

METHODS AND SYSTEMS FOR PROMPTING TOKEN-EFFICIENT OUTPUT FROM LARGE LANGUAGE MODEL

CROSS-REFERENCE TO RELATED APPLICATION

- 5 **[1]** The present disclosure claims priority from U.S. provisional patent application no. 63/587,862, filed October 4, 2023, entitled "METHODS AND SYSTEMS FOR GENERATING TOKEN-EFFICIENT PROMPTS FOR LARGE LANGUAGE MODEL (LLM)", the entirety of which is hereby incorporated by reference.

10

FIELD

- [2]** The present disclosure relates to machine learning, and, more particularly, to generation of prompts to large language models (LLM), and, yet more particularly, to prompting an LLM to generate token-efficient output.

15

BACKGROUND

- [3]** A large language model (LLM) is a deep learning algorithm that can process natural language to summarize, translate, predict and generate text and other content. A LLM may be trained to learn billions of parameters in order to model how words relate to each other in a textual sequence. Inputs to a LLM may be referred to as prompts. A prompt is a natural language input that includes instructions to cause the LLM to generate a desired output.

20

SUMMARY

- 25 **[4]** A LLM generates output one token at a time, thus the larger the number of tokens required for the output, the greater the computing resources (e.g., processing time, processor power, communication bandwidth, etc.) consumed. Similarly, a prompt to the LLM is tokenized into tokens that the LLM processes, one token at a time. The larger the number of tokens the LLM has to process in

the prompt, the greater the amount of computing resources required. As well, there is typically a limit to the number of tokens that can be inputted to the LLM or outputted by the LLM.

[5] A LLM may be prompted to generate output that includes text in a constructed language (e.g., a non-human or artificial language such as a programming language). However, because tokens and tokenizers are typically designed for processing text in a natural human language, text in the constructed language may be tokenized in a way that does not make efficient use of tokens (e.g., requiring a large number of tokens to represent a single term). This can negatively impact the consumption of computing resources by the LLM.

[6] In various examples, the present disclosure provides a technical solution that enables a LLM to provide output in a more token-efficient manner, specifically using a token-efficient language. The token-efficient language, as described herein, is used to represent text in a manner such that the text can be tokenized (e.g., using a conventional tokenizer used for LLMs) into fewer tokens than an equivalent text in a different constructed language such as a programming language. This provides a technical advantage in that fewer computing resources (e.g., memory, processing power, computing time, etc.) are required by the LLM to generate the output. Additionally, because the output can be generated by the LLM using fewer tokens, the time between prompting the LLM and receiving the complete output may be reduced, thus reducing the overall latency of the system. As well, the risk of reaching the maximum number of output tokens before the complete output is provided may be reduced.

[7] In an example aspect, the present disclosure describes a computing system including a processing unit configured to execute computer-readable instructions to cause the system to: provide a task prompt to a large language model (LLM) for performing a natural language task and generating token-efficient output based on the task prompt; and receive output from the LLM based on the task prompt, the output including text related to the task, at least a portion of the text being in a token-efficient language; where the text in the token-efficient language is tokenized by a tokenizer of the LLM into fewer tokens

than would correspond to a tokenization of corresponding text in a different constructed language.

[8] In an example of the preceding example computing system, the processing unit may be further configured to execute computer-readable instructions to cause the computer system to: convert at least the portion of the text in the token-efficient language to corresponding text in the constructed language.

[9] In an example of the preceding example computing system, the processing unit may be further configured to execute computer-readable instructions to implement a language converter, where the language converter may perform the converting.

[10] In an example of the preceding example computing system, the language converter may be specific to the constructed language.

[11] In an example of some of the preceding examples of the computing system, converting at least the portion of the text in the token-efficient language to corresponding text in the constructed language may include replacing a placeholder string of text in the token-efficient language with a mapped string of text in the constructed language, where the placeholder string of text may be tokenized into fewer tokens than the mapped string of text.

[12] In an example of the preceding example computing system, a mapping between the placeholder string of text and the mapped string of text may be stored in a memory accessible by the processing unit.

[13] In an example of any of the preceding examples of the computing system, the processing unit may be further configured to execute computer-readable instructions to cause the computer system to: generate one or more training samples that each pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task; where at least one of the one or more training samples is provided to the LLM prior to providing the task prompt or is provided to the LLM in the task prompt.

[14] In an example of any of the preceding examples of the computing

system, the processing unit may be further configured to execute computer-readable instructions to cause the computer system to, prior to providing the task prompt: fine-tune the LLM to generate token-efficient output using training samples that each pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

[15] In an example of any of the preceding examples of the computing system, the processing unit may be further configured to execute computer-readable instructions to cause the computer system to, prior to providing the task prompt: provide a training prompt to the LLM for performing the same or different natural language task, the training prompt including at least one training sample that pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

[16] In an example of any of the preceding examples of the computing system, the task prompt may include at least one training sample that pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

[17] In an example of any of the preceding examples of the computing system, the processing unit may be further configured to execute computer-readable instructions to cause the computer system to provide the text related to the task, including at least the portion of text in the token-efficient language, as input to another software module.

[18] In another example aspect, the present disclosure describes a method including: providing a task prompt to a large language model (LLM) for performing a natural language task and generating token-efficient output based on the task prompt; and receiving output from the LLM based on the task prompt, the output including text related to the task, at least a portion of the text being in a token-efficient language; where the text in the token-efficient language is tokenized by a tokenizer of the LLM into fewer tokens than would correspond to a tokenization of corresponding text in a different constructed language.

[19] In an example of the preceding example method, the method may include: converting at least the portion of the text in the token-efficient language to corresponding text in the constructed language.

[20] In an example of the preceding example method, the converting
5 may be performed a language converter.

[21] In an example of the preceding example method, the language converter may be specific to the constructed language.

[22] In an example of some of the preceding examples of the method, converting at least the portion of the text in the token-efficient language to
10 corresponding text in the constructed language may include replacing a placeholder string of text in the token-efficient language with a mapped string of text in the constructed language, where the placeholder string of text may be tokenized into fewer tokens than the mapped string of text.

[23] In an example of the preceding example method, a mapping
15 between the placeholder string of text and the mapped string of text may be stored in a memory.

[24] In an example of any of the preceding examples of the method, the method may include: generating one or more training samples that each pairs a sample natural language task with a corresponding sample text in the token-
20 efficient language for performing the sample natural language task; where at least one of the one or more training samples is provided to the LLM prior to providing the task prompt or is provided to the LLM in the task prompt.

[25] In an example of any of the preceding examples of the method, the method may include, prior to providing the task prompt: fine-tuning the LLM to
25 generate token-efficient output using training samples that each pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

[26] In an example of any of the preceding examples of the method, the method may include, prior to providing the task prompt: providing a training
30 prompt to the LLM for performing the same or different natural language task, the training prompt including at least one training sample that pairs a sample

natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

[27] In an example of any of the preceding examples of the method, the task prompt may include at least one training sample that pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

[28] In an example of any of the preceding examples of the method, the method may include: providing the text related to the task, including at least the portion of text in the token-efficient language, as input to another software module.

[29] In another example aspect, the present disclosure describes a computer-readable medium storing instructions that, when executed by a processing unit of a computing system, cause the computing system to: provide a task prompt to a large language model (LLM) for performing a natural language task and generating token-efficient output based on the task prompt; and receive output from the LLM based on the task prompt, the output including text related to the task, at least a portion of the text being in a token-efficient language; where the text in the token-efficient language is tokenized by a tokenizer of the LLM into fewer tokens than would correspond to a tokenization of corresponding text in a different constructed language.

[30] In some examples, the computer-readable medium may store instructions that, when executed by the processor of the computing system, cause the computing system to perform any of the methods described above.

[31] In another example aspect, the present disclosure describes computer program comprising processor-executable instructions that, when executed by one or more processors of a computer, cause the computer system to carry out the operations of any of the methods described above.

[32] Accordingly, there is provided a method, a computer system, a non-transitory processor-readable medium, and a computer program as detailed in the claims that follow.

BRIEF DESCRIPTION OF THE DRAWINGS

[33] Reference will now be made, by way of example, to the accompanying drawings which show example embodiments of the present application, and in which:

5 **[34]** FIG. 1A is a block diagram of a simplified convolutional neural network, which may be used in examples of the present disclosure;

[35] FIG. 1B is a block diagram of a simplified transformer neural network, which may be used in examples of the present disclosure;

10 **[36]** FIG. 2 is a block diagram of an example computing system, which may be used to implement examples of the present disclosure;

[37] FIG. 3 is a block diagram illustrating example execution of some modules of the example computing system of FIG. 2; and

15 **[38]** FIG. 4 is a flowchart illustrating an example method for prompting a LLM to generate a token-efficient output, in accordance with examples of the present disclosure.

[39] Similar reference numerals may have been used in different figures to denote similar components.

DETAILED DESCRIPTION

20 **[40]** In various examples, the present disclosure describes methods and systems that help to improve computer efficiency in prompting and receiving output from a generative machine learning model, and in particular a large language model (LLM). Typically, input to a LLM, in the form of a prompt, is tokenized into tokens that the LLM processes, one token at a time. The larger
25 the number of tokens the LLM has to process in the prompt, the greater the amount of computing resources (e.g., processing time, processor power, memory, communication bandwidth, etc.) consumed. Output from the LLM is generated one token at a time, thus the larger the number of tokens required for the output, the greater the computing resources consumed. As it should be
30 understood by one skilled in the art, the number of tokens required for

prompting the LLM and required by the LLM to generate the output can be directly related to the amount of computing resources consumed.

[41] However, tokens and tokenizers are typically designed for processing text in a natural human language (e.g., spoken language), there text
5 in a constructed language (e.g., a programming language) may be tokenized in a way that does not make efficient use of tokens (e.g., requiring a large number of tokens to represent a single term). This means a larger number of tokens is required by the LLM to generate output that includes text in a constructed language, thus negatively impacting the consumption of computing resources by
10 the LLM. Examples of the present disclosure help to improve computer efficiency by using a token-efficient language to reduce the number of tokens used in prompting and receiving output from a LLM when the output requires text in a constructed language (e.g., a programming language), thus reducing the computing resources consumed.

15 **[42]** To assist in understanding the present disclosure, some concepts relevant to neural networks and machine learning (ML) are first discussed.

[43] Generally, a neural network comprises a number of computation units (sometimes referred to as "neurons"). Each neuron receives an input value and applies a function to the input to generate an output value. The function
20 typically includes a parameter (also referred to as a "weight") whose value is learned through the process of training. A plurality of neurons may be organized into a neural network layer (or simply "layer") and there may be multiple such layers in a neural network. The output of one layer may be provided as input to a subsequent layer. Thus, input to a neural network may be processed through a
25 succession of layers until an output of the neural network is generated by a final layer. This is a simplistic discussion of neural networks and there may be more complex neural network designs that include feedback connections, skip connections, and/or other such possible connections between neurons and/or layers, which need not be discussed in detail here.

30 **[44]** A deep neural network (DNN) is a type of neural network having multiple layers and/or a large number of neurons. The term DNN may encompass any neural network having multiple layers, including convolutional

neural networks (CNNs), recurrent neural networks (RNNs), and multilayer perceptrons (MLPs), among others.

[45] DNNs are often used as ML-based models for modeling complex behaviors (e.g., human language, image recognition, object classification, etc.) in order to improve accuracy of outputs (e.g., more accurate predictions) such as, for example, as compared with models with fewer layers. In the present disclosure, the term "ML-based model" or more simply "ML model" may be understood to refer to a DNN. Training a ML model refers to a process of learning the values of the parameters (or weights) of the neurons in the layers such that the ML model is able to model the target behavior to a desired degree of accuracy. Training typically requires the use of a training dataset, which is a set of data that is relevant to the target behavior of the ML model. For example, to train a ML model that is intended to model human language (also referred to as a language model), the training dataset may be a collection of text documents, referred to as a text corpus (or simply referred to as a corpus). The corpus may represent a language domain (e.g., a single language), a subject domain (e.g., scientific papers), and/or may encompass another domain or domains, be they larger or smaller than a single language or subject domain. For example, a relatively large, multilingual and non-subject-specific corpus may be created by extracting text from online webpages and/or publicly available social media posts. In another example, to train a ML model that is intended to classify images, the training dataset may be a collection of images. Training data may be annotated with ground truth labels (e.g. each data entry in the training dataset may be paired with a label), or may be unlabeled.

[46] Training a ML model generally involves inputting into an ML model (e.g. an untrained ML model) training data to be processed by the ML model, processing the training data using the ML model, collecting the output generated by the ML model (e.g. based on the inputted training data), and comparing the output to a desired set of target values. If the training data is labeled, the desired target values may be, e.g., the ground truth labels of the training data. If the training data is unlabeled, the desired target value may be a reconstructed (or otherwise processed) version of the corresponding ML model input (e.g., in

the case of an autoencoder), or may be a measure of some target observable effect on the environment (e.g., in the case of a reinforcement learning agent). The parameters of the ML model are updated based on a difference between the generated output value and the desired target value. For example, if the value
5 outputted by the ML model is excessively high, the parameters may be adjusted so as to lower the output value in future training iterations. An objective function is a way to quantitatively represent how close the output value is to the target value. An objective function represents a quantity (or one or more quantities) to be optimized (e.g., minimize a loss or maximize a reward) in order to bring the
10 output value as close to the target value as possible. The goal of training the ML model typically is to minimize a loss function or maximize a reward function.

[47] The training data may be a subset of a larger data set. For example, a data set may be split into three mutually exclusive subsets: a training set, a validation (or cross-validation) set, and a testing set. The three
15 subsets of data may be used sequentially during ML model training. For example, the training set may be first used to train one or more ML models, each ML model, e.g., having a particular architecture, having a particular training procedure, being describable by a set of model hyperparameters, and/or otherwise being varied from the other of the one or more ML models. The
20 validation (or cross-validation) set may then be used as input data into the trained ML models to, e.g., measure the performance of the trained ML models and/or compare performance between them. Where hyperparameters are used, a new set of hyperparameters may be determined based on the measured performance of one or more of the trained ML models, and the first step of
25 training (i.e., with the training set) may begin again on a different ML model described by the new set of determined hyperparameters. In this way, these steps may be repeated to produce a more performant trained ML model. Once such a trained ML model is obtained (e.g., after the hyperparameters have been adjusted to achieve a desired level of performance), a third step of collecting the
30 output generated by the trained ML model applied to the third subset (the testing set) may begin. The output generated from the testing set may be compared with the corresponding desired target values to give a final assessment of the trained ML model's accuracy. Other segmentations of the

larger data set and/or schemes for using the segments for training one or more ML models are possible.

[48] Backpropagation is an algorithm for training a ML model.

Backpropagation is used to adjust (also referred to as update) the value of the parameters in the ML model, with the goal of optimizing the objective function. For example, a defined loss function is calculated by forward propagation of an input to obtain an output of the ML model and comparison of the output value with the target value. Backpropagation calculates a gradient of the loss function with respect to the parameters of the ML model, and a gradient algorithm (e.g., gradient descent) is used to update (i.e., "learn") the parameters to reduce the loss function. Backpropagation is performed iteratively, so that the loss function is converged or minimized. Other techniques for learning the parameters of the ML model may be used. The process of updating (or learning) the parameters over many iterations is referred to as training. Training may be carried out iteratively until a convergence condition is met (e.g., a predefined maximum number of iterations has been performed, or the value outputted by the ML model is sufficiently converged with the desired target value), after which the ML model is considered to be sufficiently trained. The values of the learned parameters may then be fixed and the ML model may be deployed to generate output in real-world applications (also referred to as "inference").

[49] In some examples, a trained ML model may be fine-tuned, meaning that the values of the learned parameters may be adjusted slightly in order for the ML model to better model a specific task. Fine-tuning of a ML model typically involves further training the ML model on a number of data samples (which may be smaller in number/cardinality than those used to train the model initially) that closely target the specific task. For example, a ML model for generating natural language that has been trained generically on publically-available text corpuses may be, e.g., fine-tuned by further training using the complete works of Shakespeare as training data samples (e.g., where the intended use of the ML model is generating a scene of a play or other textual content in the style of Shakespeare).

[50] FIG. 1A is a simplified diagram of an example CNN 10, which is an example of a DNN that is commonly used for image processing tasks such as image classification, image analysis, object segmentation, etc. An input to the CNN 10 may be a 2D RGB image 12.

5 **[51]** The CNN 10 includes a plurality of layers that process the image 12 in order to generate an output, such as a predicted classification or predicted label for the image 12. For simplicity, only a few layers of the CNN 10 are illustrated including at least one convolutional layer 14. The convolutional layer 14 performs convolution processing, which may involve computing a dot product
10 between the input to the convolutional layer 14 and a convolution kernel. A convolutional kernel is typically a 2D matrix of learned parameters that is applied to the input in order to extract image features. Different convolutional kernels may be applied to extract different image information, such as shape information, color information, etc.

15 **[52]** The output of the convolution layer 14 is a set of feature maps 16 (sometimes referred to as activation maps). Each feature map 16 generally has smaller width and height than the image 12. The set of feature maps 16 encode image features that may be processed by subsequent layers of the CNN 10, depending on the design and intended task for the CNN 10. In this example, a
20 fully connected layer 18 processes the set of feature maps 16 in order to perform a classification of the image, based on the features encoded in the set of feature maps 16. The fully connected layer 18 contains learned parameters that, when applied to the set of feature maps 16, outputs a set of probabilities representing the likelihood that the image 12 belongs to each of a defined set of
25 possible classes. The class having the highest probability may then be outputted as the predicted classification for the image 12.

[53] In general, a CNN may have different numbers and different types of layers, such as multiple convolution layers, max-pooling layers and/or a fully connected layer, among others. The parameters of the CNN may be learned
30 through training, using data having ground truth labels specific to the desired task (e.g., class labels if the CNN is being trained for a classification task, pixel

masks if the CNN is being trained for a segmentation task, text annotations if the CNN is being trained for a captioning task, etc.), as discussed above.

[54] Some concepts in ML-based language models are now discussed. It may be noted that, while the term "language model" has been commonly used to refer to a ML-based language model, there could exist non-ML language models. In the present disclosure, the term "language model" may be used as shorthand for ML-based language model (i.e., a language model that is implemented using a neural network or other ML architecture), unless stated otherwise. For example, unless stated otherwise, "language model" encompasses LLMs.

[55] A language model may use a neural network (typically a DNN) to perform natural language processing (NLP) tasks such as language translation, image captioning, grammatical error correction, and language generation, among others. A language model may be trained to model how words relate to each other in a textual sequence, based on probabilities. A language model may contain hundreds of thousands of learned parameters or in the case of a large language model (LLM) may contain millions or billions of learned parameters or more.

[56] In recent years, there has been interest in a type of neural network architecture, referred to as a transformer, for use as language models. For example, the Bidirectional Encoder Representations from Transformers (BERT) model, the Transformer-XL model and the Generative Pre-trained Transformer (GPT) models are types of transformers. A transformer is a type of neural network architecture that uses self-attention mechanisms in order to generate predicted output based on input data that has some sequential meaning (i.e., the order of the input data is meaningful, which is the case for most text input). Although transformer-based language models are described herein, it should be understood that the present disclosure may be applicable to any ML-based language model, including language models based on other neural network architectures such as recurrent neural network (RNN)-based language models.

[57] FIG. 1B is a simplified diagram of an example transformer 50, and a simplified discussion of its operation is now provided. The transformer 50

includes an encoder 52 (which may comprise one or more encoder layers/blocks connected in series) and a decoder 54 (which may comprise one or more decoder layers/blocks connected in series). Generally, the encoder 52 and the decoder 54 each include a plurality of neural network layers, at least one of which may be a self-attention layer. The parameters of the neural network layers may be referred to as the parameters of the language model.

[58] The transformer 50 may be trained on a text corpus that is labelled (e.g., annotated to indicate verbs, nouns, etc.) or unlabelled. LLMs may be trained on a large unlabelled corpus. Some LLMs may be trained on a large multi-language, multi-domain corpus, to enable the model to be versatile at a variety of language-based tasks such as generative tasks (e.g., generating human-like natural language responses to natural language input).

[59] An example of how the transformer 50 may process textual input data is now described. Input to a language model (whether transformer-based or otherwise) typically is in the form of natural language as may be parsed into tokens. It should be appreciated that the term "token" in the context of language models and NLP has a different meaning from the use of the same term in other contexts such as data security. Tokenization, in the context of language models and NLP, refers to the process of parsing textual input (e.g., a character, a word, a phrase, a sentence, a paragraph, etc.) into a sequence of shorter segments that are converted to numerical representations referred to as tokens (or "compute tokens"). Typically, a token may be an integer that corresponds to the index of a text segment (e.g., a word) in a vocabulary dataset. Often, the vocabulary dataset is arranged by frequency of use. Commonly occurring text, such as punctuation, may have a lower vocabulary index in the dataset and thus be represented by a token having a smaller integer value than less commonly occurring text. Tokens frequently correspond to words, with or without whitespace appended. In some examples, a token may correspond to a portion of a word. For example, the word "lower" may be represented by a token for [low] and a second token for [er]. In another example, the text sequence "Come here, look!" may be parsed into the segments [Come], [here], [,], [look] and [!], each of which may be represented

by a respective numerical token. In addition to tokens that are parsed from the textual sequence (e.g., tokens that correspond to words and punctuation), there may also be special tokens to encode non-textual information. For example, a [CLASS] token may be a special token that corresponds to a classification of the textual sequence (e.g., may classify the textual sequence as a poem, a list, a paragraph, etc.), a [EOT] token may be another special token that indicates the end of the textual sequence, other tokens may provide formatting information, etc.

[60] In FIG. 1B, a short sequence of tokens 56 corresponding to the text sequence "Come here, look!" is illustrated as input to the transformer 50. Tokenization of the text sequence into the tokens 56 may be performed by some pre-processing tokenization module such as, for example, a byte pair encoding tokenizer (the "pre" referring to the tokenization occurring prior to the processing of the tokenized input by the LLM), which is not shown in FIG. 1B for simplicity. In general, the token sequence that is inputted to the transformer 50 may be of any length up to a maximum length defined based on the dimensions of the transformer 50 (e.g., such a limit may be 2048 tokens in some LLMs). Each token 56 in the token sequence is converted into an embedding vector 60 (also referred to simply as an embedding). An embedding 60 is a learned numerical representation (such as, for example, a vector) of a token that captures some semantic meaning of the text segment represented by the token 56. The embedding 60 represents the text segment corresponding to the token 56 in a way such that embeddings corresponding to semantically-related text are closer to each other in a vector space than embeddings corresponding to semantically-unrelated text. For example, assuming that the words "look", "see", and "cake" each correspond to, respectively, a "look" token, a "see" token, and a "cake" token when tokenized, the embedding 60 corresponding to the "look" token will be closer to another embedding corresponding to the "see" token in the vector space, as compared to the distance between the embedding 60 corresponding to the "look" token and another embedding corresponding to the "cake" token. The vector space may be defined by the dimensions and values of the embedding vectors. Various techniques may be used to convert a token 56 to an embedding 60. For example, another trained ML model may be used to

convert the token 56 into an embedding 60. In particular, another trained ML model may be used to convert the token 56 into an embedding 60 in a way that encodes additional information into the embedding 60 (e.g., a trained ML model may encode positional information about the position of the token 56 in the text sequence into the embedding 60). In some examples, the numerical value of the token 56 may be used to look up the corresponding embedding in an embedding matrix 58 (which may be learned during training of the transformer 50).

[61] The generated embeddings 60 are input into the encoder 52. The encoder 52 serves to encode the embeddings 60 into feature vectors 62 that represent the latent features of the embeddings 60. The encoder 52 may encode positional information (i.e., information about the sequence of the input) in the feature vectors 62. The feature vectors 62 may have very high dimensionality (e.g., on the order of thousands or tens of thousands), with each element in a feature vector 62 corresponding to a respective feature. The numerical weight of each element in a feature vector 62 represents the importance of the corresponding feature. The space of all possible feature vectors 62 that can be generated by the encoder 52 may be referred to as the latent space or feature space.

[62] Conceptually, the decoder 54 is designed to map the features represented by the feature vectors 62 into meaningful output, which may depend on the task that was assigned to the transformer 50. For example, if the transformer 50 is used for a translation task, the decoder 54 may map the feature vectors 62 into text output in a target language different from the language of the original tokens 56. Generally, in a generative language model, the decoder 54 serves to decode the feature vectors 62 into a sequence of tokens. The decoder 54 may generate output tokens 64 one by one. Each output token 64 may be fed back as input to the decoder 54 in order to generate the next output token 64. By feeding back the generated output and applying self-attention, the decoder 54 is able to generate a sequence of output tokens 64 that has sequential meaning (e.g., the resulting output text sequence is understandable as a sentence and obeys grammatical rules). The decoder 54 may generate output tokens 64 until a special [EOT] token (indicating the end of

the text) is generated. The resulting sequence of output tokens 64 may then be converted to a text sequence in post-processing. For example, each output token 64 may be an integer number that corresponds to a vocabulary index. By looking up the text segment using the vocabulary index, the text segment corresponding to each output token 64 can be retrieved, the text segments can be concatenated together and the final output text sequence (in this example, "Viens ici, regarde!") can be obtained.

[63] Although a general transformer architecture for a language model and its theory of operation have been described above, this is not intended to be limiting. Existing language models include language models that are based only on the encoder of the transformer or only on the decoder of the transformer. An encoder-only language model encodes the input text sequence into feature vectors that can then be further processed by a task-specific layer (e.g., a classification layer). BERT is an example of a language model that may be considered to be an encoder-only language model. A decoder-only language model accepts embeddings as input and may use auto-regression to generate an output text sequence. Transformer-XL and GPT-type models may be language models that are considered to be decoder-only language models.

[64] Because GPT-type language models tend to have a large number of parameters, these language models may be considered LLMs. An example GPT-type LLM is GPT-3. GPT-3 is a type of GPT language model that has been trained (in an unsupervised manner) on a large corpus derived from documents available to the public online. GPT-3 has a very large number of learned parameters (on the order of hundreds of billions), is able to accept a large number of tokens as input (e.g., up to 2048 input tokens), and is able to generate a large number of tokens as output (e.g., up to 2048 tokens). GPT-3 has been trained as a generative model, meaning that it can process input text sequences to predictively generate a meaningful output text sequence. ChatGPT is built on top of a GPT-type LLM, and has been fine-tuned with training datasets based on text-based chats (e.g., chatbot conversations). ChatGPT is designed for processing natural language, receiving chat-like inputs and generating chat-like outputs.

[65] A computing system may access a remote language model (e.g., a cloud-based language model), such as ChatGPT or GPT-3, via a software interface (e.g., an application programming interface (API)). Additionally or alternatively, such a remote language model may be accessed via a network
5 such as, for example, the Internet. In some implementations such as, for example, potentially in the case of a cloud-based language model, a remote language model may be hosted by a computer system as may include a plurality of cooperating (e.g., cooperating via a network) computer systems such as may be in, for example, a distributed arrangement. Notably, a remote language
10 model may employ a plurality of processors (e.g., hardware processors such as, for example, processors of cooperating computer systems). Indeed, processing of inputs by an LLM may be computationally expensive/may involve a large number of operations (e.g., many instructions may be executed/large data structures may be accessed from memory) and providing output in a required
15 timeframe (e.g., real-time or near real-time) may require the use of a plurality of processors/cooperating computing devices as discussed above.

[66] Inputs to an LLM may be referred to as a prompt, which is a natural language input that includes instructions to the LLM to generate a desired output. A computing system may generate a prompt that is provided as input to
20 the LLM via its API. As described above, the prompt may optionally be processed or pre-processed into a token sequence prior to being provided as input to the LLM via its API. A prompt can include one or more examples of the desired output, which provides the LLM with additional information to enable the LLM to better generate output according to the desired output. Additionally or
25 alternatively, the examples included in a prompt may provide inputs (e.g., example inputs) corresponding to/as may be expected to result in the desired outputs provided. A one-shot prompt refers to a prompt that includes one example, and a few-shot prompt refers to a prompt that includes multiple examples. A prompt that includes no examples may be referred to as a zero-
30 shot prompt.

[67] FIG. 2 illustrates an example computing system 200, which may be used to implement examples of the present disclosure. For example, the

computing system 200 may be used to generate a prompt to an LLM to cause the LLM to generate output that includes text in a token-efficient language as disclosed herein. Additionally or alternatively, one or more instances of the example computing system 200 may be employed to execute the LLM. For
5 example, a plurality of instances of the example computing system 200 may cooperate to provide output using an LLM in manners as discussed above.

[68] The example computing system 200 includes at least one processing unit and at least one physical memory 204. The processing unit may be a hardware processor 202 (simply referred to as processor 202). The
10 processor 202 may be, for example, a central processing unit, a microprocessor, a digital signal processor, an application-specific integrated circuit (ASIC), a field-programmable gate array (FPGA), a dedicated logic circuitry, a dedicated artificial intelligence processor unit, a graphics processing unit (GPU), a tensor processing unit (TPU), a neural processing unit (NPU), a hardware accelerator,
15 or combinations thereof. The memory 204 may include a volatile or non-volatile memory (e.g., a flash memory, a random access memory (RAM), and/or a read-only memory (ROM)). The memory 204 may store instructions for execution by the processor 202, to cause the computing system 200 to carry out examples of the methods, functionalities, systems and modules disclosed herein.

20 **[69]** The computing system 200 may also include at least one network interface 206 for wired and/or wireless communications with an external system and/or network (e.g., an intranet, the Internet, a P2P network, a WAN and/or a LAN). The network interface 206 may enable the computing system 200 to carry out communications (e.g., wireless communications) with systems external to
25 the computing system 200, such as a LLM residing on a remote system.

[70] The computing system 200 may optionally include at least one input/output (I/O) interface 208, which may interface with optional input device(s) 210 and/or optional output device(s) 212. Input device(s) 210 may include, for example, buttons, a microphone, a touchscreen, a keyboard, etc.
30 Output device(s) 212 may include, for example, a display, a speaker, etc. In this example, optional input device(s) 210 and optional output device(s) 212 are shown external to the computing system 200. In other examples, one or more of

the input device(s) 210 and/or output device(s) 212 may be an internal component of the computing system 200.

[71] A computing system, such as the computing system 200 of FIG. 2, may access a remote system (e.g., a cloud-based system) to communicate with a remote language model or LLM hosted on the remote system such as, for example, using an application programming interface (API) call. The API call may include an API key to enable the computing system to be identified by the remote system. The API call may also include an identification of the language model or LLM to be accessed and/or parameters for adjusting outputs generated by the language model or LLM, such as, for example, one or more of a temperature parameter (which may control the amount of randomness or "creativity" of the generated output) (and/or, more generally some form of random seed as serves to introduce variability or variety into the output of the LLM), a minimum length of the output (e.g., a minimum of 10 tokens) and/or a maximum length of the output (e.g., a maximum of 1000 tokens), a frequency penalty parameter (e.g., a parameter which may lower the likelihood of subsequently outputting a word based on the number of times that word has already been output), a "best of" parameter (e.g., a parameter to control the number of times the model will use to generate output after being instructed to, e.g., produce several outputs based on slightly varied inputs). The prompt generated by the computing system is provided to the language model or LLM and the output (e.g., token sequence) generated by the language model or LLM is communicated back to the computing system. In other examples, the prompt may be provided directly to the language model or LLM without requiring an API call. For example, the prompt could be sent to a remote LLM via a network such as, for example, as or in message (e.g., in a payload of a message).

[72] In the example of FIG. 2, the computing system 200 may store in the memory 204 computer-executable instructions, which may be executed by a processing unit such as the processor 202, to implement one or more embodiments disclosed herein. For example, the memory 204 may store instructions for implementing a language converter 250 and a prompt generator 260, as discussed further below. The memory 204 may also store data such as

one or more datasets 270, which may include training samples for fine-tuning or training a LLM.

[73] In some examples, the computing system 200 may be a server of an online platform that provides the language converter 250 and/or prompt generator 260 as a web-based or cloud-based service that may be accessible by a user device (e.g., via communications over a wireless network). Other such variations may be possible without departing from the subject matter of the present application.

[74] In various examples, the present disclosure describes systems and methods for using a LLM to generate a token-efficient output. At least a portion of the text in the token-efficient output is in a token-efficient language. Notably, the text that is in the token-efficient language can be represented by tokens (e.g., when tokenized by a conventional tokenizer of the LLM) using fewer tokens than a corresponding text (with the same meaning) in a different language such as another constructed language. In the present disclosure, a constructed language (which may also be referred to as a non-natural language or artificial language) refers to a language that is not a language used for normal human communication (such as a spoken language). Examples of constructed languages include programming languages, such as JSON, Markdown, XML, etc. When text in a constructed language (e.g., a block of code in a programming language) is represented in the token-efficient language disclosed herein, the text in the token-efficient language will be tokenized into fewer tokens compared to the corresponding text in the constructed language.

[75] Tokenization of a constructed language such as a programming language can be inefficient. This is because conventional tokenizers designed for LLMs are designed to tokenize natural language (e.g., human language). A programming language is an example of a constructed language that may not be token-efficient. In particular, most programming languages have whitespace (e.g., spaces, tabs, newline characters, etc.), special characters (e.g., braces, brackets, periods, etc.) and variable names (e.g., CamelCase names, Hungarian notation, etc.) that do not efficiently translate to tokens in most tokenization schemes.

[76] A LLM processes one token at a time as input, and generates output one token at a time. When a LLM is prompted to perform a task that requires output in a constructed language (e.g., programming language) that the LLM and/or tokenizer is not optimized for, the LLM requires a large number of tokens to generate that output. For example, a LLM and its tokenizer may be mostly designed/trained on a natural language (e.g., human language) corpus. Outputs in a constructed language may include text strings that are unexpected in natural language and thus are likely to be tokenized in an inefficient way. For example, programming languages frequently use terms that are uncommon in natural language. Output from an LLM that includes such terms is likely to require a larger number of tokens to represent (e.g., requiring individual tokens to represent each letter in the uncommon term) and thus a drain on computing resources. Additionally, there is typically a maximum limit on the number of tokens that can be used. Output from the LLM that does not use tokens efficiently will more quickly reach that maximum limit. However, the disclosed solution provides a token-efficient language that can be used to represent the same output using fewer tokens, thus providing significant savings in computing resources and helping to avoid reaching the maximum limit of output tokens.

[77] In cases where output from the LLM is further processed by another ML model, the token-efficient output from the LLM can be provided as input to the other ML model. Since the token-efficient output is generally shorter (in terms of tokens) than the same output in the constructed language, this enables the other ML model to be designed for smaller inputs, which can result in a smaller ML model (and thus requiring fewer computing resources to train and deploy).

[78] In various examples, this disclosure provides a manner of constructing a token-efficient language that can be translated to a target constructed language (e.g., target programming language) such that, when the LLM generates output using the token-efficient language, fewer tokens are generated. The output in the token-efficient language may then be automatically converted to the target constructed language. In this way, the LLM may be used to assist in generating text in the target constructed language, while reducing

the consumption of computing resources by the LLM.

[79] Techniques that may be used to construct the token-efficient language based on a target constructed language include: removal of whitespace; strategic use of whitespace for common tokens with "built-in" whitespace (e.g., " name" with a leading whitespace character may be represented using a single token); removal of brackets or parentheses; combining multiple assignment commands into one command; and/or mapping unusual variable names to commonly tokenized text (e.g., replace "namespace" with "name"). The techniques used to construct the token-efficient language may be dependent on the target constructed language (e.g., dependent on the syntax used in the target constructed language).

[80] Consider, for example, the following code in JSON programming language:

```
15      {
        "instruction": "Fill the customer's first name and last name in the
form.",
        "task": "to fill out of a form",
        "commands": [
20          {
            "namespace": "forms",
            "cmd": "fill",
            "key": "first_name",
            "value": "John"
25          },
          {
            "namespace": "forms",
            "cmd": "fill",
            "key": "last_name",
            "value": "Doe"
30          }
        ]
      }
```

[81] The corresponding text in the token-efficient language may be:

forms fill: "first_name": "John"

forms fill: "last_name": "Doe"

5 **[82]** It may be appreciated that the text in the token-efficient language may be tokenized into fewer tokens than the original text in JSON. Further, the text in the token-efficient language uses fewer whitespace characters and avoids the use of brackets. In some examples, the text in the token-efficient language may replace an unusual string of text such as "first_name" (which may require
10 several tokens to represent) with a placeholder string of text such as "cat" (which may require only one token to represent), as discussed further, for greater token-efficiency (so as to be tokenized using even fewer tokens).

[83] FIG. 3 illustrates an example of how the language converter 250 and prompt generator 260 of FIG. 2 may operate to prompt a LLM to generate
15 output in the token-efficient language and convert the output to a target constructed language.

[84] The language converter 250 may be designed to convert a specific constructed language to the token-efficient language, and vice versa. There may be different forms of the token-efficient language that correspond to different
20 constructed languages. For example, the token-efficient language may use a particular format or schema for representing a particular function in JSON. However, a different constructed language (e.g., Markdown) may provide different functions that may be represented in the token-efficient language using another format or schema. In this example, the target constructed language is
25 JSON programming language.

[85] Instructions for a natural language task may be prompted to the prompt generator 260. For example, the computing system 200 may receive user input that includes a natural language task. In particular, the natural language task may require generation of text in a target constructed language.
30 For example, a natural language task that a user may input may be "generate code in JSON for an online form with first name and last name fields". The user

input may be provided in the context of a programming application or programming interface, which may implicitly or explicitly indicate to the prompt generator 260 that output in a target constructed language is desired (e.g., the programming application may be specific to JSON, or a user may explicitly select JSON as the programming language). In some examples, the prompt generator 260 may be specifically designed for prompting the LLM to generate output that includes text in the token-efficient language (e.g., the prompt generator 260 may be used in the context of a programming application only), and output from the LLM may be assumed to contain text in the token-efficient language. In other examples, the user input may be provided in a non-specific context, and the prompt generator 260 may parse the user input and identify one or more keywords such as "code" and "JSON" that indicate that output in a target constructed language (in this case, JSON programming language) is desired from the LLM.

[86] The prompt generator 260 may generate a task prompt that prompts the LLM to perform the natural language task and generate output in the token-efficient language instead of the target constructed language. The prompt from the prompt generator 260 may not need to identify the target constructed language to the LLM. The LLM may be simply prompted to generate output using the token-efficient language. In some examples, the prompt generator 260 may, prior to the task prompt, generate a training prompt that may help to fine-tune (e.g., using one-shot or few-shot training) the LLM to generate output in the token-efficient language.

[87] In some examples, the prompt generator 260 may include one or more training samples in the task prompt or training prompt. For example, given the target constructed language (which may be explicitly or implicitly indicated as described above), the prompt generator 260 may retrieve from the dataset 270 one or more training samples that each pair a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task. Alternatively or additionally, the prompt generator 260 may request the language converter 250 to generate one or more training samples. The request to the language converter 250 may identify the

target constructed language for the one or more training samples. Sample text in various constructed languages (e.g., various programming languages such as JSON, Markdown, XML, etc.) that are each labeled with a corresponding sample natural language task (e.g., a block of JSON code may be labeled with the task that the code is intended to perform) may be stored in memory. The language converter 250 may retrieve sample text in the target constructed language with its sample natural language task label from the memory, generate corresponding sample text in the token-efficient language, then pair the generated sample text in the token-efficient language with the sample natural language task label to generate a training sample (e.g., a training sample may pair a natural language task such as "make header red" with sample text in the token-efficient language such as "set color: text header "red""). The one or more training samples generated in this way may be provided to the prompt generator 260 and/or stored in the dataset 270. In this way, training samples may be generated as needed.

[88] As previously mentioned, the prompt generator 260 may insert one or more training samples (retrieved from the dataset 270 and/or received from the language converter 250) into the task prompt or training prompt to the LLM. Additionally or alternatively, the prompt generator 260 may insert one or more rules and/or syntax of the token-efficient language that the LLM is instructed to follow. For example, rules and/or syntax of the token-efficient language may be inserted into a training prompt in order to fine-tune the LLM to generate token-efficient output when subsequently prompted with a task prompt that includes a natural language task.

[89] The following is an example of a simplified training prompt that may be generated by the prompt generator 260 and provided to the LLM:

You are a helpful assistant to an online merchant.
You are able to make changes to their theme by executing 'token-lang'.
token-lang is a simple programming language.
Here is the full standard library documentation for token-lang:

All blocks have an id field.

theme move: <after_id> <id> # moves the selected block (with id) after the specified block(after_id)

5 theme add: <block-type> <new_id> # Adds a new block of type <block-type>. The <new_id> is equal to the next available id. It is always added at the end.

theme remove: <id> # Removes the block with the specified id

10 Some blocks can have color schemes, which dictate the colors within that block. A color scheme has the following:

- background: hex code color of section bkg
- text: Color of text (hex code)
- button: Color of button (hex code)

15 ...

Please help this merchant with any tasks they have. You are able to modify their theme by writing out token-lang.

- Only include the token-lang block at the end of your message.
- 20 - Sound natural and helpful

[90] This training prompt may fine-tune (or "prime") the LLM to generate output using the token-efficient language (referred to as "token-lang"), when subsequently asked to perform a natural language task such as modifying a theme of an online shop. A subsequent task prompt to the LLM may use a similar format or heading as the training prompt, but include a request such as:

25

Please help this merchant modify their theme using token-lang.

The text should be in blue.

[91] In some examples, the LLM may be prompted to generate training samples using a format as described above. For example, the LLM may be prompted (e.g., as described above) to generate sample text in a token-efficient language ("token-lang"). The sample text in the token-efficient language may be

30

extracted from the LLM-generated output and paired with the original sample natural language text to form a training sample, which may be stored in the dataset 270.

5 **[92]** The LLM may be fine-tuned using the training samples, so that the parameters in a final task-specific layer of the LLM are updated to enable the LLM to generate text in the token-efficient language when given a subsequent task prompt. In other examples, the LLM may be provided with training samples in the task prompt itself (in a technique referred to as in-context learning), which may not involve updating the parameters of the LLM.

10 **[93]** Whether the LLM is fine-tuned beforehand or whether training sample(s) are provided in the task prompt itself, the task prompt generated by the prompt generator 260 is provided to the LLM (e.g., via an API call) and the generated output may be received from the LLM. The generated output includes text in the token-efficient language, which may be converted to the target
15 constructed language by the language converter 250. In some examples, the generated output may use special characters to indicate the text in the token-efficient language, if the task prompt includes instructions to do so. This may enable the language converter 250 to automatically extract the text in the token-efficient language from the generated output (which may include other
20 text not in the token-efficient language).

[94] The language converter 250 may receive, from the prompt generator 260, identification of the target constructed language. This may enable the language converter 250 to apply the appropriate conversion rules (since different constructed languages may require different syntax) to convert
25 to text in the target constructed language. In some examples, the language converter 250 may not require identification of the target constructed language (e.g., the target constructed language may be assumed, or preset by default).

[95] The language converter 250 may apply predefined rules to parse the text in the token-efficient language and convert it to corresponding text in
30 the target constructed language. For example, the language converter 250 may insert special characters and whitespace characters as required by the target constructed language. In some examples, the language converter 250 may be a

deterministic model. In some examples, the language converter 250 may be implemented using a ML model that has been trained on the token-efficient language.

[96] In some examples, the language converter 250 may map a placeholder string of text to a mapped string of text. A placeholder string of text may be a string of text in the token-efficient language that is tokenized into fewer tokens than the mapped string of text in the target constructed language. For example, the string "cmd" in the target constructed language may be uncommon in natural human language and thus may be tokenized into the tokens [c], [m] and [d]. This would be an inefficient use of tokens. Since "cmd" may be expected to occur in the target constructed language, a placeholder string of text may be defined in the token-efficient language instead. The placeholder string of text may be a common word in a natural language, such as "cat", which may be tokenized into a single token [cat], thus requiring fewer tokens than "cmd" despite being the same string length. It should be noted that the placeholder string of text may even be a longer string than the mapped string of text, as long as the placeholder string of text is tokenized to fewer tokens than the mapped string of text. A mapping from "cat" to "cmd" may be stored in the language converter 250, such that when output from the LLM includes "cat" in the token-efficient language, the language converter 250 may automatically map this to "cmd" in the target constructed language. In some examples, such a mapping may be created by the prompt generator 260. For example, the prompt generator 260 may insert an instruction in the prompt such as "instead of 'cmd' use 'cat'", and may create a corresponding mapping between "cmd" and "cat", which mapping may be provided to the language converter 250. In some examples, the LLM may be provided with training samples (e.g., during fine-tuning) to enable the LLM to generate text in the token-efficient language using this mapping. For example, the LLM may be provided with a training sample that pairs a sample natural language task with sample text in the token-efficient language that always uses "cat" in place of "cmd".

[97] In some examples, the language converter 250 may validate that

the output from the LLM conforms to the token-efficient language (e.g., obeys certain grammar of the token-efficient language, does not use unusual or unexpected text strings, etc.) prior to converting to the target constructed language. If validation fails, the conversion may be aborted. The output that
5 could not be converted may nonetheless be stored in an output log and indicated as erroneous output, which may help improve future output from the LLM.

[98] The corresponding text in the target constructed language may be outputted to be further processed by other software applications of the computing system 200 and/or may be outputted to be displayed to a user. In
10 some examples, the text in the token-efficient language may be hidden from the user (i.e., not outputted for display). Further, it may not be necessary to provide any indication to the LLM that the token-efficient language is intended to be converted to a target constructed language.

[99] In some examples, the text in the token-efficient language may
15 also be provided as input to a subsequent software module (e.g., another ML model) prior to conversion to the target constructed language. In some examples, output from the LLM, including the text in the token-efficient language, may be stored as in an output log that the LLM may use as a context for processing subsequent prompts.

[100] It should be understood that the language converter 250 may be a
20 bidirectional language converter 250. Thus, the language converter 250 may convert from the token-efficient language to a target constructed language and also from the target constructed language to the token-efficient language. In some examples, the language converter 250 may comprise two sub-modules,
25 each being a unidirectional language converter (i.e., one sub-module converting from the token-efficient language to the target constructed language; another sub-module converting from the target constructed language to the token-efficient language).

[101] As previously mentioned, the language converter 250 may convert
30 text from the target constructed language to the token-efficient language, for example to generate training samples to fine-tune the LLM. For example, each sample text in the target constructed language may be annotated or labeled with

a corresponding sample natural language task. The language converter 250 may apply predefined rules to convert text from the target constructed language to the token-efficient language (such as removal of all brackets, removal of extra whitespace, etc.). Then the sample natural language task associated with the
5 sample text in the target constructed language may be paired with the converted text in the token-efficient language, to obtain a training sample that pairs a sample natural language task with a sample text in the token-efficient language.

[102] In some examples, the language converter 250 may be used to
10 generate mappings that may enable the use of placeholder strings in the token-efficient language, as discussed above. For example, the language converter 250 may store a dictionary of common text strings (which are likely to be represented using smaller and/or fewer tokens) that can be used as placeholders to replace uncommon text strings (e.g., CamelCase, compound words, etc.) in
15 the target constructed language. Such a dictionary may be used when converting from the target constructed language to the token-efficient language. For example, if the language converter 250, while parsing text in the target constructed language. As the language converter 250 parses text in the target constructed language, if the language converter 250 encounters a string of text
20 using CamelCase, the language converter 250 may automatically select a placeholder string of text from the dictionary of common text strings to replace the text in CamelCase. In another example, if there are multiple words or characters that commonly appear together in the constructed language (but that would not be efficiently tokenized by the tokenizer of the LLM), such as "for i=" that is commonly used in programming languages, the language converter 250
25 may replace the multi-word text string with a placeholder string of text that is a single word from the dictionary of common text strings. When the language converter 250 performs such a replacement operation, the language converter 250 may store a mapping between the original string of text and the
30 replacement string of text. The LLM may be fine-tuned or otherwise provided with training samples that illustrate the use of the placeholder string of text in the token-efficient language, so that output from the LLM will use the placeholder string of text. The stored mapping may then be used for conversion

from the token-efficient language to the target constructed language as described above.

[103] In some examples, there may be multiple instances of the language converter 250, each instance being designed for a different target constructed language (e.g., specific to the syntax, special characters and/or commands of a particular target constructed language). Additionally or alternatively, there may be multiple instances of the language converter 250, each instance being designed for a different tokenizer (e.g., specific to how a particular tokenizer tokenizes text). For example, one tokenizer may be designed to include leading whitespace in a token (e.g., " name" would be tokenized as a single token) while a different tokenizer may not include leading whitespace in a token. The token-efficient language for these two tokenizers may be slightly different to account for the different manner in which whitespace is tokenized by these tokenizers (without having to change how each tokenizer operates), and thus the instance of the language converter 250 selected for each tokenizer may be different. The appropriate instance of the language converter 250 may be automatically selected based on the target constructed language and/or tokenizer.

[104] In order for the LLM to generate output in the token-efficient language, the LLM may be fine-tuned, trained and/or prompted in a manner that enables the LLM to understand the rules and/or syntax of the token-efficient language. The LLM that has been fine-tuned, trained and/or prompted in this way may learn parameter values that enable the LLM to generate output that is a token sequence that follows the rules and/or syntax of the token-efficient language. That is, the tokens in the generated output are tokens recognized by a conventional tokenizer, but the tokens used and the sequence order is in accordance with the token-efficient language rather than a conventional constructed language (e.g., programming language) or natural language (e.g., human spoken language).

[105] In some examples, the computing system 200 may perform operations to fine-tune or train the LLM using training samples (which may be stored in the dataset 270) that pair sample natural language tasks with corresponding sample text in the target constructed language. Training samples

may be automatically generated using the language converter 250, as discussed above.

[106] FIG. 4 is a flowchart of an example method 400 for an example embodiment of the present disclosure, which may be performed by a computing system, in accordance with examples of the present disclosure. For example, a processing unit of a computing system (e.g., the processor 202 of the computing system 200 of FIG. 2) may execute instructions (e.g., instructions of the language converter 250 and prompt generator 260) to cause the computing system to carry out the example method 400. The method 400 may, for example, be implemented by an online platform or a server. The method 400 may be used to prompt a LLM to generate token-efficient output. The LLM may be a generative pre-trained transformer LLM, such as LLaMA, Falcon 40B, GPT-3, GPT-4 or ChatGPT, among others. It should be understood that the present disclosure is not intended to be limited to any particular LLM.

[107] In some examples, the method 400 may include an operation 402 in which one or more training samples are generated. As discussed above, training sample(s) may be generated using a language converter (e.g., the language converter 250 discussed above). The language converter may receive sample text in a target constructed language that is labeled with a sample natural language task and automatically generate corresponding sample text in the token-efficient language. The generated sample text in the token-efficient language may be paired together with the sample natural language task label in a training sample, such that the sample natural language task may be used as ground-truth for the corresponding sample text in the token-efficient language (or vice versa). In some examples, training sample(s) may be generated in this way ahead of time (e.g., prior to the method 400) and stored in a dataset (e.g., the dataset 270 of FIG. 2) and the operation 402 may not be performed. In some examples, training sample(s) may be generated as part of the method 400. For example, training sample(s) may be generated on-the-fly by the language converter 250 for fine-tuning the LLM and/or for use in one-shot or few-shot training in a prompt to the LLM. The ability to generate training sample(s) on-the-fly and automatically (e.g., without requiring a human to

manually create sample text in the token-efficient language for each corresponding sample natural language task) using the language converter 250 may provide savings in time and cost, as well as being more practical and flexible.

5 **[108]** In some examples, the method 400 may include an operation 404 in which the LLM is fine-tuned to generate token-efficient output. For example, one or more training samples (which may have been generated at the operation 402) that each pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task may be provided to the LLM to enable the LLM to generate token-efficient output. Fine-tuning the LLM may involve training the parameters in a final task-specific layer of the LLM. This may enable the LLM to generate token-efficient output without negatively affecting the overall performance of the LLM.

10 **[109]** In some examples, a training prompt may be provided to the LLM that includes one-shot or few-shot training. The training prompt may be a form of fine-tuning that enables the LLM to learn rules and/or syntax of the token-efficient language, without requiring hundreds of training samples. A training prompt may be similar to a task prompt that includes one-shot or few-shot training. In some examples, a prior task prompt that includes training sample(s) may be considered a training prompt for a subsequent task.

15 **[110]** In some examples, the LLM may already be trained to generate token-efficient output (e.g., pre-trained prior to the method 400) and the operation 404 may not be performed.

20 **[111]** At an operation 406, a task prompt is provided to the LLM (e.g., via an API call) for performing a natural language task and generating a token-efficient output based on the task prompt. For example, the task prompt may be generated by the prompt generator 260 as previously described with respect to FIG. 3. In particular, the natural language task may include generating output that can be used for coding or programming. The task prompt is provided to the LLM to prompt the LLM to perform the natural language task and generate token-efficient output. For example, the task prompt may be provided to the LLM via an API call to a remote LLM. The task prompt may be converted to a set

of tokens (e.g., by a tokenizer, which may be specific to the LLM) and the set of tokens may be provided to the LLM in sequential order. In some examples, tokenization of the task prompt may occur prior to the API call, such that the set of tokens is provided via the API call.

- 5 **[112]** Optionally, the operation 406 may include an operation 408. At optional operation 408, one or more training samples may be included in the task prompt, to provide in-context learning. Each training sample may pair a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task. The training
- 10 sample(s) may, for example, be stored in the dataset 270 ahead of time and retrieved from the dataset 270 and/or generated on-the-fly during the method 400 by the language converter 250. In some examples, training sample(s) may additionally or alternatively be provided to the LLM prior to providing the task prompt (e.g., training sample(s) may be used in fine-tuning the LLM at prior
- 15 operation 404).

- [113]** At an operation 410, output from the LLM is received. The output includes text related to the natural language task, and includes at least a portion of text in the token-efficient language. In particular, the portion of text in the token-efficient language can be represented by tokens (e.g., when tokenized by
- 20 a conventional tokenizer of the LLM) using fewer tokens than would be required for corresponding text in the target constructed language. As previously mentioned, the LLM generates output one token at a time. Accordingly, by prompting the LLM such that at least a portion of the output uses the token-efficient language, the output from the LLM can be provided using fewer tokens,
- 25 thus enabling a reduction in the consumption of computing resources (e.g., time, processing power, memory, etc.). Additionally, because the output can be provided using fewer tokens (and because the LLM generates output one token at a time), the time between prompting the LLM and receiving the complete output may be reduced, thus reducing the overall latency of the system. As well,
- 30 the risk of reaching the maximum number of output tokens before the complete output is provided may be reduced.

[114] Optionally, at an operation 412, the portion of text in the token-efficient language may be provided as input to another software module (e.g., another ML model). Providing the text in the token-efficient language, prior to conversion to the target constructed language, may reduce the amount of input data (i.e., reduced number of tokens) that is processed by the other ML model. This may enable the ML model to be less complex (e.g., fewer layers, fewer parameters, etc.) and thus requiring fewer computing resources to execute.

[115] Optionally, at an operation 414, the output from the LLM, including the portion of text in the token-efficient language, may be stored in an output log. The output log may store a history of outputs generated by the LLM and may provide historical or contextual information to help improve the performance of the LLM in subsequent tasks.

[116] At an operation 416, the portion of the text in the token-efficient language may be automatically converted to corresponding text in the constructed language. This may be performed using the language converter 250 (which may be specific to the target constructed language), for example. In some examples, converting the portion of text in the token-efficient language to the corresponding text in the constructed language may include automatically replacing a placeholder string of text in the token-efficient language with a mapped string of text in the constructed language. The mapping between the placeholder string of text and the mapped string of text may be stored in a memory accessible by the language converter 250. As previously discussed, the placeholder string of text is tokenized into fewer tokens than the mapped string of text, for example the placeholder string of text may be a common word in human language (thus most tokenizers may use a single token to represent the word) whereas the mapped string of text may be an uncommon word or a word not found in human language (e.g., CamelCase word, programming term, etc.). In some examples, the mapped string of text may be multiple words (or multiple commands) in the constructed language that frequently or always occur together.

[117] Optionally, the operation 416 may include an operation 418. At the operation 418, prior to converting the text in the token-efficient language to the

target constructed language, the text in the token-efficient language may be checked for validity. For example, the language converter 250 may parse the text in the token-efficient language to check for certain grammar and/or syntax expected in the token-efficient language, to check there are no unexpected text strings, etc. If the language converter 250 determines that the text in the token-efficient language is invalid, then conversion may be aborted and an error notification may be outputted. The text that could not be converted may nonetheless be stored in an output log and indicated as erroneous output, which may help improve future output from the LLM.

- 10 **[118]** In various examples, the present disclosure provides a technical solution that enables more efficient use of computing resources by a LLM. Since a LLM processes one token at a time as input, and outputs one token at a time, the use of a token-efficient language (which requires fewer tokens to represent a text) saves computing resources. In particular, a LLM may be prompted to
- 15 perform a task that requires output in a constructed language (e.g., programming language) that the LLM and/or tokenizer is not optimized for. For example, a LLM and its tokenizer may be mostly designed/trained on a natural language (e.g., human spoken language) corpus. Outputs in a constructed language may include text strings that are unexpected in natural language that
- 20 are therefore likely to be tokenized in an inefficient way. For example, programming languages frequently use terms that are uncommon in natural language. Output from an LLM that includes such terms is likely to require a larger number of tokens to represent and thus a drain on computing resources. Additionally, for very large outputs, there is a maximum limit on the number of
- 25 tokens that can be used. Output from a LLM that does not use tokens efficiently will more quickly reach that maximum limit. Examples of the present disclosure enable the LLM to generate output in the token-efficient language instead, thus providing significant savings in computing resources and helping to avoid reaching the maximum limit of output tokens.
- 30 **[119]** In cases where output from the LLM is further processed by another ML model, the token-efficient output from the LLM can be provided as input to the other ML model. Since the token-efficient output is shorter (in terms of

tokens and possibly in terms of string length) than the same output in the constructed language, this enables the other ML model to be designed for smaller inputs, which can result in a smaller ML model (and thus requiring fewer computing resources to train and deploy).

5 **[120]** Although the present disclosure has described a LLM in various examples, it should be understood that the LLM may be any suitable language model (e.g., including LLMs such as LLaMA, Falcon 40B, GPT-3, GPT-4 or ChatGPT, as well as other language models such as BART, among others).
10 Additionally, it should be understood that the present disclosure is not limited to any particular constructed language. Although JSON has been used in various examples, the present disclosure may be equally applicable to other constructed languages, including other programming languages.

15 **[121]** Although the present disclosure describes methods and processes with operations (e.g., steps) in a certain order, one or more operations of the methods and processes may be omitted or altered as appropriate. One or more operations may take place in an order other than that in which they are described, as appropriate.

20 **[122]** Although the present disclosure is described, at least in part, in terms of methods, a person of ordinary skill in the art will understand that the present disclosure is also directed to the various components for performing at least some of the aspects and features of the described methods, be it by way of hardware components, software or any combination of the two. Accordingly, the technical solution of the present disclosure may be embodied in the form of a software product. A suitable software product may be stored in a pre-recorded
25 storage device or other similar non-volatile or non-transitory computer readable medium, including DVDs, CD-ROMs, USB flash disk, a removable hard disk, or other storage media, for example. The software product includes instructions tangibly stored thereon that enable a processing device (e.g., a personal computer, a server, or a network device) to execute examples of the methods
30 disclosed herein.

[123] The present disclosure may be embodied in other specific forms without departing from the subject matter of the claims. The described example

embodiments are to be considered in all respects as being only illustrative and not restrictive. Selected features from one or more of the above-described embodiments may be combined to create alternative embodiments not explicitly described, features suitable for such combinations being understood within the
5 scope of this disclosure.

[124] All values and sub-ranges within disclosed ranges are also disclosed. Also, although the systems, devices and processes disclosed and shown herein may comprise a specific number of elements/components, the systems, devices and assemblies could be modified to include additional or fewer
10 of such elements/components. For example, although any of the elements/components disclosed may be referenced as being singular, the embodiments disclosed herein could be modified to include a plurality of such elements/components. The subject matter described herein intends to cover and embrace all suitable changes in technology.

15

CLAIMS

1. A method comprising:
 - providing a task prompt to a large language model (LLM) for performing a natural language task and generating token-efficient output based on the task
 - 5 prompt; and
 - receiving output from the LLM based on the task prompt, the output including text related to the task, at least a portion of the text being in a token-efficient language;
 - wherein the text in the token-efficient language is tokenized by a
 - 10 tokenizer of the LLM into fewer tokens than would correspond to a tokenization of corresponding text in a different constructed language.
2. The method of claim 1, further comprising:
 - converting at least the portion of the text in the token-efficient language
 - 15 to corresponding text in the constructed language.
3. The method of claim 2, wherein the converting is performed by a language converter.
- 20 4. The method of claim 3, wherein the language converter is specific to the constructed language.
5. The method of any one of claims 2 to 4, wherein converting at least the portion of the text in the token-efficient language to corresponding text in the
- 25 constructed language includes replacing a placeholder string of text in the token-efficient language with a mapped string of text in the constructed language, wherein the placeholder string of text is tokenized into fewer tokens than the mapped string of text.
- 30 6. The method of claim 5, wherein a mapping between the placeholder string of text and the mapped string of text is stored in a memory.
7. The method of any one of claims 1 to 6, further comprising:

generating one or more training samples that each pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task;

5 wherein at least one of the one or more training samples is provided to the LLM prior to providing the task prompt or is provided to the LLM in the task prompt.

8. The method of any one of claims 1 to 6, further comprising, prior to providing the task prompt:

10 fine-tuning the LLM to generate token-efficient output using training samples that each pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

15 9. The method of any one of claims 1 to 6, further comprising, prior to providing the task prompt:

providing a training prompt to the LLM for performing the same or different natural language task, the training prompt including at least one training sample that pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

25 10. The method of any one of claims 1 to 6, wherein the task prompt includes at least one training sample that pairs a sample natural language task with a corresponding sample text in the token-efficient language for performing the sample natural language task.

11. The method of any one of claims 1 to 10, further comprising:

30 providing the text related to the task, including at least the portion of text in the token-efficient language, as input to another software module.

12. A computer system comprising:

a processing unit configured to execute computer-readable instructions to cause

the computer system to perform any one of the methods of claims 1 to 11.

13. A computer-readable medium storing instructions that, when executed by a processing unit of a computing system, cause the computing system to perform
5 any one of the methods of claims 1 to 11.

14. A computer program comprising processor-executable instructions that, when executed by one or more processors of a computer, cause the computer system to carry out the operations of any one of the methods of claims 1 to 11.
10