).

> Example: Say we translate the sentence "il a m'entarte" (he hit me with a pie) from fr. to eng. We feed the decoder (whose hidden state had been initialised with an output of the encoder for that sentence) with $< start >$ - its hidden state argmax (correctly) gives "he". Then, we feed "he" back into it and it outputs "hit". Then, we feed "hit" back and it outputs "a" instead of "me" which has the second highest score. In the next iteration "he hit me with" would have had a higher total score than "he hit a with" but the have commited ourselves to "a" already.

Solution: Use a beam search - in each step of the decoder, keep track of k most probable partial translations (hypotheses), where k is a beam size.

Evaluating Machine Translation

We use **BLEU** (Bilingual Evaluation Study), which compares machine-written translations with human-written translations and compute a similarity score based on n-gram precision (with a penalty for too-short translations - otherwise the system might learn to optimize for n-gram precision by outputting short sentences)

Remaining Problems in NMT

- Out-of vocabulary words (you're trying to translate a word that does not exist in the vocabulary)
- Domain mismatch between train and test data
- Maintaining context over longer text
- NMT picks up biases from training data (can e.g. introduce gender to the translation but there is no gender specified in the original sentence)

Attention

Attention is by far the best improvement made to NMT.
Problem: In Seq2seq, there exists an *information bottleneck* - an entire input sentence needs to be represented by a single vector - last hidden state of an encoder
Idea: On each step of the decoder use a direct connection to the encoder states to focus on a particular part of the source sequence
Solution: At each step of the decoder we take its hidden state (which we would have returned as an output in a vanilla Seq2Seq) and calculate a dot product with each of the hidden states of the encoder (previously we'd discard all encoder's hidden states but the last one). This gives us an **attention score** (i.e. scalar) for each of those encoder hidden states. Then we squeeze them through softmax to get an **attention distribution** and then calculate an **attention ouput** by calculating a weighted sum of encoder hidden states weighted by the attention distribution. Lastly, we concatenate the attention output with **decoder hidden state** to compute the output.

> Remark: The attention distribution provides a "soft" alignment "for free". Soft means here that we can see a distribution of how decoder word relates to source words (as opposed to binary yes/no in SMT if the words are related to each other or not).

General Definition (Attention)

> Given a set of vectors (referred to as **values**), and a vector **query**, attention is a technique to compute a weighted sum of the values, dependent on the query.

In another words, attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query)

> Remark: We sometimes say that the *query attends to the values*, e.g. each decoder hidden state (query) attends to all the encoder hidden states (values)

Attention variants

There are several ways to compute the attention score $e \in R^N$ from $h_1, \ldots, h_N \in R^{d_1}$ (e.g. encoder hidden states) and $s \in R^{d_2}$ (e.g. decoder hidden states).

> Note: It might not be necessarily the case that $d_1 = d_2$

- Dot-product attention: $e_i = s^T h_i \in R$
- Multiplicative attention: $e_i = s^T W h_i \in R$, where $W \in R^{d_2 \times d_1}$ needs to be learned
- Additive attention: $e_i = v^T \ tanh(W_1 h_i + W_2 s) \in R$, where $W_1 \in R^{d_3 \times d_1}, W_2 \in R^{d_3 \times d_2}, v \in R^{d_3}$ are weight matrices (vector) respectively. d_3 is a hyperparameter (**attention dimensionality**).

Lecture 9: Practical Tips for Projects

Tips for learning RNNs

1. Use LSTM or GRU
2. Initialize recurrent matrices to be orthogonal
3. Initialize other matrices with a sensible (small) scale
4. Initialize forget gate bias to 1 (default to remembering)
5. Use adaptive learning algorithms: Adam, AdaDelta etc.
6. Clip the norm of the gradient (1 - 5 is a good threshold to start with)
7. Either only dropout vertically or use Bayesian Dropout

Lecture 10: Question Answering

> **Stanford Question Answering Dataset (SQuAD)** is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage, or the question might be unanswerable.

SQuAD 2.0 introduced (in addition to SQuAD 1.0 dataset) questions which have no answer.

SOTA for SQuAD 2.0:

- BERT/ALBERT
- Stanford Attentive Reader (A Thorough Examination of the CNN/Daily Mail Reading Comprehension Task)/Stanford Attentive Reader++

- BiDAF (Bi-Directional Attention Flow for Machine Comprehension)
- FusionNet
- ELMo

FusionNet experiments with different representations of attention. ELMo and BERT use contextual word representations (same word can have many representations).

Lecture 11: Convolutional Networks for NLP

(Discrete) Convolution is defined as:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m]$$

In NLP, filters (patches) capture some information about the text e.g. one (learned) filter might be responsible for detecting sentinent of a text, whereas the other can capture some specific topic of a text.

> In NLP, the *number of channels* refers to the word embedding size (dimension)/feature

- CNNs do very well on **text classification** tasks. Their advantage is that they parallelize well on GPUs.
- RNNs are not ideal for classification and are way slower than CNNs. They work well for **sequence tagging, language models** (especially with attention).

Regularization

A good practice is to use **Dropout** - masking vector r of Bernoulli random variables with probability p (a hyperparameter) of being 1. Then, during training, we delete features:

$$y = \sigma(W^{(S)}(r \circ z) + b)$$

where σ is *softmax*. Dropout prevents co-adaptation (overfitting to seeing specific feature constellations). At test time we need to scale the vector by probability p:

$$\hat{W}^{(S)} = pW^{(S)}$$

Residual block

Residual block is an (neural) architectural component, where we take an input x and copy it into two copies - one of them is passed through a series of transformations: $conv \rightarrow relu \rightarrow conv$ (let's call it $F(x)$) and then this output is added directly to the other copy x.

> Note: In order to sum x and the transformed x, you need to pad the first copy of x before passing through $conv$

The output from the residual block is thus:

$$relu(F(x) + x)$$

Highway block

Highway block is similar to the residual block but x and $F(x)$ are weighted before summing them, so the output from the block is:

$$relu\big(F(x) \circ T(x) + x \circ C(x)\big)$$

where T, C are some learned vectors (weights) and $\circ$ is a Hadamard product (element-wise vector product).

> Note: Highway and residual blocks are necessary to train (very) deep neural networks

Batch Normalization

Batch Normalization (BatchNorm) means standardising the output from the convolution layer (given a batch of data). It makes models less sensitive to parameter initialization (outputs are automatically scaled) and makes tuning of learning rates simpler.

Convolution Block

A Basic ConvBlock is essentially a series of three transformations: $Conv \rightarrow BatchNorm \rightarrow ReLU$

Lecture 12: Subword Models

Linguistics

Subword models are based on the idea that we can build models on character level (or n-grams of characters). The rationale being that *morphemes* are smallest semantic units, e.g. in the word *unfortunately* - un, fortun(e), ate, ly - each have their semantic meaning. In some langauges (e.g. german) words can be very long making up a huge vocabulary space.

Human language writing systems can be:

- Phonemic, where each letter is a sound (maybe digraphs, e.g. Polish "sz" pronounced as "sh")
- Fossilized phonemic, e.g. English
- Syllabic/moraic - e.g. Korean, where each letter is a syllable
- Ideographic (syllabic) - e.g. Chinese, where characters have a meaning
- Combination of the above - e.g. Japanese, which is partally moraic and partially ideographic

Character-based LSTM

Some models build character-level embeddings (each character has its own embedding) and then train a model based on that (e.g. Character-Aware Neural Language Models).

The architecture is:

$$Character\ embedding \rightarrow CNN \rightarrow Highway\ network \rightarrow LSTM \rightarrow prediction$$

The empirical observation is that the *Highway network* allows the model to learn semantic similarity. e.g. without that layer, the model would say that "richard" is similar to "rich" (character-level similarity) but with a highway network it gives some other name as the most similar vector.

Character-level models can also recognize that the word "loooook" is similar to "look", whereas word-level models output "out of vocabulary" errors.

FastText embeddings

In FastText the authors represent words as char n-grams augmented with boundary symbols and as a whole word, e.g. *where = <wh#, whe, her, ere, re#>,* (where "#" is a boundary symbol). Then the word embedding is represented as a sum of representations of those "sub words".

Lecture 13: Contextual Word Embeddings

Reflections on word representations

One representation for each word:

- Word2vec
- GloVe
- FastText

The problems with those representations are:

- A word has the same representation regardless of the context in which it occurs
- Words have different **aspects**, including semantics (i.e. meaning), syntactic behaviour (e.g. arrive/arrival) and conotations.

> Note: In a vanilla NLM (Neural Language Model) the hidden states already captured information about word's context

Pre-ELMO and ELMO

TagLM (Peters et. al, 2017) uses simultaneously two word embeddings - it uses one context-independent word embedding (like Word2Vec), but also simultaneously trains another one, context-dependent embedding using standard recurrent language model (biLSTMs), where hidden states

convey the context information. Then it combines the two representations to perform tagging (Named Entity Recognition).

ELMo (Peters et. al 2018) (Embeddings from Language Models) - improvement over TagLM

ELMo, when added to some baseline model beat SOTA in many other tasks (SQuAD, SNLI, SRL, Coref, NER, SST-5)

ULMfit and onward

The idea behind ULMfit (Universal Language Model Fine-tuning) was to train a Language Model (unsupervised) on a big general domain corups (use biLM), then fine-tune on target (domain) task data and finally, introduce an objective specific to the task (here - classification objective), therefore turning LM into a text classifier. So what we end up with is a single NN with two, separate, top level layers with different objectives, performing different tasks.

BERT is a scaled (training time: ~320-560 GPU days) version of ULMfit (1 GPU day). GPT-2 (Generative Pre-trained Transformer, OpenAI) is a scaled version of BERT (~2048 TPU days)

Transformer architectures

The motivation for transformers is that RNNs are inherently sequential (not possible to parallelize, i.e. scale).

Since **attention** gives us access to any state - the idea was to get rid of the RNN altogether. This was presented in Attention is all you need (Aswani et. al 2017). The paper presents a non-recurrent sequence-to-sequence encoder-decoder model and the task is a machine translation with parallel corpus.

Multi-head attention

The problem with simple self-attention is that it gives only one way for words to interact with one-another

Transformer block

Transformer block consists of two "sublayers":

- Multihead attention
- 2-layer feed-forward neural net (with ReLU)

The input x goes first to the multihead attention layer and the output from that layer is then added (and normalized) with the initial input x (Residual connection) to give y. Then, y is fed into FFNN and added again to y (another residual connection) and normalized.

> Note: The normalization used is a **Layer Normalization** (LayerNorm). It changes the input features to have a mean of 0 and a variance of 1 (and adds two parameters)

Transformer

Transformer is an encoder that takes an input embedding, combines it with the positional encoding and then passes through N transformer blocks (in the original paper $N = 6$) stacked vertically.

BERT

BERT (Devlin, 2018) (Bidirectional Encoder Representations from Transformers). It is a Transformer encoder with (multi-head) self-attention, which allows for no locality bias (long-distance context has "equal opportunity").

BERT tries to address a problem of a language model being unidirectional (predict next word given history). It instead trains a model to fill in the gaps in sentences (some words are randomly removed).

There is another objective which learns relationships between sentences. It does it by learning a binary classification task - for given sentences A, B, classify whether B is the actual sentence that proceeds A, or a random sentence (*True* if B follows A and *False* otherwise).

BERT sentence pair encoding

For a given input sequence, BERT encodes each word by summing up three terms:

1. Token embeddings (usual word embedding for each word)
2. Segment embeddings (represents whether a word comes from a sequence A or B, assuming there is a separator in the input)
3. Position embeddings (where in the sentence a particular word/token is)

BERT model fine tuning

BERT can be fine-tuned (after pre-training) for many tasks (like ULMfit), e.g.

- MNLI (Multi-Genre Natural Language Inference)
- NER (Named Entity Recognition)
- SQuAD (Stanford Question Answering Dataset)

by substituting the top (prediction) layer by a layer specific to that task.

> Remark: Natural language inference is the task of determining whether a “hypothesis” is true (entailment), false (contradiction), or undetermined (neutral) given a “premise”.

Lecture 14: Transformers and Self-Attention

Self-Attention

Self-attention allows a word (embedding) to "re-express" itself (i.e. change its embedding) in terms of a weighted combination of its neighbourhood. So self-attention takes an input word embedding (and

embeddings of its neighbourhood) and combines them into new word embedding for that particular word (now - aware of its context).

> Remark: Self-attention is permutation-invariant (i.e. if we permute the words in an input, they're gonna have the same representation after self-attention). That's why, in order to maintain order, we need the position representations.

> Remark 2: When computing "self-attention" representation for a given word w we first make a linear transformation of that word and call it a *query*, and then we compute linear transformations of each of its neighbours and call them *keys*. Those transformations are made in order to project vectors to such a space where dot product is a good proxy for *similarity*.

(Encoder) attention output:

$$A(Q, K, V) = softmax\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V$$

where Q - query (transformed input word), K - key (transformed neighbouring word), $\sqrt{d_k}$ is a scaling factor and V is a value.

> Remark: Attention is just a weighted average of values

Complexity comparison

- Self-Attention: $O(L^2 d)$
- RNN(LSTM): $O(Ld^2)$
- Convolution: $O(Ld^2 w)$

L is a length of a sequence and d is the dimension of the representation (embedding) and w is a kernel width.

Lecture 15: Natural Language Generation

Decoding algorithms

> Reminder: Decoder works by predicting the next word given a current input, which is initialized as $< start >$ token. It can, at each step, assume the current input word is an $argmax$ over all most likely words produced in the previous timestep (greedy decoding), or it can consider k of most probable partial sequences (hypotheses) at each timestep (beam search decoding).

We can also use *sampling* to generate sentences from a decoder, i.e. instead of picking words from the decoder, we'd sample them based on the probability distirbution outputted by a decoder. We can also use *top-n sampling*, i.e. restrict sampling only to the most n probable words.

> Note: High n gives more diverse/risky output and low n gives very generic/safe output (but it might be too generic, e.g. in a dialogue system it would be outputting a sentence that could be used almost always, in any situation)

Softmax temperature
Softmax temperature is another way to control the diversity of the sentence generation output. The idea is to add a parameter τ (temperature) to the softmax distribution, which makes the distribution more uniform (high temperature - it *melts* the distribution) or less uniform (low temperature).

NLG tasks and neural approaches to them

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is the main text summarization evaluation metric. *ROUGE* is based on recall, whereas *BLUE* metric was based on precision.

Dialogue

Dialogue has many problems, e.g.

Irrelevant response problem

Seq2Seq often generates response that's unrelated to user's utterance. A possible solution would be to optimize for *Maximum Mutual Information (MMI)* between input S and response T:

$$MMI(S,T) = logp(T|S) - logp(T)$$

MMI discourages high $p(T)$, i.e. responses that have high probability themselves (are too generic).

Repetition problem

Solution - directly block repeating n-grams during beam search (we keep already used n-grams in memory). A more complex solution would be to train a *coverage mechanism* - objective that prevents attention from attending to the same words multiple times.

Lecture 16: Coreference Resolution

Coreference resolution task is about identifying all **mentions** that refer to the same real word entity.

> e.g. in *"Barack Obama nominated Hillary Rodham Clinton as his secretary of state on Monday"*, "Barack Obama" and "his" are coreferent to each other - they refer to the same entity.

> Remark: No NLP system can currently deal with **split antecedence**, i.e. if there is a word "they" in text - figuring out what entities it refers to.

Mention detection

There are three types of mentions:

1. Pronouns (solution - use part-of-speech tagger)
2. Named entities (solution - use a NER system, e.g. *hw3*)
3. Noun phrases (solution - use a parser, e.g. constituency parser)

Anaphora vs. Coreference

A related linguistic concept is **anaphora** when a term (anaphor) refers to another term (antecedent).

> e.g. Barack Obama said he would sign the bill

"Barack Obama" is an antecedent and "he" is an anaphor. The difference between the anaphora and coreference is that, in coreference, all mentions refer to the same "real" object. But an anaphor does not need to have a reference.

> e.g. No dancer twisted her knee

"No dancer" does not have any reference, but "her" is an anaphor for "no dancer".

Coreference models

There are four types of coreference models:

- Rule-based
- Mention pair
- Mention ranking
- Clustering

Knowledge-based Pronomial Coreference

Sometimes coreference can only be resolved by having some intristic "knowledge":

> e.g. *"She poured water from the pitcher into the cup until it was full"* vs. *"She poured water from the pitcher into the cup until it was empty"*. In both cases "it" refers to a different subject.

Such sentences are sometimes referred to as **Winograd Schema** (recently proposed as an alternative to the Turing Test, i.e. solving coreference is essentially solving AI).

Coreference evaluation

There are a couple of popular metrics to evaluate coreference resolution: MUC, CEAF, LEA, B-CUBED, BLANC

Lecture 17: Multitask Learning

> Multi-task learning (MTL) is a subfield of machine learning in which multiple learning tasks are solved at the same time, while exploiting commonalities and differences across tasks.

Curriculum learning - training a model with simple examples first, with increasing difficulty

Lecture 18: Constituency Parsing, TreeRNNs

Constituency parsing is another type of parsing (next to *Dependency parsing*). Linguists call it sometimes *Phrase structure grammars* and Computer Scientists call it *Context-free grammars*.

Task: Map phrases into vector space

> e.g. "The country of my brith"

Solution: Principle of compositionality - the meaning of a sentence is determined by 1. the meanings of its words, 2. the rules that combine them.

Using the principle of compositionality, for an example above, we'd create a vector for "The country" and a vector of "of my birth" by combining "of" with "my birth" (we recursively parse the sentence tree and then combine vectors into one final vector for the entire sentence).

Compositionality Through Recursive Matrix-Vector Spaces

The word *very* does not have any meaning on its own. It acts as an "operator" on another word (e.g. "very good"). The idea is to model the operators with matrices (multiplying the matrix representation of the word "very" with a vector "good" gives a new word vector "very good").

In the paper, each word is associated with both a vector and a matrix.

TreeRNNs

The idea behind Tree RNNs is to parse a given sentence into a tree (the words of the input sentence are leaf nodes) bottom-up by passing neighbouring nodes through some NN which defines whether the children form a good consitutent that can be a part of a parse tree (e.g. in "on the mat", "on" and "the mat" are good constituents, but "on the" and "mat" aren't). The NN outputs two things: a scalar (score) describing whether the children nodes form a good constituent and a vector representation for that constituent. This vector representation is then used to build nodes higher in the tree until a root node is build (which defines an entire input sentence).

> Example: In a *sentiment analysis* task, the approach was to parse a review (here: movie review) into a tree, and then, going bottom-up, compute a sentiment score for each subtree.

Lecture 19: Bias in AI

Human Reporting Bias

> The frequency with which people write about actions, outcomes or properties is not a reflection of real-world frequencies or the degree to which a property is characteristic of a class of individuals.

> e.g. In a study, in a large corpus of text, the word "murdered" appears 10x times more than "blinked". This is becasue humanns tend not to report "typical" things.

Biases

Bias in data

Reporting bias - what people share is not a reflections of real-world frequencies
Selection bias - selection does not reflect a random sample (e.g. distribution of Amazon Mechanical Turk workers around the world),
Out-group homogeneity bias - perception of out-group members as more similar to one another than are in-group members, e.g. "they are alike; we are diverse", or from the perspective of a European - "all Asians look the same",
Biased Labels - e.g. when given a photo of a culture-specific custom, an annotator from another culture can fail to annotate it currently (e.g. they might not be able to recognize a groom/bride/wedding from a photo)

Bias in interpretation

Confirmation bias - tendency to search for, interpret, favour, recall information in a way that confirms preexisting beliefs,
Overgeneralization - coming to a conclusion based on information that is too general and/or not specific enough,
Correlation fallacy - confusing correlation with causation,
Automation bias - propensity for humans to favour suggestions from automated decision-making systems over contradictory information without automation

Fairness

Equality of Opportunity - equal recall for each (sub)class
Predictive Parity - equal precision for each subgroup (class)

Types of Bias

Low Subgroup Performance - The model performs worse on one subclass than overall. To calculate it we can use subgroup (subclass) AUC.

> Remark: Interpretation of AUC - given two randomly chosen examples (one in-class and the other not in-class), AUC is the probability that the model will give the in-class example the higher score.

Subgroup shift - The model systematically gives scores to one subgroup higher/lower. We can use BPSN/BNSP (Background Positive Subgroup Negative/Background Negatvie Subgroup Positive) AUC.

Lecture 20: Future of NLP + Deep Learning

Unsupervised learning

Pre-training (Machine Translation)
The idea of pre-training in machine translation is to pre-train an encoder and a decoder as language models in their target (in machine translation task) languages. This can be especially useful if we don't have a lot of translated (labeled) data for one of the languages (but there is monolingual corpus of text for that language). This allows the encoder/decoder to already have some information before actually starting to feed in labeled training data for the machine translation.

Back-translation (Machine translation)
Train the model on its own translations, i.e. given a sentence s (e.g. english), compute its translation (e.g. english to french) and use that translation to train the opposite model (e.g. french to english) expecting to get back s as a result.

Unsupervised word translation

Cross-lingual word embeddings - the idea is to try to match words within pairs of languages based on their embedding. The underlying assumption is that the embedding space can have a similar structure across languages. We can then learn an (orthogonal) matrix W (orthogonal to ensure that it is constrained to rotation) such that $WX = Y$ (where X is an embedding space in language x and Y is an embedding space in language y)

We can learn W by adversarial training. We train a discriminator which predicts whether an embedding is from Y or whether it comes from a transformed X, i.e. WX. We then train W to confuse the discriminator.

Training huge models

Comparison of the number of parameters across the models:

- Medium-sized LSTM - 10M
- ELMo - 90M
- GPT - 110M
- BERT-Large - 320M
- GPT-2 - 1.5B

Parallelism
There are two types of parallelism:

- Model parallelism - we divide model into parts and distribute them across different machines (see *Mesh-Tensorflow* framework)
- Data parallelism - we divide data into parts and distribute across different machines (all machiens have the same copy of the model)

Zero-shot learning

Zero-shot learning is about model performing a task without being trained for that task.

GPT-2 was trained as a language model but it can perform reading comprehension, summarization, question answering and translation - all zero-shot.

Lecture 21: Low Resource Machine Translation

> Definition: A language pair can be considered *low resource* when the number of of parallel sentences is in the order of 10k or less

Model Regularization
Three popular ways to regularize Language models:

- Punish L2 norm of hidden units
- Dropout
- Label smoothing (our label vector is not one-hot encoded but has some small ϵ for other classes/words)

Semi-supervised learning

> Note: Lecture introduced some concepts already covered in previous lecture

In a semi-supervised setting we have both: parallel data (original + translation) dataset D and a monolingual dataset M (for one of the languages - usually for the high-resource one). The idea is to use M to model $p(x)$ (x is a sentence in the source language).

Solution: Use DAE - Denoising Autoencoding. We can either pretrain the encoder based on monolingual data or add a DAE loss to the supervised cross-entropy term:

$$L^{DAE}(\theta) = -logp(x|x+n)$$

where n is noise (dropping/swapping words). In this setting both encoder and the decoder use the same language x. In this way we train the encoder to understand the language x. After pre-training we replace the decoder with the original decoder trained on parallel data to produce sentences in a target language y.

> Remark: DAE makes sure decoder outputs fluently in the desired language

Self-training
In self training we take two copies of input x (coming from a monolingual dataset M):

- the first copy of x is passed through encoder + decoder to get $\bar{y}$
- the second copy gets added with noise (dropping/swapping) and then passed through the same encoder and decoder to get y

We then train the model by calculating a loss between $\bar{y}$ and y (e.g. cross entropy).

Multi-task/multi-modal

The approach here is to use **Multilingual training**, i.e. share the same encoder and the same decoder with all the language pairs. During training, prepend a language identifier (token) to the source sentence (to inform decoder of desired lanauge). This makes the encoder produce shared representations.

Conclusions

In a semi-supervised, low-resource machine translation, the best results are obtained by using **back-translation + multi-lingual training + DAE**

Lecture 21: BERT and Other Pre-trained Language Models

History of contextual representations

- *Semi-Supervised Sequence Learning (Google, 2015)* - the idea was to pretrain the entire model as a language model and then fine tune it on the actual task (here: sentiment classification)
- *ELMo: Deep Contextual Word Embeddings (AI2 & University of Washington, 2017)* - the idea was to train separate Left-to-right and Right-to-left language models (biLSTMs) and use them as pre-trained embeddings to an existing model architecture (instead of using e.g. GloVe/Word2vec embeddings)
- *GPT1: Improving Language Understanding by Generative Pre-Training (OpenAI, 2018)* - the idea was to train a deep (12 layer) transformer language model and then fine-tune on a target (classification) task

Advantages of Transformers vs. LSTM

1. Self-attention = no locality bias. Long distance context has "equal opportunity"
2. Single multiplication per layer = efficiency on TPU - effective batch size is number of words, not sequences

Problem with previous methods (rationale for BERT)

Problem: Previous language models (ELMo, GPT) only use lef context or right context (or concatenation of the two), but language understanding is bidirectional.

Why were LMs unidirectional (prior to BERT)?

- Directionality is needed to generate a well-formed probability distribution. Solution: we don't care about this in our setting (we only do when building an actual LM).
- Words can "see themselves" in a bidirectonal encoder. Solution: use masking.

Distillation

Distillation is a technique used to "compress" a model for inference. For example, BERT is not deployed "as is" to production. Instead, *distillation* technique is used to build a smaller (~50x) model (based on the original, huge SOTA model) that uses "teacher" model during training. The "student" model trains based on the output produced by the teacher, for a specific task (the original model might work well for more than one task).

> Note: *Student* objective is usually MSE or Cross Entropy

Distilaltion works because the teacher model (being a massive language model) has millions of learned latent features which are not useful for many NLP tasks - only a subset of them is useful for a given task. Distillation allows the (student) model to only focus on those features.