

Undergraduate Research Opportunities

Programme in Science

Basics and Applications on Natural Language Processing

Dick Jessen William (A0200677H)

AY21/22 Semester 2

Supervisor: Dr. Zhang Lei

Department of Mathematics

National University of Singapore

Contents

1	Introduction	4
2	Basics and Notations	4
2.1	Linguistics	4
2.2	Probability	5
2.3	Linear Algebra and Calculus	6
3	Rudimentary Techniques	6
3.1	First Model: n-grams	7
3.2	Neural Language Models	10
4	High Dimension for NLP	12
4.1	Shortlist	13
4.2	Hierarchical Softmax	14
4.3	Importance Sampling	15
4.4	Combining Models with n-grams	17
5	Recurrent Neural Network (RNN)	18
5.1	Unfolding the Network	19
5.2	RNN	20
5.3	Updating RNN	21
5.4	Encoders and Decoders	23
6	Neural Machine Translation	24
6.1	Attention Mechanism	25
6.2	Transformer Model	26
7	Generative Pre-trained Transformer 3	30
7.1	Motivation and Prior Work	30
7.2	Model Specification and Training	31

7.3	Model Usage and Performance	34
7.4	Improvements	35
8	Codex: Writing Code to Write Code	36
8.1	Introduction and Task Evaluation	36
8.2	The pass@k Metric	38
8.3	Model Training	39
8.4	Model Results and Improvements	40
9	Applications: Solving and Generating College Math Questions	42
9.1	Introduction and Dataset	42
9.2	Methodology	43
9.3	Model Result	46
A	A Small Note on Decode Output	48
B	How Machine Grade Essays: BLEU Score	50
C	Sampling with Temperature	51
D	BERT, Alternative to GPT	52
E	Text to Code Samples	54
F	Algorithms	56
G	Materials	57

1 Introduction

A Natural Language Processing (NLP) problem is a branch of Artificial Intelligence that focuses on how computers can interpret human language. The field of NLP is present in many tasks, such as auto-completion in emails and translation of texts via Google Translate. With the new inventions and discoveries in the field recently, there are now things that can be done consistently that are impossible just a few years ago.

In this UROPS, we will start by examining some basic strategies to tackle this problem using simple techniques, and then explore the dimensionality problem of NLP tasks, which will be present through our exploration. Then, we will cover the usage of Recurrent Neural Networks (RNN) to solve NLP tasks by exploiting their recurrent nature. After that, we will focus on a specific task, Machine Translation (translating between two languages), and use this example to introduce two important models: the Attention Mechanism and the Transformer Model. Also, we will explore a revolutionary model GPT-3, that boasts revolutionary feats unlike any other. We finally end the report by examining the applications of GPT-3 to build a model to translate text to code, and later use it to solve and generate college mathematics problems.

2 Basics and Notations

First, we talk about some notations and prerequisites.

2.1 Linguistics

We define some keywords related to linguistics. First, define a dictionary D as the list of possible words. For example, the simplest dictionary is just the set of all English words, with members like "I", "Canada" and many more.

In the context of NLP, we are mainly concerned about tokens. We define a

token w similar to the words in linguistics, but now it can also extend to other languages, such as Python codes, where the tokens will be ":", "for", and so on. Unless otherwise stated, tokens are case insensitive. If the meaning is obvious, we can interchange tokens and words in sentences. Then, a sentence is just a combination of tokens. For example, in the sentence $x = \text{I am human}$, we can let $x_1 = \text{I}$, $x_2 = \text{am}$ and $x_3 = \text{human}$ and write $x = x_1, x_2, x_3$. We define naturally the length of a sentence as the number of tokens it has. Finally, we can denote $|D|$ as the size of the dictionary D .

We also mention a word dataset a lot in the discussion. A dataset means a list of sentences. For example, a possible dataset might be an article or a novel. A dataset also works as a multi-set of tokens. We can access the frequency of a word w in a dataset V by $C_V(w)$. We only write it as $C(w)$ if the context is clear.

2.2 Probability

Then, we touch on some probability basics. First, define a sample space S as the set of possible outcomes of some process. For example, S may be the result of a coin flip. Then, for each element $s \in S$, we can assign a number $p(s) \in (0, 1)$. We require that $\sum_{s \in S} p(s) = 1$, which means every process's outcome must be on the sample space. Then, we define an event E as a subset of S . We now define the probability of an event E as $P(E) = \sum_{e \in E} p(e)$. We can easily see that $0 \leq P(E) \leq 1$.

We also want to consider conditional probability. For events A, B , we denote $P(A|B)$ as the likelihood of event A happening given that event B happens. The useful theorem here is the Bayes rule, which states that $P(A|B) = \frac{P(A \cap B)}{P(B)}$.

In the context of NLP, we define $P(w_i)$ given a dictionary D as how likely a word w_i appears in the usage of dictionary D . For example, in English dictionary, $P(\text{I})$ might be much higher than $P(\text{Combinatorics})$. We can also do this for sentences. We write this using conditional probability. For example, $P(\text{rice}|\text{I want})$ denotes the probability of the word "rice" appearing after "I want". Most of the time, we

do not state what D is in the context, because it does not matter. We also want to write how likely a sentence C will be continued by a word i by $P(i|C)$.

Finally, we define what a language model is. It is a function of probabilities for a sequence of tokens. In other words, given a sentence $w = w_1, w_2, \dots, w_n$, a language model M will map w into $P(w)$, the probability of w appearing in a text.

2.3 Linear Algebra and Calculus

We will use a fair share of Linear Algebra and Calculus in this report. Here, some of the more important terms are defined. For a vector v , write v_i as the i -th entry of the vector v .

Firstly, a $n \times k$ matrix is called sparse if the number of zero entries is $O(\sqrt{nk})$. This bound is in a sense not important, as the ideas that work on sparse matrices will work in normal matrices, although performance is compromised. A sparse vector is defined similarly.

Next, a one-hot vector is a vector that checks a membership of an element. It has one 1 entry, and the rest is zero. For example, we might write a one-hot encoding for "Hello" as $[0, 0, \dots, 1, \dots, 0]$, where the dimension of the vector is the size of the dictionary.

For a multivariable function f and a vector v , we denote the $\nabla_v f = (\frac{\delta f}{\delta v_1} \frac{\delta f}{\delta v_2} \dots \frac{\delta f}{\delta v_n})$ as the partial derivative of f . Also, recall the chain rule for vector notation, which states that if $y = g(x)$ and $z = f(y)$, then $\nabla_x z = (\frac{\partial y}{\partial x})^T \nabla_y z$.

Define also $1_{x,y}$ as a function that returns 1 if $x = y$, and 0 otherwise. Finally, \log here means base-2 logarithm.

3 Rudimentary Techniques

First, we will cover the methods used to model languages without deep learning. This will help us to appreciate later the power of newer techniques. Most explana-

tions are taken from section 12.4 of the book by Goodfellow [9].

3.1 First Model: n-grams

The simplest one is the n-gram model. Firstly, a n-gram is just a n-tuple of tokens. In the n-gram model, the k-th token is determined by looking $n - 1$ words behind it. The model does this by enumerating all k-tuple of tokens and count their frequencies. Suppose that $P_k(w)$ is the probability of a sentence w appear in the dictionary, given that w has length k . Then, to predict a general sentence, we use the chain rule to get the probability. Notice that for words x_1, x_2, \dots, x_t , we have

$$P(x_1, x_2, \dots, x_t) = P(x_1, x_2, \dots, x_{n-1}) \prod_{i=n}^t P(x_i | x_{i-n+1}, \dots, x_{i-1}).$$

We also note that by chain rule,

$$P(x_t | x_{t-n+1}, \dots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \dots, x_t)}{P_{n-1}(x_{t-n+1}, \dots, x_{t-1})}.$$

An interesting tangent is we often use both n and $n - 1$ gram to ease the model training, as we now simply look for values of each conditional probability on the right-hand side (instead of manually checking the sentences). The first $n - 1$ words can then be predicted by the $n - 1$ gram model. For a concrete example, suppose we want to find how likely a sentence "I like fresh orange" appears using a 3-gram model and 2-gram model. Using the formula, the model will assign

$$\begin{aligned} P(\text{I like fresh orange}) &= P_3(\text{I like fresh}) \times P(\text{orange} | \text{I like fresh}) \\ &= P_3(\text{I like fresh}) \times \frac{P_3(\text{like fresh orange})}{P_2(\text{like fresh})}. \end{aligned}$$

However, this simple model has issues. Firstly, it cannot estimate n-grams which it has not encountered yet. If this happens, either the denominator becomes 0, which does not make sense, or the numerator becomes 0, which causes the overall

probability to become 0.

We can counter this by smoothing. One basic example is called Laplace Smoothing, where we add a small number to every possible word. Mathematically, if $P(w_i) = \frac{|w_i|}{N}$, then $P_{laplace}(w_i) = \frac{|w_i|+1}{N+|V|}$, where N is the number of words observed and V is our dictionary.

For example, suppose in a simple dictionary (that we made of), there are only 3 words: Send, Me, and Money. Suppose we have this table of frequencies from a toy dataset (all other possible combinations do not appear).

Word	Frequency
Send Me	12
Me Money	10
Send Money	8
Me Send	4
Money Send	2
Send	30
Me	40
Money	25

Now, we want to find $P(\text{Send Money Me})$, using both 2-gram and 1-gram model. We will find a problem, because we need to calculate $P_2(\text{Send Money}) \times \frac{P_2(\text{Money Me})}{P_1(\text{Me})}$, and the numerator is 0, as there are no "Money Me" in our dataset. Then, we will predict that Send Money Me never appears, which is not really convincing because it might appear in the future. Using smoothing, we can easily calculate the probability as $\frac{12+1}{36+9} \times \frac{(0+1)/(36+9)}{(40+1)/(95+3)} = 0.0153$. according to the model.

Another idea is using the back-off model. The main idea for this model is if the n -gram frequency is too low, use the $n - 1$ gram model instead, and so on. We will set a constant C_0 as the threshold frequency on whether we will consult the smaller n -gram. Also, because the sample space of all $n - 1$ grams is no longer added to 1, we need to rescale the probability by a constant a_n to ensure that the sum of

probabilities stays at 1. In other words,

$$P(x_{i-n+1}, \dots, x_{i-1}, x_i) = \begin{cases} P_n(x_{i-n+1}, \dots, x_i) & \text{if } C(x_{i-n+1}, \dots, x_i) > C_0, \\ a_n \times P(x_{i-n+2}, \dots, x_{i-1}, x_i) & \text{otherwise.} \end{cases}$$

However, the n-gram model has problems. Firstly, the set of possible n-grams is $|V|^n$, which is very large because of the size of our dictionary V . For instance, in English, $|V| \approx 500000$. Hence, even for a 3-gram model, the space requirement is very big. However, most of these sentences will never appear (as three random English words jotted together rarely makes a good phrase), and this makes the probability of almost all n-grams very close to zero. Also, if we represent every word with a one-hot vector, then the distance between every different word is identical, which is not desirable as similar words should be nearer (help vs helped ideally should be seen as more similar than hello vs chocolate by a good model). Hence, we need a model that can do this.

One way is to introduce class-based language models. We can see this as partitioning our dictionary set to subsets C_1, C_2, \dots, C_i , with each C_i becoming one class. Suppose that our tokens in the dictionary V are divided to $|C|$ classes by a function $f : V \rightarrow C, f(x_i) = c_{g(i)}$ for some g that maps $\{1, 2, \dots, |V|\}$ to $\{1, 2, \dots, |C|\}$. A n-gram model is also a class model if

$$P(x_i | x_{i-n+1}, \dots, x_{i-1}) = P(x_i | c_{g(i)}) P(c_{g(i)} | c_{g(i-n+1)}, \dots, c_{g(i-1)}).$$

Notice that there are only $|C|^n - 1 + |V| - |C|$ independent parameters needed, the $|C|^n - 1$ is for the n-gram of a "new language" of the classes (this is for the right part, and all must sum to 1 so we decrease one parameter), and $|V| - |C|$ on the form $P(w_i | c_{g(i)})$ (this is for the left part, there are $|C|$ equations each having a constrain of sum equal to 1, so need to decrease $|C|$ parameters). Because $|C| < |V|$ most of the time, this greatly saves space. It is also possible to combine this with the backoff

model. However, since we squash the words into a much smaller space, information is lost in the transition.

3.2 Neural Language Models

Neural Language Models are designed to solve the curse of dimensionality by modeling natural language sequence by a distributed representation of words. This will enable us to recognize similar words (like the one in class-based models) but now not only we also know that they are similar, but also they are not identical. We will use the notion of word embeddings. An embedding translates a very large vector into a lower-dimensional space, while still preserving semantic relationships (similar words mapped to similar vectors, according to some similarity measure, like cosine similarity).

For example, suppose we have a simple dictionary containing the tokens {Man, Dog, House, Cat, Large, Sleep, Teacher}. We can trivially make a one-hot encoding for each word, mapping it into a seven-dimensional vector. However, we can create a custom embedding (by hand) such as follows.

Word	Living	Human	Verb	Adjective
Man	1	1	0	0.5
Dog	0.9	0.2	0	0.1
House	0.1	0	0	0
Cat	0.8	0	0	0.1
Large	0	0	0	1
Sleep	0	0	1	0
Teacher	0.7	1	0.1	0

Many other good embedding exists for this toy example, but this one is chosen to make a point on why we should do embeddings in NLP.

There are two important features that we will look at. Firstly, the dimension is only four, instead of seven. Secondly, we can see roughly which words are similar to each other (Man - Teacher), (Dog - Cat), etc. However, for large dictionaries like

English, it is impossible to fill in numbers manually. We introduce a method called TF-IDF as an illustration of how to create a simple embedding. Note that other methods, such as Google's Word2Vec also exist and can be used.

TD-IDF stands for Term Frequency Inverse Document Frequency. The main idea is that we want to find out how relevant a word in some text (for example, a news article). For a set of datasets D , a dataset $d \in D$ and a word t , define $tf(t, d)$ as the fraction of the words in d that is equal to t and $idf(t, D) = \log_{10} |D|/n_t$, where n_t denotes how many datasets in D contains t . Finally,

$$tf_idf(t, d, D) = tf(t, d) \times idf(t, D).$$

For example, suppose I have four datasets, all contained in a set D .

1. d_1 : Have three cats.
2. d_2 : Have two dogs.
3. d_3 : Cats like dogs.
4. d_4 : Dogs eat two hot dogs.

First, compute tf for each word-dataset pair. For example, the tf for d_4 and "dogs" is $\frac{2}{5}$ because "dogs" appear twice.

	Have	Cats	Dogs	Three	Two	Like	Eat	Hot
1	1/3	1/3	0	1/3	0	0	0	0
2	1/3	0	1/3	0	1/3	0	0	0
3	0	1/3	1/3	0	0	1/3	0	0
4	0	0	2/5	0	1/5	0	1/5	1/5

Then, compute IDF. For example, the IDF for "Have" is $\log_{10} 4/2 = 0.301$ because 2 of the 4 sentences contains "Have".

Finally, combine to get TF-IDF. The larger the number means the word is more important in the particular document.

	Have	Cats	Dogs	Three	Two	Like	Eat	Hot
IDF	0.301	0.301	0.201	0.602	0.301	0.201	0.201	0.201

	Have	Cats	Dogs	Three	Two	Like	Eat	Hot
1	0.100	0.100	0	0.200	0	0	0	0
2	0.100	0	0.067	0	0.067	0	0	0
3	0	0.100	0.067	0	0	0.067	0	0
4	0	0	0.080	0	0.040	0	0.040	0.040

Now, we can use this for the embedding. For example, "Have" is represented as $[0.1, 0.1, 0, 0]$.

4 High Dimension for NLP

In most application of NLP, such as machine translation (Google Translate), speech completions, and et cetera, we want our models to produce words as units in the output, rather than characters. Consider this 'naive' way of determining which word to be chosen as output: given a hidden representation, (we can assume it as a vector), do an affine transformation (equivalent to the map $v \rightarrow Av + b$ for suitable sized matrices A, b) to an output space (will be sized $1 \times |V|$, with V as our dictionary), then apply softmax to the vector. Mathematically, given a vector $v = (v_1, v_2, \dots, v_{|V|})$, define $\text{softmax}(v)$ as

$$\text{softmax}(v)_i = \frac{e^{v_i}}{\sum_{j=1}^{|V|} e^{v_j}}.$$

We do the softmax operator to represent the probability of each word being chosen (as in, the probability of a word with index i is $\text{softmax}(v)_i$). While this works, this has severe complexity issues. While training, we need to do multiplication of a large matrix with a vector (to compute the weights in the neural network, using the formula $o = Wi + b$, where i is the previous layer, W is the weight matrix, b is the bias and o is the output vector). The complexity is $O(|V||h|)$, where h

is the dimension of the hidden representation. Because $h \approx 10^3$ and $V \approx 10^5$ in most applications, this calculation dominates most of the time in neural model training. Worse, it even affects testing. Because we want to know which word to output, we need to consider every possible word in the output, so we must do the full multiplication (even when we want to find the probability of a word appearing, we still need every vector member because of softmax). Hence, we will list some common workarounds for this problem.

4.1 Shortlist

The early neural models dealt with the high cost of softmax-ing a large vector by limiting $|V|$. Then, building from this approach, a new method is found. We consider a subset L of V which is called a shortlist that contains the most frequent words from the language V and the tail $T = V \setminus L$ for the rest of the words in the dictionary. The idea is we only use a neural network to check over L and use an n-gram model on T . Now to combine these two predictions, the neural network (that we use in L) has to output the probability that the word appears from T . We can do this by appending the output vector an extra sigmoid unit ($\sigma(x) = \frac{1}{1+e^{-x}}$) to predict X , the probability that the next word is in the shortlist. Then, we can combine both predictions to predict,

$$P(y = i|C) = \begin{cases} P(y = i|C, i \in L)X & \text{if } i \text{ is in the short list} \\ P(y = i|C, i \in T)(1 - X) & \text{otherwise.} \end{cases}$$

Note that the first case is computed using the neural model, and the second one is calculated using the n-gram model. This is substantially faster than the naive model because firstly, the output size on the network is significantly smaller. This drastically lowers the computational complexity. The disadvantage of this model is that the generalization power of the NLM is only limited to the words in L . Other methods are derived to counter this.

4.2 Hierarchical Softmax

One older approach to reducing computational burden is to decompose probabilities hierarchically. This method reduces the $|V|$ factor in our calculation complexity to as low as $\log |V|$. We can do the hierarchy by building categories of words, then creating subcategories, sub-subcategories, and so on. For instance, we can make categories of animals, and make subcategories of pets and wild animals, and then continue by making a sub-sub category of aquatic pets and land pets, and so on. For simplicity, suppose each group has two subgroups. Finally, we can assign words to a suitable group. Then the probability that of a word w chosen, given that the word is in a category c_k , which is a subcategory of c_{k-1} , and so on until the highest category c_0 , is

$$P(w) = P(c_0) \times P(c_1|c_0) \times \cdots \times P(c_k|c_{k-1}, \cdots, c_1, c_0) \times P(w|c_k, \cdots, c_1, c_0).$$

This formula is imminent by looking at the categorical tree, and traversing down until we find our desired word. This means that we can represent w as a bit vector $(c_0(w), c_1(w), \cdots, c_k(w))$, where $c_i(w) = 0$ if in the i -th level w belongs to the left subgroup and 1 otherwise (here, left and right denotes their position in the binary tree). Then, for a dictionary with $|V|$ words, we need $\log_2 |V|$ bits.

To find the probabilities from each category to its list of subcategories, we can use a simple logistic regression model and provide the same input word to all of these models. We will know the ground truth for each input because, from the training data, we already know the true path to the correct category. The models will be trained by using cross-entropy as the loss function, which is defined as follows: Given two discrete probability distributions P and Q , the cross-entropy loss is equal to $c(P, Q) = -\sum_{i=1}^{|P|} P_i \lg Q_i$. This is good for our purpose because there are a finite amount of subcategories for each category (classification problem), and it is equal to maximizing the log-likelihood of the correct sequence of decisions. In the training,

each node i will have a $1 \times h$ vector u_i . Then, $P(\text{the } i\text{-th node equals } 1) = \frac{1}{1+e^{-h \cdot u_i}}$. Essentially, we will train the u_i in the training phase.

While it is tempting to try to optimize our tree structure, it is not a practical thing to do. The computational savings is not worth it because the approach only works on the final calculation to find the output. Suppose that we have l hidden layers, with each layer containing n_h nodes (hence, the output in this layer will be a n_h sized vector) and assume that n_b is the average number of bits required to identify a word. Then, updating weights in all hidden layer needs $O(l \times n_h^2)$ and the output needs $O(n_h \times n_b)$. Hence, if $n_b \leq l \times n_h$, it is better to reduce n_h instead of n_b . Note that $\log 500000 \approx 20$, so we can reduce n_b to 20. However, n_h will be much larger, around 1000. Hence, it is more optimal to make a tree with depth 2 and branching factor $\sqrt{|V|}$ rather than a tree with $\log |V|$ depth and branching factor 2.

We only talk about defining word hierarchy without actually telling how to divide the words. Early work uses pre-defined hierarchies but it is later found that the grouping can also be learned using a neural model. However, since the hierarchy is discrete, it is hard to optimize with gradient descent, and we must resort to discrete optimization methods to have a good approximation.

This model still has several weaknesses. Firstly, testing is still hard, as we need to find the probability of all words. It is also hard to find the most likely word in the tree data structure. Finally, empirical tests show that this method performs worse when the class choice is bad.

4.3 Importance Sampling

One way to speed up training is to not calculate the gradient from all the words that do not appear in the next position, but just update the weights of some of the words sampled randomly. We want incorrect words to have a low probability under the model. Note that when we calculate the gradient, using the output layer formula

from before, (o is the output layer before softmax)

$$\begin{aligned}
\frac{\partial \log(P(y|C))}{\partial \theta} &= \frac{\partial \log \text{softmax}(o)_y}{\partial \theta} \\
&= \frac{\partial}{\partial \theta} \ln \frac{e^{o_y}}{\sum_i e^{o_i}} \\
&= \frac{\partial}{\partial \theta} o_y - \ln \sum_i e^{o_i} \\
&= \frac{\partial o_y}{\partial \theta} - \frac{1}{\sum_i e^{o_i}} \frac{\partial \sum_i e^{o_i}}{\partial \theta} \\
&= \frac{\partial o_y}{\partial \theta} - \frac{1}{\sum_i e^{o_i}} \sum_i \frac{\partial e^{o_i}}{\partial \theta} \\
&= \frac{\partial o_y}{\partial \theta} - \frac{1}{\sum_i e^{o_i}} \sum_i e^{o_i} \frac{\partial o_i}{\partial \theta} \\
&= \frac{\partial o_y}{\partial \theta} - \sum_i \frac{e^{o_i}}{\sum_i e^{o_i}} \frac{\partial o_i}{\partial \theta} \\
&= \frac{\partial o_y}{\partial \theta} - \sum_i P(y = i|C) \frac{\partial o_i}{\partial \theta}.
\end{aligned}$$

We are using two basic chain rule facts: $\frac{\partial}{\partial \theta} \ln x = \frac{1}{x} \frac{\partial x}{\partial \theta}$ and $\frac{\partial}{\partial \theta} e^x = e^x \frac{\partial x}{\partial \theta}$. The last two steps are correct because by our definition, the probability of word i is chosen by the model is $\text{softmax}(o)_i$. The first term is the positive phase, increasing o_y and the second one is the negative phase, pushing o_i down based on the probability. If we try to estimate this, we need to have a distribution for $P(y = i|C)$, which is undesirable. Instead we can use a method called biased importance sampling. When negative word n_i is sampled, its gradient is weighted by $w_i = \frac{p_{n_i}/q_{n_i}}{\sum_{i=1}^m p_{n_i}/q_{n_i}}$. Here, we denote p_{n_i} as the probability that the word chosen is n_i . Note that the procedure here is to first sample the m words using the q distribution, then assign the weights w_i accordingly.

Then, we can approximate

$$\sum_{i=1}^{|V|} P(y = i|C) \frac{\partial o_i}{\partial \theta} \approx \sum_{i=1}^m w_i \frac{\partial o_{n_i}}{\partial \theta}.$$

where $\frac{\partial o_{n_i}}{\partial \theta}$ will be sampled from q . Here, what we do is basically a Monte-Carlo simulation, but we only do it m times, for some m . More details can be found here [2]. In principle, the (randomly) chosen m words will decrease in weight, while the rest stays the same. Then, we will model q using a simple model, such as 1-gram or 2-gram. The computational complexity will be reduced from V to m , where m is the number of word sampled, because we only update gradients of the randomly selected words (and the actual positive word).

There are several other methods to reduce the computational cost of training NLP models. Firstly, there is a ranking loss model, where the output of the model for each word is viewed as a score and makes the score of the correct word a lot higher than other words. The ranking loss proposed is $L = \sum_i \max(0, 1 - a_y + a_i)$. The i -th term will be zero if a_y 's score beats a_i by more than 1. However, this model does not give estimated probabilities.

4.4 Combining Models with n-grams

We have seen other models with better accuracy and advantages than the classic n-gram. However, this does not mean that n-grams are useless. One advantage of the n-gram model is that we can have high model capacity while requiring little computation time, by using data structures like a hash map ($O(1)$ lookup, think of it as a big table that can be accessed quickly). Hence, scaling the number of the dictionary does not affect the n-gram's performance: No matter how large is the dictionary size, we can get the probabilities instantly as they already stored by the hash map.

To add capacity to neural models without sacrificing performance time is to combine the approach with n-gram as an ensemble model. Assuming independent mistakes, ensemble methods will reduce test error. It is also possible to pair it with a maximum entropy model. In other words, we append the network with an extra set of inputs that directly connects to the output, without inferring other parts of

the model. They will be the indicators of whether some n-grams are present in the input, so it will be very sparse. The new inputs will have at least $|V|^n$ parameters, with V as our dictionary, but the added computation time is small because most of the entries are zero.

5 Recurrent Neural Network (RNN)

To understand machine translation, we need to understand RNN. RNN is a specialized Neural Network that is designed to process a sequence of data. The crux is that no matter how long is the sequence, we only make only one network to solve it. (For example, we do not need to make two separate models for sentences with 4 words vs sentences with 5 words).

To achieve this, we will use the idea of parameter sharing, which is using the same parameter in the different parts of the model. We do this because if the models use different parameters on different model parts, they cannot generalize to longer sentences. Also, it will not be able to share information between different positions. For example, consider two sentences "I was born in 2001" and "In 2001, I was born". If we want to use a conventional model to ask when I was born, it needs to learn that "2001" is the answer, no matter the position of "2001" itself. Hence, a normal neural network (with different parameters) will need to learn every grammar rule, in the sense that it needs to accommodate all possibilities.

For now, suppose that our inputs are x^t , where $t = 1, 2, \dots$ be the time step. The "time" here does not necessarily mean real-time (like a price in a stock market in a given time), but can just mean its position in the sequence (like the position of "2001" in the earlier sentence). Here, x^t is the token that is read by the network at time t . It is to be noted that RNN has a lot more useful than just machine translation, it can be used in time series analysis [17] or image processing [8], but such usage is not our focus.

5.1 Unfolding the Network

We will explore how to formalize the structure of a self-repeating neural network. This corresponds to a network with shared parameters.

First, consider the function $s^t = f(s^{t-1}, \theta)$, with s^t is a state on time t and θ is a bunch of hyperparameters. Note that s^t, s^{t-1}, \dots are functions, not a power of variables. It is easy to see that the function can be unfolded by using the definition repeatedly. Now, consider a new function $h^t = f(h^{t-1}, x^t, \theta)$, where x^1, x^2, \dots is an series of external signals (new inputs independent from f). Now, unfolding this will give us the entire signal up to the t -th member. This will be the function for our hidden layer. We note that now, the amount of hidden layer is dependent with the number of the input itself.

When the network is trained to predict the "future" (next word) given the "past" (list of previous words), it learns how to use the hidden layer h^t as a summary of the relevant aspect of the past sequence. Note that it is lossy since while the length of x may vary, the dimension of h is constant. The model might prioritize some aspects with more precision, depending on the usage. For instance, in an autocomplete, most weights will be placed on the most recent word, while little information will be stored in h about the earlier words.

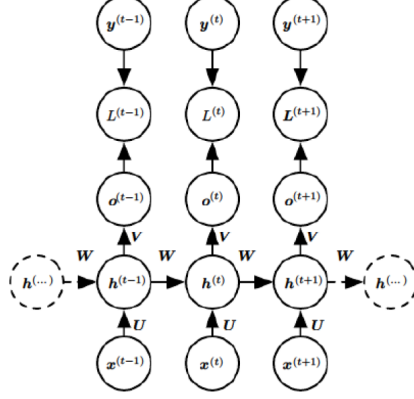
Denote the unfolded recurrence after t steps as g^t , and note that

$$h^t = g^t(x^t, x^{t-1}, \dots, g^1) = f(h^{t-1}, x^t, \theta).$$

Note that this representation has two advantages. First, the transition function f is used every time, with the same parameter θ . Second, the learned model has the same size regardless of the input length, because it is stated in terms of transition, instead of the history of variables. Hence, only one model is needed to cover all time steps and sequence length.

5.2 RNN

Now, we formalize our first design of RNN.



Here, x are our sentences, with words x^1, x^2, \dots, x^n that are already embedded into a vector, h is the hidden layer defined recursively, similar to the one in the previous section. The matrix U is the weight matrix that connects the input layer and the hidden layer, W is the weight matrix connecting hidden layers (saved as θ in the equation above) and V is the weight matrix from the hidden layer to the output layer. The loss is then computed by comparing each training data's output $\text{softmax}(o)$ with ground truth y , with o containing the entire output vector. There are other variants of RNN, such as one with only one input or one output, or a variant where output was given every step and connects to the next time step only from the previous output.

Now, the forward propagation equations for this particular RNN is as follows. For $t = 1, 2, \dots$, (the t in the vector v^t denotes its value on time t).

1. $a^t = b + Wh^{t-1} + Ux^{t-1}$,
2. $h^t = f(a^t)$,
3. $o^t = c + Vh^t$,
4. $y^t = g(o^t)$.

Here, we take $f(x) = \tanh(x)$ and $g(x) = \text{softmax}(x)$ for simplicity, although other choices of f, g are possible. To make sense of this equation, we can observe the arrow directions. Here, b and c are bias vectors. In this example, the input and output will have equal sizes. Total loss of the one training data is then just the sum of all loss over a time step. For example, if the loss function is the negative log-likelihood of y^t given inputs x^1, x^2, \dots, x^t , then

$$\begin{aligned} L(\{x^1, x^2, \dots, x^t\}, \{y^1, y^2, \dots, y^t\}) &= \sum_{i=1}^t L(t) \\ &= \sum_{i=1}^t L(x^i, y^i) \\ &= - \sum_{i=1}^t \log P_{\text{model}}(y^i | x^1, x^2, \dots, x^i). \end{aligned}$$

Note that here, $P_{\text{model}(X)}$ is the probability of X happening according to the model parameters. The number $\log P_{\text{model}}(y^i | x^1, x^2, \dots, x^i)$ can be seen by looking at the relevant position on the output vector.

5.3 Updating RNN

Finding the gradient of this loss function is very costly. There are $O(l)$ computations (l is the number of layers), and we cannot count them using a parallel system because each gradient must be calculated sequentially (they need the previous gradient). Also, they all must be stored to gradient update, so memory cost is also $O(l)$.

To be more precise, we will use the normal back-propagation algorithm (see appendix F) to train the RNN. In the equations above, we have parameters U, V, W, b, c as well as sequence of vectors x^t, h^t, o^t, L^t . We want to find $\nabla_N L$ for each vector, based on what follows it on the graph. Firstly,

$$\frac{\partial L}{\partial L(t)} = \frac{\partial}{\partial L(t)} \sum_{i=1}^t L(i) = 1.$$

Then, assuming loss is the negative log-likelihood, using the tanh function between layers, and using softmax as the final activation function on o to get the probability vector \hat{y} , we can see that

$$\begin{aligned}
(\nabla_{o^t} L)_i &= \frac{\partial L}{\partial o_i^t} = \frac{\partial L}{\partial L(t)} \frac{\partial L(t)}{\partial o_i^t} = \frac{\partial}{\partial o_i^t} (-\log \text{softmax}_t(o^t)) \\
&= \frac{\partial}{\partial o_i^t} \left(-\log \frac{e^{o_i^t}}{\sum_{i=1}^t e^{o_i(t)}} \right) \\
&= \frac{\partial}{\partial o_i^t} \log \sum_{i=1}^t e^{o_i(t)} - \log e^{o_i^t} \\
&= \frac{e^{o_i^t}}{\sum_{i=1}^t e^{o_i(t)}} - 1_{i,t} \\
&= \hat{y}_i^t - 1_{i,t}.
\end{aligned}$$

We continue to work backwards. In the final step l , h^l only affects o^l . Hence, using chain rule we have

$$\nabla_{h^l} L = \left(\frac{\partial o^l}{\partial h^l} \right)^T \nabla_{o^l} L = V^T \nabla_{o^l} L.$$

Then, we calculate backwards from $t = 1, 2, \dots, l-1$. Notice that o^t and h^{t+1} are affected by h^t . Set

$$H_{ij}^t = \begin{cases} 1 - h_i^{t^2} & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

Essentially H^t is the Jacobian for the tanh at time t . Hence, by chain rule again (the version with 2 dependency variables),

$$\nabla_{h^t} L = \frac{\partial h^{t+1}}{\partial h^t} (\nabla_{h^{t+1}} L) + \frac{\partial o^t}{\partial h^t} (\nabla_{o^t} L) = W^T \nabla_{h^{t+1}} L H^{t+1} + V^T \nabla_{o^t} L.$$

This can be seen because $\tanh(x)' = 1 - \tanh^2(x)$.

Now, we have calculated the gradients on the internal nodes. Our next step is finding the parameter nodes. We will use the backpropagation algorithm again.

However, $\nabla_W f$ considers the contribution of W for all calculations. Hence, we introduce W^t which is equal to W but only usable at time t . Then, the true contribution is $\nabla_{W^t} f$. Now we can conclude that,

1. $\nabla_c L = \sum_t \left(\frac{\partial o^t}{\partial c} \right)^T \nabla_{o^t} L = \sum_t \nabla_{o^t} L$,
2. $\nabla_b L = \sum_t \left(\frac{\partial h^t}{\partial b^t} \right)^T \nabla_{h^t} L = \sum_t H^t \nabla_{h^t} L$,
3. $\nabla_V L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^t} \right) \nabla_{V o_i^t} = \sum_t (\nabla_{o^t} L) (h^t)^T$,
4. $\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^t} \right) \nabla_{W^t h_i^t} = \sum_t H^t (\nabla_{h^t} L) (h^{t-1})^T$,
5. $\nabla_U L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^t} \right) \nabla_{U^t h_i^t} = \sum_t H^t (\nabla_{h^t} L) (x^t)^T$.

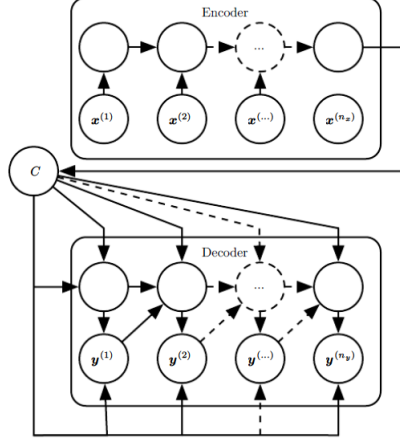
5.4 Encoders and Decoders

In the RNN above, we see that while we can make one model that accepts any length, we still have a restriction that the input and output length must be the same. However, this should not be the case in some usage, such as translation. For example, "Anak Perempuan" in Indonesia, (length 2) is "Daughter" in English (length 1).

For now, we call the input of the RNN the context. We want to represent this context into an input sequence x . A very simple idea is to create two RNN architectures, an encoder, and a decoder. Then, the encoder will read the input and outputs an embedding C . This C will be inserted to the decoder (C has a fixed length) to generate the output Y . Here, $|X|$ can be not equal to $|Y|$. Then, these two models will be trained jointly to maximize the average of $\log P(Y|X)$. The final layer of the encoder is the input of the decoder. Also, there is no restriction on building these two RNNs, in the sense that they do not need to be identical in architecture.

A problem is when the encoder spits out C which has a small dimension that cannot summarize the input well enough. The proposal is to make C a variable-

length sequence. Also, an attention mechanism is introduced to associate elements of C to the output sequence.



6 Neural Machine Translation

We will discuss a task of translation, which is to read a sentence (for instance, in English) to another sentence with equal meaning. At a high level, there is one component that proposes many candidate translations. For example, some language puts adjectives after nouns ("Apel Merah" in Bahasa Indonesia, which translated directly into Apple Red). The proposal mechanism suggests variants of the suggested translation, including "Red Apple". Then, a second component chooses the best variations, and "Red Apple" will score better than "Apple Red".

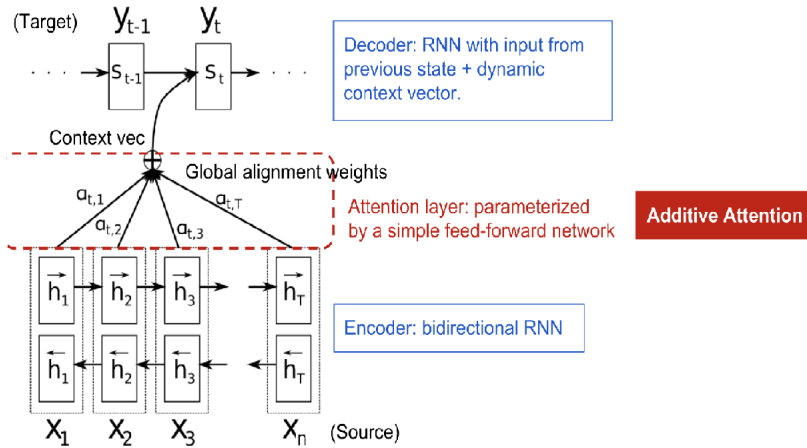
The earliest solution to this problem is to use a neural language model. They use not only the back-off n-gram model but also the maximum entropy language models, which just basically add a softmax layer to predict the next word.

Because machine translation needs to produce an output sentence instead of a single word, we need to extend the model to be conditional. Given a source sentence s_1, s_2, \dots, s_n , we want to find tokens t_1, t_2, \dots, t_k (k is not fixed) that maximizes $P = P(t_1, t_2, \dots, t_k | s_1, s_2, \dots, s_n)$. However, this assumes that the preprocessed sequence is fixed length. Hence, we should train an RNN to maximize P . We use the encoder-decoder idea similar to the one discussed above. To generate sentences,

we need a way to represent our source. Earlier, we were only able to represent words or phrases. We want to ensure that sentences that have a similar meaning ("I eat rice" vs "Rice is eaten by me" vs "Saya makan nasi") have similar representations.

6.1 Attention Mechanism

This paper examines the idea from the paper by Bahdanau [1]. If we use a fixed size representation to capture all details of a long sentence (a news article), it will be very difficult (not impossible, but takes a lot of time and computational power). The main reason this happens is that it already "forgets" the early part of the sentence when translating the later part. Another, more efficient, approach is to read the text first, and then produce each translated word by focusing on a different part of the input.



In this process, there are three main components, the encoder, the decoder, and the attention layer, which give the idea for the program on which words to focus while translating. The encoder is similar to the one introduced earlier (the figure uses bidirectional RNN, but we can use any RNN). The decoder is a RNN with equation $s_t = f(s_{t-1}, y_{t-1}, c_t)$ for word in position t . The vector c_t is the sum of hidden states in input sequence. And then we define $c_t = \sum_{i=1}^n \alpha_{t,i} h_i$.

The alignment model's job is to assign these $\alpha_{t,i}$ weights. This function states how well the two words y_t and x_i match. To define alignment, we need a notion of

score between words. Intuitively, a high score means that y_t is chosen because of x_i (For example, "Saya makan nasi" \rightarrow "I eat rice", because "nasi" means "rice", their alignment most likely is high). There are a lot of ways to define the score. This alignment score can be trained by a simple network. In other words, the score function is trained by a neural network. Other methods of choosing the score function, such as the cosine vector between two words or the dot product can also be used. After we define $score(x, y)$ with our choice of implementation, we define the alignment function

$$\alpha_{t,i} = align(y_t, x_i) = \frac{e^{score(s_{t-1}, h_i)}}{\sum_{j=1}^n e^{score(s_{t-1}, h_j)}}.$$

6.2 Transformer Model

This paper follows the paper by Vaswani et. al. (2015) [15]. A problem in RNN is that all training must be done sequentially, which prevents parallelization. We will learn about the novel Transformer model, a new architecture that does not use recurrence and focuses only on the attention mechanism to map dependencies from input to output. Because there is no recurrence, we can employ parallelization in our weight computation.

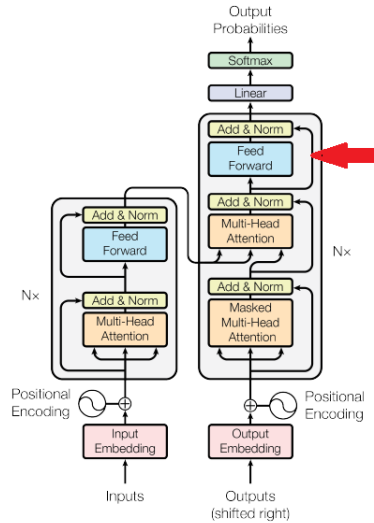


Figure 1: The Transformer - model architecture.

Let us start with the new attention system. We consider three vectors: Query,

Key, and Value. More precisely, given an input vector X and three matrices W_q, W_k, W_v , we define three vectors:

- Query Vector $q = XW_q$, this represents the current word.
- Key Vector $k = XW_k$, this is indexing for the value vector. The key-value relation here is similar to the hash map data structure.
- Value Vector $v = XW_v$, this represents the info from the input word.

The plan is now to for any query q_i , calculate the dot product of q_i and k to obtain a numeric value for each key. After this step, we want the closest key value to be the highest value, then apply softmax to find the probability (we can divide the value by the square root of the key vector to prevent the dot product to be too big, but this is optional). We then multiply this by the value vector. Finally, the larger the number, the more attention that word is given. Now, we can combine q, k, v into a matrix Q, K, V by simply stacking them. We then can find every attention using this formula

$$Attention(Q, K, V) = softmax(QK^T / \sqrt{d_k})V.$$

We examine the encoder(s) section. Same as before, it maps a input sequence (x_1, x_2, \dots, x_n) to a representation sequence (z_1, z_2, \dots, z_n) . In each piece of the encoder, there are two sublayers. The first one is a multi-head self-attention mechanism on the input vectors, which is similar to the normal attention mechanism, but parallelized. The second sub-layer is a simple network.

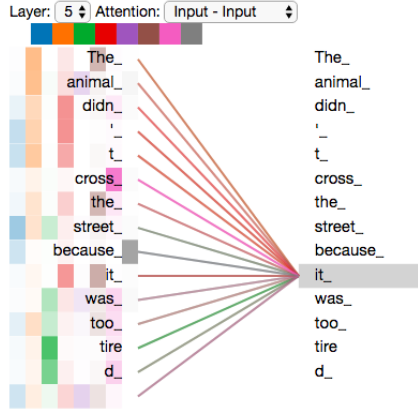
Next, we see the decoder. It takes the vector z obtained from the encoder and generates (y_1, y_2, \dots, y_m) as the final output sequence. There are three sub-layers. The first one is a masked multi-head self-attention mechanism from the output of the previous iteration. This masked multi-head is modified to prevent the decoder from take attention to future words. Hence, the i -th word can only see the first i

words. The second one is a multi-head attention mechanism from the encoder and the previous sub-layer. The final one is a simple feed-forward network. As a final note, in the paper, there are 6 layers of encoders and 6 layers of decoders. Also, each layer has the same output dimension (512).

Notice that between layers on the encoders and decoders, there are Add & Normalize layers and a residual connection (a design to skip some of the hidden layers, which is the crooked path in the figure that skips a layer, see the red arrow on the figure for an example). The residual connection simply adds the input vector and the output vector (note that the dimensions are the same, so the matrix addition is legal). Then, the Normalize part refers to the Layer Normalization operation. This is similar to the more common Batch Normalization, but the key difference is the former averages out so that the mean of each feature is zero and the variance is one, while the latter averages out each batch (one training data) so that the mean is zero and variance is one.

For the multi-head attention, we want to let our model focus on different positions by calculating self-attention with different sets of q, k, v and take the average. To do this, we reduce the size of q, k, v by creating multiple matrices ($W_{q,i}, W_{k,i}, W_{v,i}$) depending on the number of heads (8 in the paper). Now, for each triplet, calculate Z_i by a similar method to the above. To find the final output Z , first, we concatenate the Z_i 's sideways and write it as Z' . Suppose our dimension of Z is $r \times c$. Then, dimension of Z' is $r \times nc$. Now, train a weight matrix W_o with dimension $nc \times d$, so that $r \times d$ is the correct output size. Simply multiply $Z' \times W_o$ as our final output.

Intuitively, each different head, with the random initialization, will learn different attention rules. The good thing is that each head can be calculated separately, so it essentially allows us to have more "brains" with similar time training. In the picture below, the eight heads look at different things, which may represent different logic.

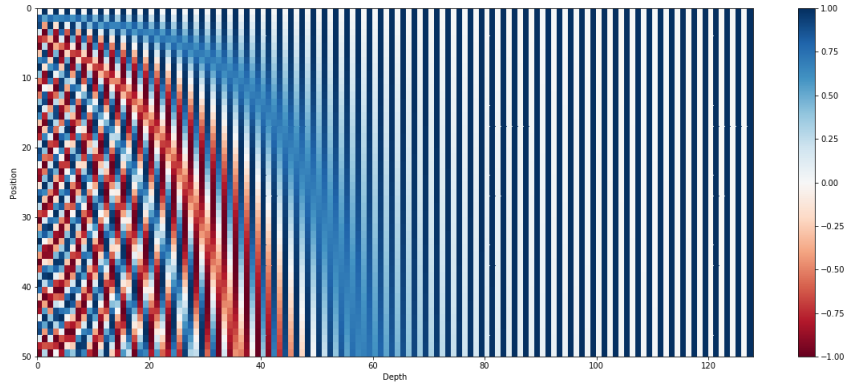


Another important idea is to do the positional encoding on the input. We need to do this because we need to account for the ordering of the word from the input as different ordering has a different meaning (remember, we do not use RNN here, so we can not exploit the benefits of RNN). To do this, we add positional encoding vectors, based on a specific pattern. The addition of these positional encodings is done before entering both encoders and decoders. The paper defines the positional encoding vectors for any position, PE_{pos} as follows.

$$PE_{pos,2i} = \sin(pos/10000^{2i/d_{model}}),$$

$$PE_{pos,2i+1} = \cos(pos/10000^{2i/d_{model}}).$$

The values can be seen in the graph below.



Note that this is not the only valid positional encoding. However, this is chosen by the paper because it seems by empirical results that it can learn by relative positions

since PE_{pos+k} is a linear function from PE_{pos} . It is also a possibility to use learned embeddings instead.

The final two layers are a linear layer to project from the final layer into the output vector. This vector o contains all the known vocabulary by the model. A simple softmax layer turns this into a probability, and a simple scan on which is the largest model gives the output word.

7 Generative Pre-trained Transformer 3

We turn our attention to the Generative Pre-trained Transformer [3], a revolutionary model that sets a new standard in NLP.

7.1 Motivation and Prior Work

While Transformers is very good, they have several limitations. Firstly, different tasks (English to French, English to Russian, Completing English sentences, etc) need different parameters (ranging from hundreds of millions to tens of billions), different datasets (that are very large and very different in each task) or even different architectures (which means a lot of work to do), to reach the best possible performance. We would like to build up the idea from the transformer to introduce GPT-3, a single model by OpenAI that can be used in a lot of different tasks. Then, to solve different tasks, we use this model and give a few descriptions of what the model is supposed to do with the data given.

There are several motivations for why we want to do this switch. Firstly, this will be very practical in daily use. To train a model in solving a task, there is a need for a huge labeled dataset (the data should be solved before being given to the program). Because there are a lot of useful tasks to do (translation, summarization, etc.), this tedious and hard process must be done for each one of them.

Secondly, it is easier to exploit false relations in data when the model expressive-

ness is high (can output a lot of "answers") and narrowness of training distribution. In the normal pre-train and fine-tune paradigm (train into a large dataset, and then fine-tune for a specific task), this can cause problems. For instance, there is a case where the generalization under this method is poor because the model is too specific on the distribution. Hence, it is possible that while a model does well on a specific benchmark dataset, it may not do so well in the real-time setting.

Finally, it is because of how humans learn. We do not need to be given a lot of examples to understand a command. Mostly, none ("Give me an example of a sad song") or a few ("Here are two examples of a poem, now make one") examples are enough for people to understand and do the task with some degree of competence.

To achieve this, there are some methods that we can do. Firstly, we can apply meta-learning, where the model learns a lot of skill and pattern recognition at the training time, and then recognize the task given at inference time. However, tests on the benchmark dataset show that the model is lagging inaccuracy on tasks behind the respective task's state of the art. So, we turn to the other revolutionizing trend in the field: parameter increase. For an illustration, the number of parameters of models has increased from 10^8 in 2018 (GPT-1, the earlier version of this model) to 1.7×10^{10} parameters in 2020 (Turing Project, a similar model by competitors in Microsoft). Empirical results show that an increased amount of parameters lead to improvements in performance. Hence, while there is no more big breakthrough in methodology such as Transformers, the field of NLP now races on computation power.

7.2 Model Specification and Training

During the unsupervised pre-training (i.e., a dataset without any labels), the model will develop a set of skills and learn to recognize patterns. Here, the model will not learn anything, but rather learn about how to learn. Then, given the prompt by the user (for example, "Please translate from English to French"), it recognizes the

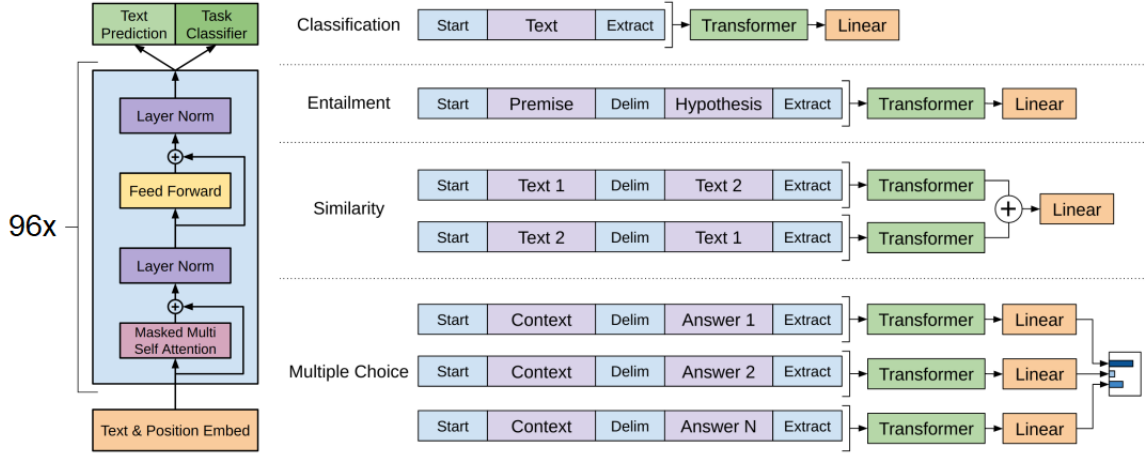
task and does it based on the training.

We shall define a few different types of how can we train a model.

1. A fine-tuned learning is an approach that updates weight regularly using a supervised training model. This is the most specific, but easy way to solve NLP problems. GPT-3 can be used on this, but it is not the main focus. The best models for one specific task are usually trained by this kind.
2. A zero shot model predicts the answer by just the task description and the prompt. For example, ("Please add these two numbers. 2 plus 3 equals ..."). No weights are updated.
3. An one-shot model predicts the answer by reading the task description, a single example, and the prompt. For example, ("Please add these two numbers. Two plus three equals five, seven plus one equals ... "). No weights are updated.
4. A few-shot model predicts the answer by reading the task description, a few examples, and the prompt. For example, ("Please add these two numbers. Two plus three equals five, seven plus one equals eight, three plus three equals ..."). No weights are updated.

Then, to solve different tasks, we use this model and switch the last layer depending on the given problem.

The OpenAI team releases various GPT-3 variants with different sizes, but we focus on the largest one, called GPT-3 175B or simply GPT-3. There are 1.75×10^{11} parameters (more than 10 times the parameter of the Turing Project, just in 2020), It has 96 layers with 12288 dimensions, 96 heads with 128 dimensions, 3.2×10^6 batch size, and 0.6×10^{-4} learning rate. The overall architecture is similar to the previous version of GPT-2. It is essentially a bunch of decoders (without an encoder) stacked on top of each other. Here is the overall picture of the GPT-2 model [13] (which is also shared by GPT-3).



The left figure is a decoder that gets stacked 96 times, then the output has two components, the first one determines the type of the task, and the second one will output the predicted text. The right figure is some example format of the possible prompts. For instance, the classification task's prompt will consists of a start token $\langle s \rangle$, used as the marker to begin the sentence, the actual text (usually a document or a news article), and an extract token $\langle e \rangle$. Then, these will be taken to the model, and the output will be attached to a simple neural network to give the classification.

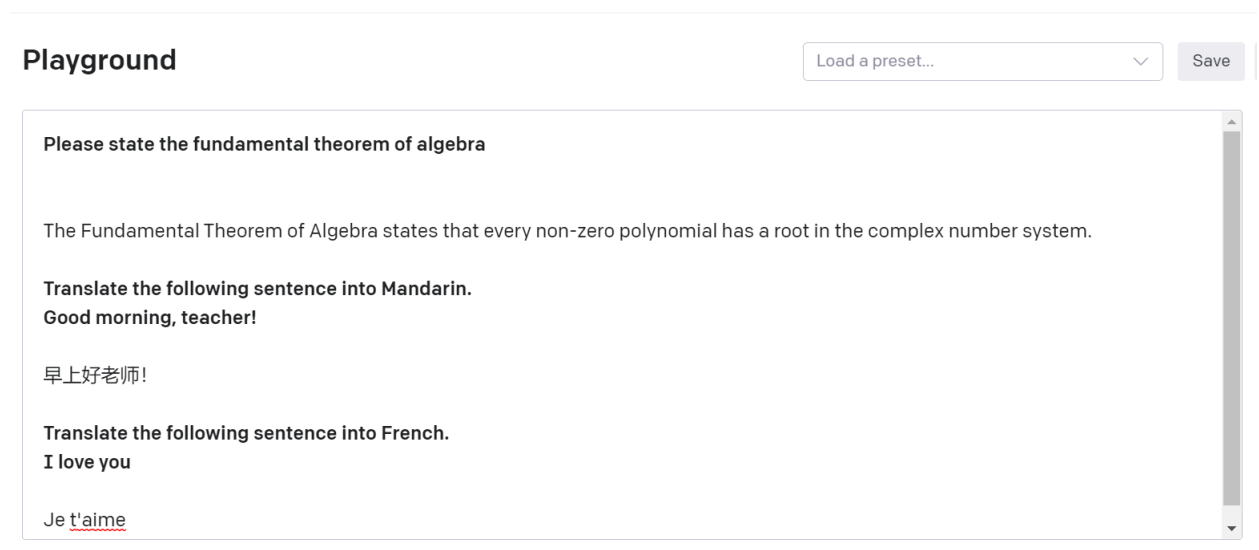
Several modifications are made from GPT-2 to GPT-3. The first one is that GPT-3 uses alternating dense and locally banded sparse attention patterns in the transformer layers. Sparse Attention is a variation of the attention mechanisms that only computes a limited amount of similarity scores from sequences, resulting in a sparse matrix (with zeroes for uncalculated scores). This alternative is chosen to reduce the time complexity of the matrix multiplications to $O(n\sqrt{n})$. In the GPT-3, the decoders used will alternate from the normal one to the sparse variant.

To use the GPT-3, it first went through unsupervised training to read a lot of texts from online sources such as Common Crawl, WebTexts2, Books1, Books2, and Wikipedia articles, with a weighted mix in the corpus. The model gets trained for 3×10^{11} tokens, and it took a huge amount of resources, with 3640 petaflop/s-day, and one petaflop/s-day is equal to using 8 V100 GPUs for an entire day (for

reference, a V100 GPU is equivalent to running one high-end laptop in terms of computing power). The time, money, and energy cost is so large that when the developers discovered a bug in the dataset, it is infeasible to retrain the model from scratch.

7.3 Model Usage and Performance

After the training was done, the GPT-3 now has its 175 billion parameters known and it is ready to be used for other tasks. If we want to use this model ourselves (which is available in the sandbox here). The weights are hidden because GPT-3 is privately owned by Microsoft (who is now an investor in OpenAI). The cool thing is that after the painstaking training, all the GPT-3 needs is for us to state the problem, give some examples if needed, and it will give a response without needing to specialize it for that specific task. For example, in the picture below, I run three separate queries, one about closed-book knowledge, one about translation from English to Mandarin, and finally the last one is a translation from English to French.



As we can see, the model does a good enough job for these questions (the FTA should state non-constant instead of non-zero, as $P(x) = 1$ has no root), despite

being three completely different prompts. Under the traditional method of NLP, we need to train three different models (may have the same architecture), find lots of relevant text for each of these three questions (which has zero intersection), spend some time training them (can be done all at once, but computational cost increases) and then create three different apps. All of these work is cut down by GPT-3 after the initial training.

We know that GPT-3 is great at being a generalist, but now we turn the attention on how good it is compared to the models designed exclusively for a given task. For different tasks, a state of the art model will be examined, along with the dataset used by the model and we use GPT-3 to train on that dataset. According to the paper, there are some specific tasks that have comparable or even favorable performance, such as completing a text in a paragraph and closed book question answering. However, some tasks still leave a lot to be desired, such as reading comprehensions and language inference.

7.4 Improvements

There are several things that can be improved on the model. Firstly, it struggles in text synthesis and other NLP tasks. Although general text generated by GPT is good, they can repeat themselves down the line ("I was born in 2002, ..., in 2002 I was born ...), contradict themselves ("I am a human, , I am a cat), or just throwing nonsense. It is also struggle for common sense tasks ("If I put water in fridge, will it become ice?").

Looking at the algorithm side of GPT-3, because it uses encoders without bidirectional architectures such as bidirectional RNN, it struggles to solve tasks that need to reread the text after knowing the questions or fill the blank tasks. A future improvement is to instill bidirectional model with the size comparable to GPT-3.

Other problems is the efficiency of the training. Humans, although read less texts in their lives than the GPT-3, they can solve tasks that GPT-3 struggles

easily. This means that the "training phase" can be more effective, but the source of this improvement is still not known. Also, like deep learning models, it is hard to check the inside mechanics of the GPT-3, which hinders its interpretability.

Finally, we talk about the impact of GPT-3 beyond NLP tasks. In a sense, the power of the model can be used against itself. For instance, the GPT-3's passable ability to generate texts can be used maliciously from email scams to academical misinformation. A tests by GPT-3 creator has shown that human can only predict barely more than half accuracy in detecting fake news generated by the model. The paper also states the issues of potential bias by the model in respect to gender, race, religion and so on. The environmental issues also looms large, as the power used to train GPT-3 is high. However, because we only train once, after the initial training, the power amortized for the expected lifetime of the model is efficient.

8 Codex: Writing Code to Write Code

8.1 Introduction and Task Evaluation

We will discuss Codex [4], a GPT-based model that is fine tuned to specialize its ability to write Python codes. We also touch a bit on the history of code-writing attempts before this paper.

The challenge of program synthesis is a task to construct a program to satisfy a task. This is way harder than program verification, which the goal is to just check whether a program works as intended. The idea of program synthesis has been on since the sixties, but with the advent of language models and presence of code in large datasets, there is now a lot of progress on this field. Early testing on the GPT-3 model shows ability to generate simple Python programs. This is exciting because GPT-3 is a general tool, not exclusive to code generation. Hence, a group of OpenAI researchers tries to fine tune the GPT, called Codex, to excel at programming tasks.

To test the coding power, the researcher makes some programming codes with

unit tests (here, we only concerned about the correctness). We did not use BLEU (see appendix B) like most sentence generation problem because past papers has shown that BLEU fails to capture semantic meanings of some code. Indeed, they calculated the BLEU of correct/wrong answers and they showed a poor correlation. It includes questions such as math, algorithms, and simple interview questions. Some examples are shown below (taken from the paper). White background is the prompt, and yellow background is the (successful) response from Codex.

```
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]

def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0, len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

To test, we give Codex a little leeway. Instead on requiring it to give a simple answer, it is allowed to output several answers, and the test is considered a success if any one of the outputted code works as intended. We give this relaxation because in the real world, humans also make errors on coding, and they can fix before being published ("debugging").

To make context on the research, Codex managed to solve 28 percent on the problems only using one code (one attempt), an improvement from 11 percent by another group (they also uses GPT, but has only half of the variables). In contrast, almost all untrained GPT score near zero percent (GPT can write simple code, but not to the extent of the dataset). A more advanced model, Codex-S, which is trained with real questions and answers, managed to do up to 37 percent. When we allow Codex-S to made 100 attempts, it solves 77 percent of the dataset. However, testing each sample is very inefficient in real life, as some attempt may be very bad. Instead, when Codex-S outputs only the most likely answer (according to itself), the accuracy is around 44 percent.

8.2 The pass@k Metric

We first formalize the pass@k metric. Basically, pass@k means that we generate k problem, and return the fraction of the correct code (which gets at least 1 passed code). However, this has high variance. Instead, we redefine it by generating $n \geq k$ samples per task, find the total of correct answers c and calculate

$$\mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right].$$

We can see that this is equivalent, as the $\frac{\binom{n-c}{k}}{\binom{n}{k}}$ is the probability of among c correct answers, none are chosen. (Basically, we add more samples). Note that c follows the $\text{Bin}(n, p)$ distribution and $P(c = i) = \binom{n}{i} p^i (1-p)^{n-i}$. We can show this is unbiased, i.e.

$$\mathbb{E}_c \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] = 1 - (1-p)^k,$$

where p is the probability of the code produces a correct answer. Indeed (note if $n - c < k$, $\binom{n-c}{k} = 0$ by definition),

$$\begin{aligned} \mathbb{E}_c \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] &= 1 - \mathbb{E}_c \left[\frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \\ &= 1 - \sum_{i=0}^{n-k} \frac{\binom{n-i}{k}}{\binom{n}{k}} \binom{n}{i} p^i (1-p)^{n-i} \\ &= 1 - \sum_{i=0}^{n-k} \binom{n-k}{i} p^i (1-p)^{n-i} \\ &= 1 - (1-p)^k \sum_{i=0}^{n-k} \binom{n-k}{i} p^i (1-p)^{n-k-i} \\ &= 1 - (1-p)^k. \end{aligned}$$

As a side note, calculating large binomials may cause numerical instability due to factorials are very large. Hence, the paper use the following, more stable, definition.

$$pass@k = \begin{cases} 1 & \text{if } n - c < k \\ 1 - \prod_{i=n-c+1}^n (1 - \frac{k}{i}) & \text{otherwise .} \end{cases}$$

8.3 Model Training

Back to the model itself, now we inspect the fine-tuning process. As stated before, the GPT used is the one with 12 billion parameters. The training dataset compromises from published (public, free to see) codes from GitHub (website for collaborative coding, similar to Overleaf but codes instead of latex).

Firstly, the researchers find it surprising that directly using the pre-trained model gives little improvement. They hypothesized that this is because GPT's dataset is so large. To inherit the text representations from GPT, the tokenizer of the code is aligned to the tokenizer of GPT. However, this has a side effect, because words that appears on codes has different distributions from the words that appears on the normal text (code using the variable "eat" is rarer than text using word "eat"). Special tokens are made to take care whitespaces (many code has whitespaces to substitute for space, like `merge_sort()`).

Then, when testing to calculate $pass@k$, each question is phrased like the figure earlier in the section (which is similar format with codes present on interviews for software engineering jobs). Then, Codex will generate until one of the end tokens appears. To sample, we do not use the normally used top-k sampling. Instead, we employ nucleus sampling, which means that we select the smallest set of token words based on the probability according to the previous tokens (hence, we greedily from the most probable to the least probable), until it reaches some limit p . After that, we ignore the tokens outside the set and sample from there. Here, we set $p = 0.95$.

The test result shows that the loss obeys the power law in model size (in short, test loss is generally inversely proportional to the the parameters, hence why it is often wise to expand the number of variables without fearing diminishing returns).

The sampling temperature (high temperature means more linguistic, see appendix C) is set to 0.8 for the model.

There is also a case where we cannot directly test the code that are generated (for example, an autocompleter in an coding environment). However, we must only give one output to prevent the user to be confused. In this case, empirical results shows that we should choose the sample with the highest mean token log probability.

8.4 Model Results and Improvements

Now we talk about the results gained by the Codex. It shows improvements from the competitors, GPT-J and GPT-Neo by other researchers in the HumanEval dataset. The results of GPT-J version with 6 billion parameters is equal to the Codex version with 300 million parameters. Then, Codex is tested in the newly made APPS [10] dataset, which consists of competitive programming codes (harder than interview questions). Below is an example problem from this dataset.

Problem

You are given two integers n and m . Calculate the number of pairs of arrays (a, b) such that: the length of both arrays is equal to m ; each element of each array is an integer between 1 and n (inclusive); $a_i \leq b_i$ for any index i from 1 to m ; array a is sorted in non-descending order; array b is sorted in non-ascending order. As the result can be very large, you should print it modulo $10^9 + 7$. Input: The only line contains two integers n and m ($1 \leq n \leq 1000$, $1 \leq m \leq 10$). Output: Print one integer – the number of arrays a and b satisfying the conditions described above modulo $10^9 + 7$.

Here, we give lenience by removing the time limit on the codes (which is the hard part on the real competitions) and only consider codes that pass the "training example". It still shows to be better than its competitors.

Next, we talk about the possibility of fine-tuning Codex into a specific usage. When scraping GitHub, there are often files or texts that are unrelated to making code (raw files, configurations, etc.). We want to add more problems to "repair" the distribution of the Codex task. The researcher sources questions from programming competitions and GitHub codes with continuous integration. For this stage, the Codex is run for each question to filter these which are too hard or have a non-

deterministic element. After training with this dataset, it shows that the new Codex-S increases in many metrics over the original Codex. The heuristics on Codex also works similarly with Codex-S.

Finally, the researchers explain Codex-D, which is specialized in declaring the explanation (in human words) about what is the code function. (In the picture earlier, it is the words between the `"""` strings). Because the scoring description is hard to do automatically, it is done manually by the researchers. Some common failings in the Codex-D are copying entire code or forgetting important constraints ("return the answer" vs "return the answer in capital letters"). This direction is harder to train because in most codes on GitHub, the descriptions are not really helpful and sometimes informal.

Now we move on to Codex's weaknesses. Firstly, it is sample inefficient, as human software developers have seen less code than Codex but perform much better. It also sometimes spit out undefined variables/functions or forgot some basic grammar. Like GPT, it sometimes struggles with long tasks and has problems associating which variable in the question to the code. In the sample below (given in the paper), it forgets to subtract 4 from w and failed to return the proper product. It also performs worse when the example deliberately includes faulty examples.

```
def do_work(x, y, z, w):  
    """ Add 3 to y, then subtract 4  
    from both x and w. Return the  
    product of the four numbers. """  
    t = y + 3  
    u = x - 4  
    v = z * w  
    return v
```

Finally, as with GPT, Codex has also problems beyond coding. In future development when Codex is better than it is now, it can eliminate jobs and help create malicious code as cyber attacks. It also can be prompted to deliver racial bias given the right input.

9 Applications: Solving and Generating College Math Questions

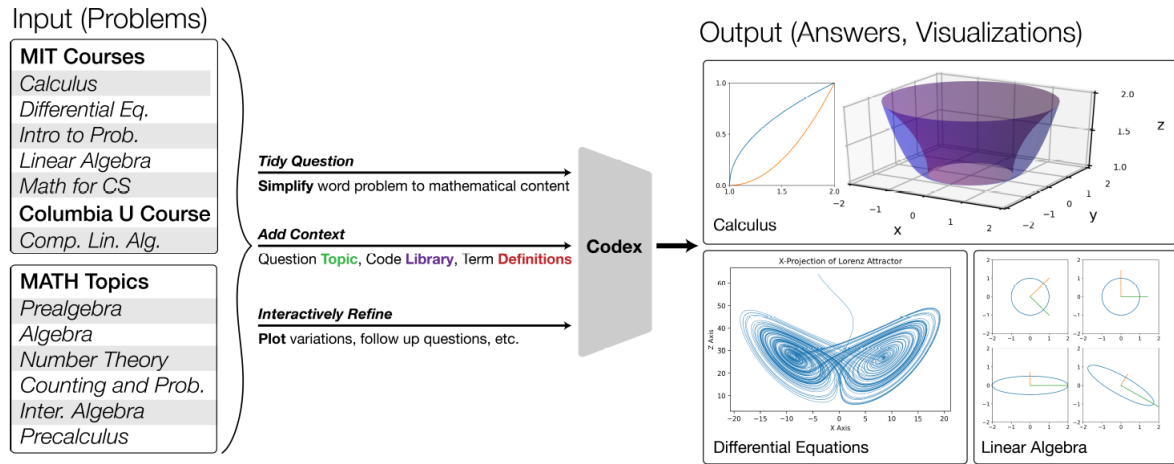
9.1 Introduction and Dataset

We now turn our attention to an application of the recent development of language models in teaching [6]. Before the paper, several attempts have been done using text-based pre-training. However, it either does not generate good results or is too specific on one level or subject. It is known that text-based learning has poor results in solving math problems. The new idea is to combine text-based and code-based (Codex) pre-training to train the code. The paper will show that not only it can generate answers, it also can create questions of similar quality to human-made problems. Also, it will be trained to solve questions from entry-level university courses and give visualizations as well as new problems related to the question. The model also shows a 7-8 percent increase on MATH benchmark dataset, which consists of competition math (AMC problems, which is similar to SMO but for the United States) questions. Here is a sample question in the dataset.

Problem: Tom has a red marble, a green marble, a blue marble, and three identical yellow marbles. How many different groups of two marbles can Tom choose?
Solution: There are two cases here: either Tom chooses two yellow marbles (1 result), or he chooses two marbles of different colors ($\binom{4}{2} = 6$ results). The total number of distinct pairs of marbles Tom can choose is $1 + 6 = \boxed{7}$.
Problem: If $\sum_{n=0}^{\infty} \cos^{2n} \theta = 5$, what is $\cos 2\theta$?
Solution: This geometric series is $1 + \cos^2 \theta + \cos^4 \theta + \dots = \frac{1}{1 - \cos^2 \theta} = 5$. Hence, $\cos^2 \theta = \frac{4}{5}$. Then $\cos 2\theta = 2 \cos^2 \theta - 1 = \boxed{\frac{3}{5}}$.
Problem: The equation $x^2 + 2x = i$ has two complex solutions. Determine the product of their real parts.
Solution: Complete the square by adding 1 to each side. Then $(x + 1)^2 = 1 + i = e^{\frac{i\pi}{4}} \sqrt{2}$, so $x + 1 = \pm e^{\frac{i\pi}{8}} \sqrt[4]{2}$. The desired product is then $(-1 + \cos(\frac{\pi}{8}) \sqrt[4]{2})(-1 - \cos(\frac{\pi}{8}) \sqrt[4]{2}) = 1 - \cos^2(\frac{\pi}{8}) \sqrt{2} = 1 - \frac{(1 + \cos(\frac{\pi}{4}))}{2} \sqrt{2} = \boxed{\frac{1 - \sqrt{2}}{2}}$.

Another good thing about the paper is that the idea is simple and applicable to other problems. The idea is to use a Transformer model pre-trained ("trained on large database") on the text and fine-tuned ("trained on specific task") on code. Codex will be used in the second part. Another idea is to add context information

to the problem, such as topic, domain knowledge, and which libraries to use (for plotting, etc.). The rationale is sometimes in a class, there is "overhead information" that is used (for example, in data science courses, we use NumPy), which must be relayed to the model to ensure fairness.



To begin the training, random questions are taken from the courses and topics in the figure above (25 for university courses, 5 for the MATH topics). To make sure that the model did not overfit the data, a test is done with new course material (for the new Academic Year) yet to be seen by the model. Sample questions are provided in Appendix E and the paper. The materials and the class code for these six classes are found in Appendix G.

9.2 Methodology

The input for the model will be a problem and our goal is to transform the question into a programming task. Then, Codex will take this programming task and return the answer by code. However, not everything can be directly fed into Codex. Three ways are mentioned: adding contexts to the question, tidying texts, and interactive refinement.

The idea of adding context is that students in these classes have implicit knowledge of the course contents. These may include class notations (for example, we

assume vectors are written horizontally), implicit assumptions (in the intro to linear algebra classes, assume all matrices are real, not complex), and some standard operating procedures explained in the lecture notes or accompanying textbooks (in basic calculus, we learn how to use L'hôpital to find limits).

In more detail,

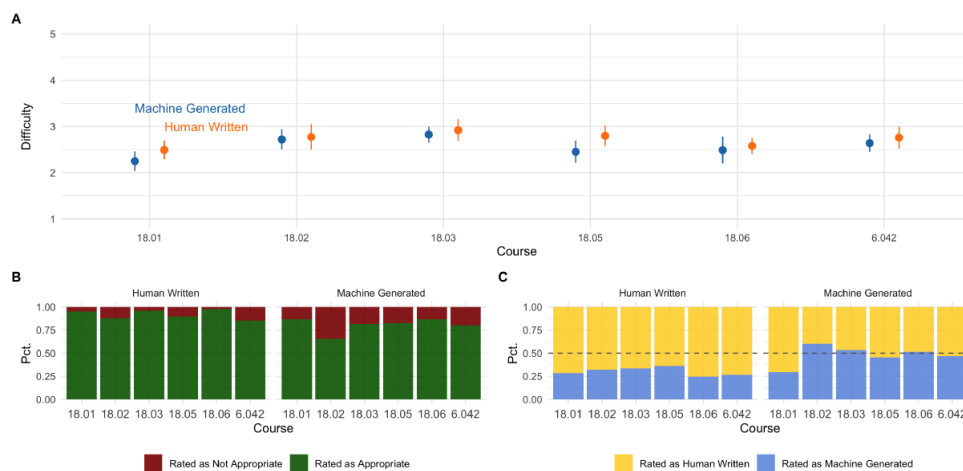
1. Topic context states the question domain so that students (and the model) can use the proper method to solve them. For example, solving questions relating to "analysis" is very different, whether it is about "real analysis" or "complex analysis". Usually, only the big topic ("Linear Algebra") and the subtopic ("Eigenvalues and Eigenvectors") are needed. Then, the word "In Linear Algebra ..." is added to the problem statement.
2. Library context states the programming languages and packages that are needed to solve the problem. This is important because, without these, the model might use them while not importing them, resulting in a syntax error. This is mainly useful for plotting functions.
3. Definition context is mainly used to streamline what methods or conventions are used to supposedly solve the question in class. For example, in discrete mathematics class, it is often okay to leave the answer in the form of $\binom{n}{k}$ instead of needing to calculate the value. Also, it is useful to explain things that are obvious for humans but not to models, like there are 12 months a year or there are 52 cards in a poker deck.
4. (Optional) Question tidying (similar to denoising in normal datasets) is sometimes used to remove useless information. For example, in the question: "*After Superman defeats Lex Luthor, he decides to buy some candies.* If one candy costs 0.1 dollars, how many candies can Superman buy with 30 dollars?", the italicized part is useless. It also involves breaking long sentences into smaller components and converting tasks into programming format.

5. (Interaction) Some questions have multiple parts that are needed to chain together to create the final answer. For this, we can run the model multiple times to get the desired answer, changing the prompt every time.

For examples of this process, consult part E of the Appendix.

Next, we learn about how we can generate new questions for each course. The prompt is done in the following manner: First, from a course, problem dataset, create a list of a random amount of questions, and we feed this as a prompt to generate the next question (kind of like few-shot learning).

To test the quality of the questions, a survey is conducted among the university students who take the relevant courses. They will be given 5 original questions and 5 generated questions and requested to assess the suitability and difficulty of the questions, as well as guess whether the problem is human-made or machine-made. The result of the surveys shows that the questions are on a similar level to each other and human questions are a bit more suitable for a course. It is also seen that differentiating between man-made problems and machine problems is not a trivial task. The full survey is in the table below. Panel A is the difficulty rating by students (1 easiest, 5 hardest), Panel B is suitability, and Panel C is the student's guess of the source problem.



Here are some samples of generated questions, taken from the paper. Note that

each question takes only one second to make. The similarity is calculated using Cosine Similarity.

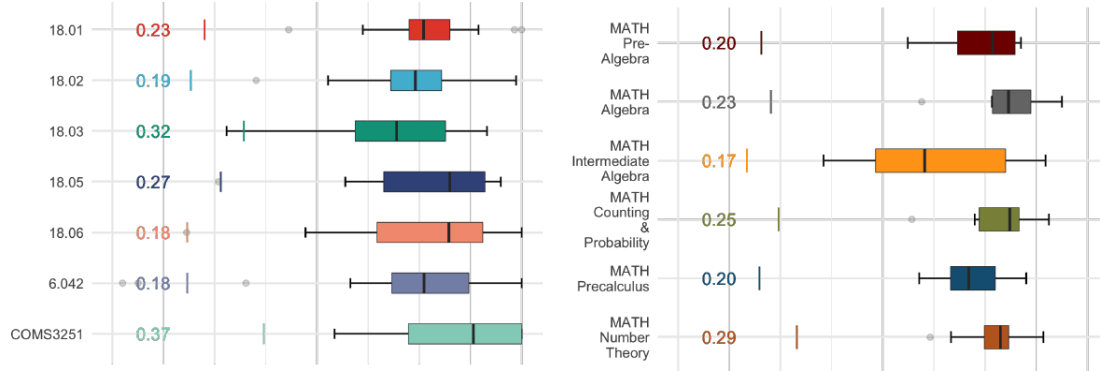
ID	Course	Machine-generated question	Closest question in the dataset	Similarity
1	18.01 Single-Variable Calculus	Find the area of the region bounded by the curve and the x-axis. $y = x^2 \sin(x), 0 \leq x \leq \pi$	Find the area of the region under the given curve from 1 to 2. $y = (x^2 + 1)/(3x - x^2)$	0.61
2	18.02 Multi-Variable Calculus	Find $a \times b$. $a = \langle 9, -2, 1 \rangle$, $b = \langle -2, 1, 1 \rangle$	Find $a \times b$. $a = \langle 5, -1, -2 \rangle$, $b = \langle -3, 2, 4 \rangle$	0.87
3	18.03 Differential Equations	Use the method of separable variables to solve the initial-value problem $\frac{dy}{dx} = 5e^x$, $y(2) = 12$ when $x = 2$	Separate variables and use partial fractions to solve the initial value problems. Use either the exact solution or a computer-generated slope field to sketch the graphs of several solutions of the given differential equation, and highlight the indicated particular solution. $f'(x) = 3f(x)(5 - f(x))$, $f(0) = 8$	0.21
4	18.05 Introduction to Probability and Statistics	Let X be a uniformly distributed random variable over the interval $[0, 1]$. Find $\mathbb{E}[X^2]$	Let X be the result of rolling a fair 4-sided die. Let Y be the result of rolling a fair 6-sided die. You win $2X$ dollars if $X > Y$ and lose 1 dollar otherwise. After playing this game 60 times, what is your expected total gain?	0.29
5	18.06 Linear Algebra	Write a Matlab code to determine if the given matrix $A = [1, 1; 4, 4]$ is positive semidefinite and if it is negative semidefinite.	Find $A'A$ if the columns of A are unit vectors, all mutually perpendicular.	0.21
6	6.042 Mathematics for Computer Science	A student is taking a test consisting of n multiple-choice questions. Each question has five possible answers, and only one of them is correct. The student knows that the probability that any particular question is answered correctly is $\frac{1}{5}$. Let X be the number of questions answered correctly by the student. What is $\mathbb{E}(X)$?	MIT students sometimes delay laundry for a few days. Assume all random values described below are mutually independent. A busy student must complete 3 problem sets before doing laundry. Each problem set requires 1 day with probability $\frac{2}{3}$ and 2 days with probability $\frac{1}{3}$. Let B be the number of days a busy student delays laundry. What is $\mathbb{E}(B)$?	0.47

9.3 Model Result

The result shows that the model can solve questions from courses correctly. We want to check how good was the "translation" process from the original questions and the final prompt that is given to Codex. To use this, we consider the Sentence-BERT (see appendix D) between the prompt and the question and calculate its cosine similarity

$$S_C(u, v) = \frac{u \cdot v}{||u|| ||v||}.$$

Here is the result in detail. The boxplot shows the average cosine similarity between a raw question and its cleaned form. The colored line is the average cosine similarity, taken by averaging each pair of questions in the dataset.



However, several challenges need to be solved for this to be viable for mass usage in universities and schools. Firstly, it struggles with images due to the nature of the model. Hence, for questions with figures such as some high school geometry or basic statistics questions, we need some way to translate the image into text. In this experiment, most questions do not contain tables or images. It is plausible that this can be combined with specialized image processing models (like this one [11], that can extract table content to text) to combat this problem.

Secondly, it still struggles with long proof questions, like the one appearing in advanced math classes. However, it is conjectured that this is due to the lack of training data used, as short proofs are done successfully by Codex in the experiment. Some progress has been made on attempting to do longer, more complex proofs. For example, [16] has managed to disprove a conjecture in graph theory by using Artificial Intelligence. However, no "universal" proof machine existed yet as of now.

Also, while it can create many similar problems for a tutorial setting, it is harder to make an entire exam by itself, due to difficulties in controlling the difficulty of the question and no guarantee that the subtopics tested have the same distribution as the intended one. There is currently no way for itself to predict the difficulty of a question. Finally, some questions take too much time in theory, such as first-order DE, diophantine equations, and so on. We use a "cheating" assumption that because it is given as homework/exam, we know that the question is solvable within a reasonable time.

A A Small Note on Decode Output

In the decoder architecture, we have stated that we pick the most probable word in each word to create the desired translation. In this appendix, we consider beam search, an algorithm to improve the output on the decoder. GPT-3 also adapts this method while returning the final answer [7].

To give an easy explanation, we will give an example. Suppose we want to translate Bahasa Indonesia to English (we can use any other language). Pretend that we already have an encoder-decoder for this.

Now, let our sentence be $C = \text{"Saya mau makan nasi goreng"}$. The first step is to determine the beam width w . The beam width tells us how many words do we consider each time. The normal greedy search is a beam search with $w = 1$. For this illustration, let $w = 4$, similar to the one GPT-3 uses. After this, we input the sentence to our model and let the model find four (the value of w) most common first words, instead of just one. Suppose the four most common words are "I", "Me", "We", and "My". For the next step, we calculate the most common bigram with the first word being the top 4 most common words. We can use Bayes Theorem to see that, for instance,

$$P(\text{I want} | C) = P(\text{I} | C)P(\text{I want} | C, \text{I}).$$

We do this for all words in English dictionary (we calculate the probability $4|V|$). After that, we collate the top 4 most probable 2-gram, and suppose it is "I want", "Me want", "I like", and "My want". We then repeat the process until all words are ended. Finally, the word with the highest probability will be chosen. Higher w will give a better translation, but it also costs more time and computation complexity.

To improve beam search, we can include length normalization. This is useful to encourage lengthy sentences. For a translated sentence Y , define $lp(Y) = \frac{(5+|Y|)^\alpha}{6^\alpha}$. Then, normalize the score to get the final score $s(Y, X) = \frac{\log P(Y|X)}{lp(Y)}$. There are

also other source of normalization, but the GPT-3 paper only states on this kind of normalization. GPT-3 uses $\alpha = 0.6$ in their algorithm.

B How Machine Grade Essays: BLEU Score

For simple word translation or multiple-choice questions, it is easy to check how wrong or correct is the answer, as the possible answers are few. For continuous variables, such as house prices, we usually model the error with the simple absolute error. However, for text generations or translations, there is no obvious way to see how close (or far) a generated sentence is from the intended answer. Even worse, there is a possibility that a sentence has two perfectly valid translations ("He is very rich" and "He is wealthy" both tell us the same thing). However, there is a clear difference between a good translation and the bad one (the translation done by a machine is worse than the one done by a human, for example).

Hence, we will examine BLEU (Bilingual Evaluation Understudy) score [12], which is a common method to check the quality of outputted sentences. Before we start, we need a reference for human translations. Usually, this data consists of human-written texts that are not yet seen by the model.

First, we will calculate the precision of each n -gram. We define $BLEU_n$ as the fraction of n -grams that appears in the reference texts. However, for each n -gram, we limit how many appear on one reference text. This is done to prevent cases where "The the the the the" is rated highly because "the" is on the intended sentence "The cat is sleeping quietly". Under this rule, the $BLEU_1$ score is only $\frac{1}{5}$ instead of 1. The lower n in $BLEU_n$ corresponds to how "correct and meaningful" the sentence is, while the high n corresponds to well-formed sentences. Finally, the $BLUE$ score is just the mean of these $BLEU_n$ scores. However, this penalizes long sentences, because $BLEU_n$ is lower for larger n . For instance, if the reference is "The cat is sleeping quietly" and the guess was "The cat", $BLEU$ will be 1 but we failed to convey the entire meaning of the sentence. Hence, we introduce a BP variable, or Brevity Penalty. It is set to 1 if the sentence is longer than the target, and $e^{1-|R|/|C|}$ if it is shorter, where $|R|$ is the answer length and $|C|$ is the candidate length. Then, our final BLEU is equal to $BP \times meanOf BLEU_n$.

C Sampling with Temperature

If we try the GPT sandbox, we come across a tab called temperature. This small section explains what does it affect. In summary, when choosing an output, the probability of the i -th word w_i is chosen is $P(w_i) = \frac{p_i}{\sum p_j} = p_i$ because probabilities add to 1. Now, we define a new function, $P_\tau(w_i) = \frac{p_i^{1/\tau}}{\sum p_j^{1/\tau}}$. We consider two cases, when $\tau \rightarrow 0$ and $\tau \rightarrow \infty$. In the former case, $P_\tau(w_i) \rightarrow \frac{1}{|N|}$, where $|N|$ is the number of possible words. In the latter case, $P_\tau(w_i) \rightarrow 1$ if p_i is the most probable word, and 0 otherwise.

Setting the temperature gives us the choice of our model. If we want our output to be more linguistically diverse, we set τ to be small (more words are probable to be outputted, but it might be wrong). Conversely, if we want to make our model grammatically correct, set τ to be large to ensure the picked words are the best ones.

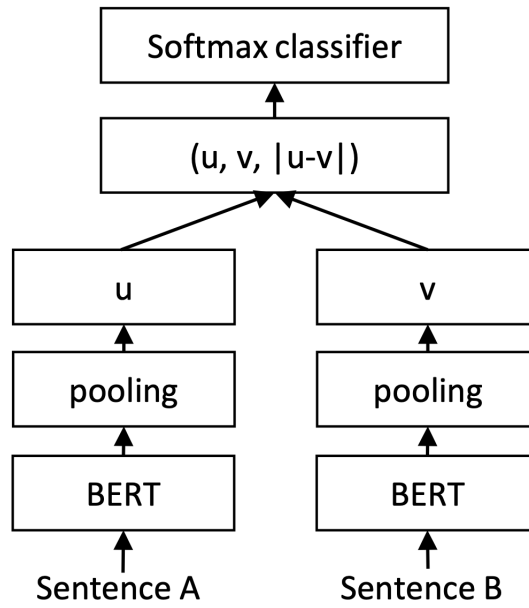
D BERT, Alternative to GPT

In this section, we delve a bit into BERT [5], a model similar in idea to GPT made by Google. We will mainly focus on what makes it different from GPT. Unlike GPT, BERT is open source (we can access its weights).

Like GPT, it is also based on the Transformer model. However, instead of only using the decoders, now we only use the encoders for the model. This is because BERT's purpose is to generate a language model, not to make a prediction.

Another differentiating feature of BERT is unlike GPT, it is bidirectional, hence improved performance on tasks needing to look back at the sentence (reading comprehension). However, in terms of size, BERT is much smaller than GPT (magnitude of 400). Also, because BERT only outputs the embedding of a word, we need to make a model that uses this embedding to output a prediction.

One problem with BERT is the speed of some tasks, especially those related to similarity is very slow. For instance, finding the two most similar sentences in a dataset of n sentences takes $O(n^2)$ time. This happens because to check similarity, both sentences must be processed one at a time. Also, the embedding from the original BERT does not have a semantical meaning. Hence, Sentence-BERT [14] is created to solve the issue. The idea is now we want to precompute embeddings that are semantically meaningful for each sentence and use them to measure using cosine similarity which is fast to compute.

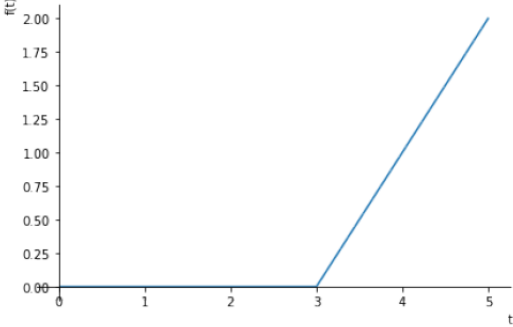


In other words, we create a Siamese network. A siamese network is two identical networks (same weights) that want to learn a similar distance between two inputs. Our input for the softmax classifier will be the embedding vector u , v , and their difference. The researchers trained this network with texts and tasks whether two sentences are similar, opposite or neither. Then, to check similarity at inference time, we detach the classifier and combine u and v to find its cosine similarity. This is faster and more relatable because the similarity score is fast to compute and because of our pretraining, u and v have a good semantical meaning.

E Text to Code Samples

Topic	Analytical Geometry and Polar Coordinates
Original question	Find the center and radius of the circle described in the given equation. $x^2 + 2x + y^2 = 4$
Codex input	Using Sympy, find the radius and center of $x^2 + 2x + y^2 - 4 = 0$ by finding the radius and center attributes of Circle(Eq($x^2 + 2x + y^2 - 4$)).
Codex output	<pre> from sympy import * x, y = symbols('x y') eq = x**2 + 2*x + y**2 - 4 circle = Circle(eq) print(circle.radius) print(circle.center) </pre>
Solution	Center: (-1,0); Radius: $\sqrt{5}$

Topic	Central Limit Theorem, Probability
Original question	To head the newly formed US Dept. of Statistics, suppose that 50% of the population supports Erika, 20% supports Ruthi, and the rest are split between Peter, Jon and Jerry. A poll asks 400 random people who they support. Use the central limit theorem to estimate the probability that less than 25% of those polled prefer Peter, Jon or Jerry?
Codex input	Suppose that 30% of the population supports Peter. A poll asks 400 random people who they support. Write a program that uses the central limit theorem to compute the probability that less than 25% of those polled support Peter.
Codex output	<pre> import random def poll(n, p): count = 0 for i in range(n): if random.random() < p: count += 1 return count def main(): peter_support = 0.3 n = 400 p = 0.25 poll_size = 1000 count = 0 for i in range(poll_size): if poll(n, peter_support) < p * n: count += 1 print(count / poll_size) if __name__ == "__main__": main() </pre>
Solution	0.0145

Topic	Periodic and Piecewise Continuous Input Functions
Original question	Find the inverse Laplace transform $f(t)$. Then sketch the graph of f . $F(s) = \frac{e^{-3s}}{s^2}$.
Codex input	Use the inverse_laplace_transform to find original function from $e^{(-3*s)}/s^{**2}$. Graph the solution.
Codex output	<pre> from sympy import * from sympy.abc import s, t from sympy.integrals import inverse_laplace_transform from sympy.plotting import plot f = inverse_laplace_transform(exp(-3*s)/s**2, s, t) print(f) plot(f, (t, 0, 5)) </pre>
Solution	$-(\log(\exp(-t)) + 3) * \text{Heaviside}(t - 3)$ 

F Algorithms

Algorithm 1 Forward Propagation

Require: Network depth l ,

Require: Weight matrix and bias parameters W^i and b^i , $i = 1, 2, \dots, l$

Require: Input x and Output y .

Require: Activation function f .

$h^0 = x$

for $i = 1, 2, \dots, l$ **do**

$a^i = W^i h^{i-1} + b^i$

$h^i = f(a^i)$

end for

$\hat{y} = h^l$

$J = L(y, \hat{y})$

Algorithm 2 Backward Propagation. \odot denotes element-wise matrix multiplication

$g = \nabla_{\hat{y}} J$

for $i = l, l-1, \dots, 1$ **do**

$g = \nabla_{a^i} J = g \odot f'(a_i)$

$\nabla_{b^i} J = g$

$\nabla_{W^i} J = g(h^{i-1})^T$

$g = (W^i)^T g$

end for

G Materials

The materials used may be slightly different from the ones used in the paper, but the general topics covered should be very similar.

1. MIT Calculus: Fall 2010 18.01
2. MIT Differential Equations: Spring 2010 18.03
3. MIT Intro to Probability: Spring 2014 18.05
4. MIT Linear Algebra: Spring 2010 18.06
5. MIT Math for Computer Science: Fall 2010 6.042
6. Columbia Computational Linear Algebra: Syllabus COMS3251

References

- [1] Dzmitry Bahdanau, Kyunghyun Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ArXiv*, 1409, 09 2014.
- [2] Y. Bengio and Sofian Audry. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 19:713–22, 05 2008.
- [3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.
- [4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya

- Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.
- [5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.
- [6] Iddo Drori, Sunny Tran, Roman Wang, Newman Cheng, Kevin Liu, Leonard Tang, Elizabeth Ke, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. A neural network solves and generates mathematics problems by program synthesis: Calculus, differential equations, linear algebra, and more. *CoRR*, abs/2112.15594, 2021.
- [7] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. *CoRR*, abs/1702.01806, 2017.
- [8] Erol Gelenbe, Hakan Bakircioglu, and Taskin Kocak. Image processing with the random neural network (rnn). In *Proceedings of 13th International Conference on Digital Signal Processing*, volume 1, pages 243–248. IEEE, 1997.
- [9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [10] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.
- [11] Shubham Singh Paliwal, Vishwanath D, Rohit Rahul, Monika Sharma, and Lovekesh Vig. Tablenet: Deep learning model for end-to-end table detection and tabular data extraction from scanned document images. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 128–133, 2019.

- [12] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.
- [13] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.
- [14] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.
- [15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.
- [16] Adam Zsolt Wagner. Constructions in combinatorics via neural networks, 2021.
- [17] Jun Zhang and Kim-Fung Man. Time series prediction using rnn in multi-dimension embedding phase space. In *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, volume 2, pages 1868–1873. IEEE, 1998.