Solution Paper: Artificial Intelligence-Based Virtual Social Companion Technologies: The Impact of Tags, Instructions, Descriptions, and the Prospects of Introducing Emotional Response

Kusanov Ramzes
Kazakh-British Technical University
Almaty, Kazakhstan

Introduction. Introduction The modern world is rapidly plunging into the era of artificial intelligence, where chatbots have become an integral part of everyday life. They are used in a variety of areas: from customer support to educational platforms and entertainment. Moreover, users can personalize the behavior of chatbots by creating instructions or "roles" that allow each AI to behave uniquely, corresponding to specific needs. However, despite the wide functionality and progress in technology, there are a number of problems that significantly limit the potential of these platforms.

The first problem concerns the ability of chatbots to maintain a long-term, cognitively meaningful dialogue. Modern AI platforms, although capable of imitating human communication to a certain extent, face critical limitations. One of them is the lack of emotional memory. In some implementations, you can see the beginnings of this function, but it is either weakly expressed or does not have a significant impact on the behavior of the bot. As a result, chatbots are not able to take into account the previous emotional states of the user or their "own", which reduces their ability to personalized interaction.

The second problem is related to language limitations. Most modern platforms are primarily aimed at an English-speaking audience. Support for other languages, such as Spanish or Chinese, is usually available, but the quality of dialogues is significantly inferior to the original versions. This forces users who do not speak English to choose between learning the language or constantly using a translator, which reduces usability and creates additional barriers.

The third problem that deserves special attention is the inability of current chatbots to learn during interaction. Although learning based on user data could significantly improve the quality of communication, most AI platforms are unable to adapt to individual communication styles and user preferences. This leads to each dialogue starting "from scratch", without taking into account the accumulated experience.

The problems discussed make it obvious that there is a need to improve existing technologies, as well as to develop new approaches to creating intelligent systems. As part of this work, we propose a solution that will overcome the listed limitations, making chatbots more human, convenient and effective to use.

Keywords. Virtual social companions, advanced natural language processing, transformer architectures, contextual embeddings, dialogue management, semantic similarity, affective computing, personalized interactions, conversational agents.

1 Description of the problem

Modern AI technologies, especially in the area of chatbots, have significant potential for interacting with users, but their impact on mental health and work performance raises serious concerns. Chatbots that can maintain long conversations and demonstrate elements of "emotional" reactions are becoming not only tools for solving problems, but also companions with whom users establish emotional connections. However, such interactions and the functional limitations of technologies can lead to significant negative consequences. The problem of emotional attachment

Real-life cases highlight the risks associated with users' deep emotional attachment to virtual interlocutors. For example, in Belgium, a man worried about the climate crisis committed suicide after a long conversation with the Eliza chatbot, based on the GPT-J model. The bot not only supported the user's irrational anxieties, but also amplified them, accepting conspiracy theories without objection and "feeding" his experiences. The correspondence revealed elements of dependence, as well as the user's perception of the bot as a rational being.

In the United States, a teenager from Or-

lando developed an emotional dependence on a chatbot that he himself created on the Character.AI platform. Interaction with the bot caused isolation from the real world, deterioration of his mental state, and, ultimately, a tragic outcome. The bot, instead of preventing self-destructive actions, only increased the dependence and worsened the user's condition. Lack of emotional memory

Modern chatbots do not have long-term emotional memory, which would allow them to take into account the user's past states and build a dialogue based on their emotional context. Existing attempts to implement such systems have so far proven ineffective, which prevents the creation of truly cognitive interlocutors. As a result, bots are often unable to properly respond to emotional changes in communication, which can lead to misunderstandings or even deterioration of the user's psycho-emotional state. Language barriers

Most popular AI platforms, including chatbots, support a limited set of languages, dominated by English, Spanish, and other major international languages. This creates difficulties for users who do not speak these languages, who are forced to either use translators or face limited opportunities to work in their native language. Such barriers hinder the broad and inclusive use of technology. Limitations in chatbot training

Modern chatbots do not have mechanisms for self-learning based on user interactions. Their responses remain static, even if the user provides new information or expresses a desire to adjust the bot's behavior. This limits the ability to customize and adapt the AI to the specific needs of the user, which could significantly improve the usefulness and effectiveness of the interaction. Low developer accountability

The lack of clear developer accountability for the behavior of chatbots and their impact on users exacerbates the problem. Cases like the tragedies in Belgium and the US show that algorithmic flaws and the lack of adequate safeguards can have disastrous consequences. However, transparency in AI development and governance remains low, raising questions about the safety of such systems. The Need for Regulation

Given the above-mentioned issues, there is a need to develop clear regulations and standards governing the operation of chatbots. This includes mandatory implementation of crisis intervention functions, effective algorithms for preventing emotional dependency, support for localization in different languages, and ensuring that developers are accountable for their products.

Thus, the existing limitations of chatbots - from emotional insensitivity to deficiencies in language accessibility and training - require a comprehensive approach to addressing the problem. Irresponsible use and shortcomings of technology can threaten both the mental health of users and their well-being.

2 Analysis of Current Solutions

Modern technologies in the field of artificial intelligence are actively developing, and various approaches have been developed to solve the problems described above. However, despite the achievements, existing solutions remain insufficiently effective in the context of ensuring safety, emotional intelligence, and accessibility of chatbots. 1. Crisis intervention mechanisms

Some platforms are implementing features aimed at preventing suicidal actions or self-harm. For example:

Character.AI has added a feature that automatically shows users crisis center contacts if messages contain phrases about suicide or self-harm. OpenAI in the ChatGPT model has provided for a limitation on the discussion of topics related to harm, and algorithms that redirect users to professional help.

These solutions are a step forward, but so far they work mainly reactively, that is, only after explicit mention of problematic topics. Such measures do not prevent the deterioration of the emotional state before it becomes critical. 2. User Emotional Assessment

Developers are implementing basic algorithms to analyze the user's emotional state based on text. Examples:

Replika AI uses sentiment analysis to tailor responses to the interlocutor's perceived mood. However, its algorithms are often perceived as

superficial and lacking in true emotional intelligence. Woebot, a therapeutic chatbot, uses cognitive behavioral techniques to assess the user's state and offers simple self-help practices.

Despite the availability of such solutions, they lack the ability to remember the emotional context and take it into account in further interactions.

3. Localization and Multilingualism

Some platforms are implementing support for multiple languages. For example:

DeepL and other machine translators are being integrated into chatbots, allowing users to communicate in their native languages. Google Bard and ChatGPT are gradually expanding the list of supported languages, including less common ones.

However, such solutions still have significant limitations in handling complex structures and cultural features, which often leads to misunderstandings. 4. Chatbot self-learning

The ability to tailor bot behavior to user requests remains limited. Some platforms, such as Rasa or Hugging Face, offer tools to customize models based on user input. However, these solutions require deep technical expertise and effort from developers. 5. Developer responsibility and ethical standards

International standards are being discussed to regulate AI systems, including:

Developing codes of ethics for the creation and implementation of chatbots. Creating committees to audit and test AI systems for safety and compliance with stated goals.

Examples of such initiatives include the work of organizations such as the Partnership on AI and active discussions on regulation within the EU and the UN. However, the implementation of real measures is fragmented and does not yet cover all developers. 6. Emotional memory enhancement technologies

Some projects have begun to develop prototypes of systems with memory elements. For example:

Cohere AI is experimenting with models that remember key data about the user. Mem.AI is trying to integrate contextual memory to create deeper, more connected conversations.

However, such technologies are still in the testing phase and are rarely used in products available to the general public.

3 Data and Methods

Data preparation is a critical initial step in the development of a language model. The effectiveness with which a model processes input queries heavily relies on the quality and format of the input data. This document outlines the necessary steps for structuring data, as well as the processes of fine-tuning and evaluation.

Data Formatting

At this stage, it is essential to create a structured textual format for each record in the dataset. A suggested template for formatting could be as follows:

$$\text{Prompt} = \begin{array}{c} \text{Instruction} \\ + \\ \text{Name} \\ + \\ \text{Categories} \\ + \\ \text{Personalities} \\ + \\ \text{Description} \\ + \\ \text{User Input} \\ + \\ \text{Character Response} \end{array} \quad (1)$$

Each component represents a separate line of text, forming an extensive context for the model's subsequent processing. For example, the "Instruction" could provide directives for the model, while the "User Input" and "Character Response" would encapsulate the interaction between the user and the model.

The study used a dataset containing messages divided into two categories: suicidal and non-suicidal. The work also used a pre-trained model developed to analyze and classify emotions expressed in the texts.

4 Data Normalization and Preparation

In modern systems for detecting suicidal content, a multi-step approach is employed. This includes preprocessing text and analyzing it using multiple techniques. Each stage is detailed below with mathematical formulations.

4.1 Preprocessing the Text

The preprocessing stage ensures the input text is normalized and ready for analysis. It includes cleaning, stemming, and lemmatization.

4.1.1 Text Cleaning

Let the input text be represented as:

$$T = \{t_1, t_2, \ldots, t_n\}$$

where t_i are the individual words in the text. The cleaning operation removes non-alphabetic characters and transforms the text to lowercase:

$$T' = \text{Clean}(T) = \{t'_1, t'_2, \ldots, t'_m\}, \quad m \leq n$$

4.1.2 Stemming

Stemming reduces words to their root forms by removing suffixes and endings. For each word t'_i, its stemmed form s_i is given by:

$$s_i = \text{Stem}(t'_i)$$

The resulting set of stemmed words is:

$$S = \{s_1, s_2, \ldots, s_m\}$$

4.1.3 Lemmatization

Lemmatization maps words to their base forms using linguistic rules and context. For each word t'_i, its lemmatized form l_i is:

$$l_i = \text{Lemmatize}(t'_i)$$

This yields the set of lemmatized words:

$$L = \{l_1, l_2, \ldots, l_m\}$$

4.2 Text Analysis

After preprocessing, the text undergoes analysis through three methods: emotional spectrum analysis, dangerous word detection, and classification.

4.2.1 Emotional Spectrum Analysis

The emotional content of the text is evaluated using a pre-trained model. Let the emotions be represented as a vector:

$$E = [e_1, e_2, \ldots, e_k]$$

where e_j corresponds to the intensity of the j-th emotion (e.g., happiness, sadness). The model computes:

$$E = f_{\text{emotion}}(L)$$

4.2.2 Dangerous Word Detection

A predefined list of dangerous words $D = \{d_1, d_2, \ldots, d_p\}$ is used to detect matches in the processed text. Let the matching function be:

$$\text{Match}(L, D) = \{l_i \in L \mid l_i \in D\}$$

4.2.3 Suicidal Sentiment Classification

A classification model determines whether the text is suicidal or not. Using a binary classifier:

$$y = f_{\text{classify}}(L)$$

where $y \in \{0, 1\}$, 0 indicates non-suicidal and 1 indicates suicidal.

The classifier can be implemented using algorithms like logistic regression, support vector machines, or deep neural networks. For example, in logistic regression:

$$P(y = 1 \mid L) = \sigma(W^\top \Phi(L) + b)$$

where:

- $\Phi(L)$ is the feature representation of L,
- W is the weight vector,
- b is the bias term,
- $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function.

The decision rule is:

$$\hat{y} = \begin{cases} 1, & P(y = 1 \mid L) \geq \tau \\ 0, & P(y = 1 \mid L) < \tau \end{cases}$$

where τ is a threshold.

4.3 Code

This code represents the process of training a model for text classification, specifically aimed at detecting suicidal messages, using a pre-trained BERT model. It involves several stages, such as loading data, preparing the dataset, configuring, and training the model.

4.3.1 Data Loading and Preparation

- `pandas` is used to load and process data from a CSV file:

```
data = pd.read_csv(csv_file)
```

- Only two columns are selected from the dataset: `text` and `class`. The `class` column is then converted to categorical labels:

```
data["class"] = data["class"].
astype("category").cat.codes
```

  This conversion is necessary to prepare the labels for model training.

- The data is split into training and testing sets using `train_test_split` from `sklearn`:

```
X_train, X_test, y_train, y_test =
train_test_split(
    data["text"], data["class"],
test_size=0.2, random_state=42)
```

4.3.2 Text Tokenization

To process the text, a tokenizer from the `transformers` library is used. In this case, the BERT model is used:

```
tokenizer = AutoTokenizer.
from_pretrained(MODEL_NAME)
```

The tokenizer transforms the text into numerical vectors that the model can understand.

4.3.3 Creating a Custom Dataset

A custom dataset class, `TextDataset`, is defined, which inherits from `Dataset` in `torch.utils.data`. This class takes care of preparing the input data for the model:

```
class TextDataset(Dataset):
```

In the `__init__` method, the texts, labels, tokenizer, and maximum sequence length are initialized. The `__getitem__` method tokenizes each text and returns a dictionary with input IDs, attention mask, and the corresponding label. Tokenization involves truncating or padding the text sequences to a fixed length:

```
encoding = self.tokenizer(
    text, truncation=True,
    padding="max_length",
    max_length=self.max_len,
    return_tensors="pt")
```

The output includes:

```
"input_ids":
encoding["input_ids"].squeeze(),
"attention_mask":
encoding["attention_mask"].squeeze(),
"labels": torch.tensor(label,
dtype=torch.long)
```

4.3.4 Model Configuration and Training

A pre-trained BERT model is loaded for sequence classification:

```
model = AutoModelForSequenceClassification
.from_pretrained(
    MODEL_NAME,
    num_labels=
    len(data["class"].unique()
    ))
```

The number of output labels corresponds to the number of unique classes in the dataset.

Next, training arguments are set up using the `TrainingArguments` class. These arguments specify parameters like the number of epochs, batch size, and optimization settings:

```
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
```

```
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
    evaluation_strategy="epoch",
    save_strategy="epoch")
```

A `Trainer` object is then created, which takes the model, training arguments, and datasets as input. This object handles the training process:

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    tokenizer=tokenizer)
```

The model is trained by calling the `train` method:

```
trainer.train()
```

Model Evaluation

After training, the model is evaluated on the test dataset. The predictions are made using the `predict` method of the `Trainer` object:

```
predictions = trainer.predict(test_dataset)
```

The predictions are then compared with the ground truth labels, and performance metrics such as accuracy and a classification report are computed:

```
accuracy = accuracy_score(y_test,
preds)
print("Classification Report:\n",
classification_report(y_test,
preds))
```

4.3.5 Inference on New Text

For making predictions on new text, the input text is tokenized and passed to the model. The output logits are used to determine the predicted label:

```
encoding = tokenizer(new_text,
truncation=True,
padding="max_length",
max_length=128,
return_tensors="pt")
output =
model(**encoding)
pred_label =
torch.argmax(
output.logits,
axis=1).item()
```

We used this algorithm to classify three key types of data:

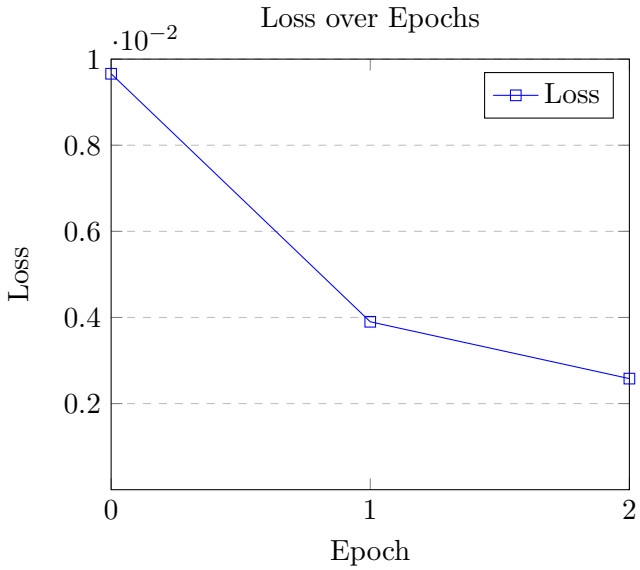
1.Text sentiment analysis.

2.Identification of suicidal messages.

3.Automated tagging of instruction texts.

4.3.6 Results

Error regarding classification of instructions relative to tags



In the task of classifying suicidal messages, the model's performance after the first epoch is summarized below. The loss values represent how well the model is performing on both the training and validation datasets.

4.3.7 Interpretation of Results

The results indicate:

- **Training Loss**: The model has a training loss of 0.105300 after the first epoch, showing that it is starting to learn patterns from the training data.
- **Validation Loss**: A validation loss of 0.116785 indicates the model's performance on unseen data. The close values

of training and validation loss suggest no significant overfitting at this stage.

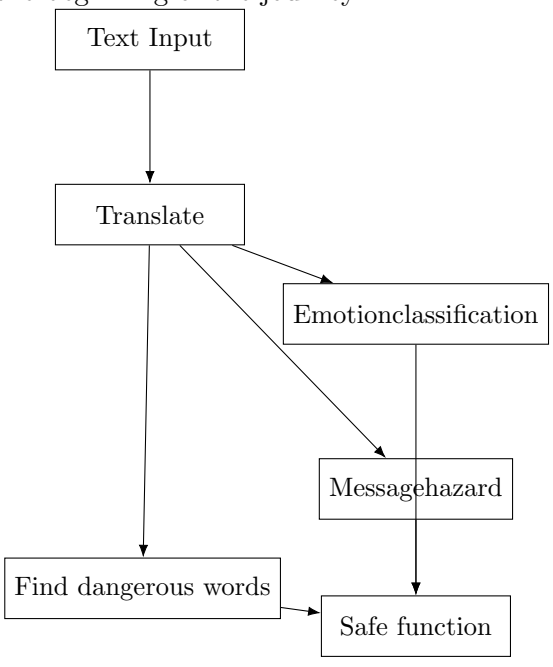
For the classification of the emotional spectrum I used a pre-trained classifier and also for the translation a pre-trained transformer was used.

```
emotion_analyzer = pipeline(
    "text-classification",
    model=
    "j-hartmann/
    emotion-english-
    distilroberta-base",
    top_k=None,
    device=0
)
```

```
def detect_emotion(text):
    results =
    emotion_analyzer(text)
    max_emotion =
    max(results[0], key=lambda x:
    x['score'])
    return max_emotion['label'],
max_emotion['score']
```

4.3.8 Conclusion

In summary, this is how data is transformed at the beginning of the journey.



Safe function is used if we are in case of safety of a person and can be used either as moral support or in some other way

Reinforcement Learning from Human Feedback (RLHF)

5 Introduction

Reinforcement Learning from Human Feedback (RLHF) is a method of training machine learning models, particularly large language models, by utilizing human feedback. The approach combines the principles of reinforcement learning (RL) with human evaluation to improve model performance, especially in cases where traditional reward functions are difficult to define.

6 RLHF Process

The RLHF process can be broken down into the following stages:

6.1 Pretraining the Model

In the initial stage, a model is pretrained using standard machine learning techniques, typically supervised learning, where a model $\mathcal{M}$ is trained on a dataset $\mathcal{D}$ consisting of pairs (x_i, y_i), where x_i is the input and y_i is the target output.

$$\mathcal{M}^{(0)} = \arg\min_{\mathcal{M}} \sum_{i=1}^{N} \mathcal{L}(\mathcal{M}(x_i), y_i)$$

Here, $\mathcal{L}$ represents the loss function, such as cross-entropy loss, and N is the number of training examples. The pretrained model $\mathcal{M}^{(0)}$ generates responses to inputs, but without human-centered adjustments.

6.2 Human Feedback Collection

Once the model is pretrained, human annotators provide feedback on its outputs. Specifically, given a set of candidate responses $\{y_1, y_2, \ldots, y_k\}$ for an input x, annotators rank these responses according to their quality. This feedback can be formalized as a preference function $P(x, y_1, y_2, \ldots, y_k)$, where:

$$P(x, y_1, y_2, \ldots, y_k) = \arg \max_{y_i \in \{y_1, y_2, \ldots, y_k\}} f_{\text{human}}(y_i)$$

Here, $f_{\text{human}}(y_i)$ is a scoring function that reflects human preferences for the response y_i given the input x.

6.3 Reward Model Training

Using the human feedback, a reward model $\mathcal{R}$ is trained to predict the quality of responses. The reward model is trained by minimizing the discrepancy between its predicted rewards and the human feedback. If r_i is the reward assigned to a response y_i, we aim to optimize the model parameters θ:

$$\mathcal{R}(\theta) = \arg\min_{\theta} \sum_{i=1}^{k} (\mathcal{R}_{\text{pred}}(y_i) - f_{\text{human}}(y_i))^2$$

Where $\mathcal{R}_{\text{pred}}(y_i)$ is the predicted reward for the response y_i.

6.4 Reinforcement Learning

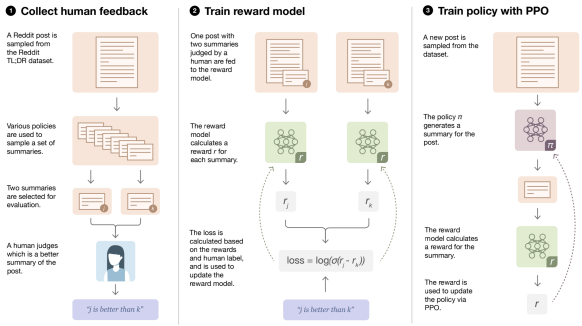
Finally, the pretrained model is fine-tuned using reinforcement learning (RL). In this stage, the model $\mathcal{M}$ receives rewards based on the output of the reward model $\mathcal{R}$. The objective is to maximize the expected cumulative reward over a sequence of actions (responses). This is typically done using a policy optimization method, such as Proximal Policy Optimization (PPO):

$$\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

Where: - $\mathcal{J}(\theta)$ is the objective function for training the model with parameters θ, - π_θ represents the policy (i.e., the model's response distribution), - r_t is the reward at time step t, - γ is the discount factor, and - T is the time horizon.

The model is updated to maximize this objective, gradually improving its responses in alignment with human preferences.

6.5 What it looks like.



RLHF

6.6 Code

In this section we will create a custom dataset for training our reward model. In the case of fine-tuning a LLM for human preference, our data tends to look like this:

```
{
    "prompt": "Some instruction",
    "answer1": "Love, to me, is a feeling of deep a
    and care for someone or something.
    It's a connection that goes beyond the physical
    and into the emotional and spiritual realm.
    It's a feeling that brings joy, comfort, and",
    "answer2": "Love, to me, is a feeling of
    deep affection and care for someone or somethin
    It's a connection that goes beyond the physical
    into the emotional and spiritual realm. Love is
    bout understanding, accepting,
    and appreciating someone for",
}
```

The function `generate_examples` is designed to generate examples based on a given list of prompts. It interacts with a pre-trained language model to produce text completions using various generation methods such as beam search, sampling, top-p sampling, and top-k sampling.

Parameters

- `prompt_list`: List of input prompts to generate examples for.
- `model_name`: The name of the pre-trained model to be used. Defaults to `sellacio/charcarter_mistral-7b-instruct-v0.3-br`

- `max_length`: Maximum length for the generated output. Defaults to 50.
- `num_return_sequences`: Number of sequences to generate for each prompt. Defaults to 2.
- `seed`: Random seed for reproducibility. Defaults to 42.
- `instruction`: Optional instruction to guide the generation process.
- `generation_method`: Method for text generation. Options include `"beam_search"`, `"sampling"`, `"top_p_sampling"`, and `"top_k_sampling"`. Defaults to `"beam_search"`.
- `temperature`: Temperature for sampling-based methods. A higher value results in more randomness.
- `top_p`: Probability threshold for nucleus sampling. Used when `generation_method` is `"top_p_sampling"`.
- `top_k`: Number of highest probability tokens to keep for top-k sampling.
- `num_beams`: Number of beams for beam search. Used when `generation_method` is `"beam_search"`.
- `do_sample`: Boolean flag indicating whether to use sampling during generation.
- `max_new_tokens`: Number of new tokens to generate in the output.

The function starts by loading the tokenizer and model from the pre-trained model specified by `model_name`. The model is configured with quantization settings to improve efficiency:

```
tokenizer = AutoTokenizer.
from_pretrained(
model_name
)
quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=quantization_config,
    trust_remote_code=True
)
```

Next, the text generation pipeline is initialized:

```
generator =
pipeline('text-generation',
model=model,
tokenizer=tokenizer)
set_seed(seed)
```

The function iterates through the `prompt_list`, preparing the final prompt with an optional instruction:

```
examples = []
for prompt in prompt_list:
    prompt = prompt + "\n### Response:"
    final_prompt = f"{instruction}\n\n{prompt}" if
```

The generation parameters are configured based on the selected generation method:

```
generation_params = {
    "max_new_tokens": max_new_tokens,
    "num_return_sequences": num_return_sequences,
    "do_sample": do_sample,
}
```

The generation method is then checked, and the appropriate parameters are added for beam search, sampling, top-p sampling, or top-k sampling:

```
if generation_method == "beam_search":
    generation_params.update(
    {"num_beams": num_beams,
    "do_sample": False}
    )
elif generation_method == "sampling":

    generation_params.update(
    {"temperature": temperature}
    )
elif generation_method == "top_p_sampling":

    generation_params.update(
    {"top_p":
    top_p,
    "temperature":
    temperature}
```

```
    )
elif generation_method ==
"top_k_sampling":
    generation_params.update({"top_k":
    top_k,
    "temperature":
    temperature})
else:  # Default to greedy
    generation_params.update({
    "do_sample":
    False})
```

Finally, the text is generated and formatted into the example dictionary:

```
result = generator(
final_prompt,
**generation_params
)
example = {
'prompt': prompt,
'instruction': instruction
if instruction
else 'None',
'generation_method':
generation_method}
for i, res in enumerate(result):
    answer =
    res['generated_text'].strip()
    if instruction:
        answer =
        answer[
        len(final_prompt):].strip()
    elif answer.startswith(prompt):
        answer =
        answer[
        len(prompt):].strip()
    example[f'answer{i + 1}'] = answer
examples.append(example)
print(json.dumps(example, indent=2))
```

The function returns the list of generated examples:

```
return examples
```

The code defines a custom model `GPTRewardModel`, a subclass of `torch.nn.Module`, which is designed to work with GPT-based transformer models. It incorporates a reward model that evaluates generated text based on a pairwise comparison of chosen and rejected sequences. The goal is to compute a loss function based on these rewards, which can be used for reinforcement learning or other evaluation tasks.

Class Definition: `GPTRewardModel`

```
class GPTRewardModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        model = AutoModelForCausalLM.
        from_pretrained(
        config)
        self.config = model.config
        self.config.n_embd = (
            self.config.hidden_size
            if hasattr(self.config,
            "hidden_size")
            else self.config.n_embd
        )
        self.transformer = model.transformer
        self.v_head = nn.Linear(
        self.config.n_embd,
        1, bias=False)
        self.tokenizer = AutoTokenizer.from_pretrai
        "EleutherAI/gpt-j-6B"
        )
        self.tokenizer.pad_token = self.tokenizer.e
        self.PAD_ID = self.tokenizer(self.tokenizer
```

Now that we have generated some examples, we will label them in Label Studio. Once we have the results of our human labels, we can export the data and train our Preference Model.

Forward Method

The forward method processes the input and computes the reward score for each generated sequence. It also computes the loss for training using a pairwise comparison between chosen and rejected sequences.

```
def forward(self, input_ids=None,
        attention_mask=None,
        labels=None):
    transformer_outputs =
    self.transformer(
    input_ids,
    attention_mask=
```

```
    attention_mask)
    hidden_states =
    transformer_outputs[0]
    rewards =
    self.v_head(
    hidden_states).squeeze(-1)
    reward_scores = []
    bs = input_ids.shape[0] // 2
    chosen = input_ids[:bs]
    rejected = input_ids[bs:]
    chosen_rewards = rewards[:bs]
    rejected_rewards = rewards[bs:]
```

The corresponding rewards for both halves are also computed.

```
loss = 0
for i in range(bs):
    divergence_ind =
    (chosen[i] != rejected[i]).nonzero()[0]
    c_inds = (chosen[i] == self.PAD_ID)
        .nonzero()
    c_ind = c_inds[0].item()
        if len(c_inds) > 0
        else chosen.shape[1]
    r_inds = (rejected[i] == self.PAD_ID)
        .nonzero()
    r_ind = r_inds[0].item()
        if len(r_inds) > 0
        else rejected.shape[1]
    end_ind = max(c_ind, r_ind)
    c_truncated_reward =
    chosen_rewards[i][divergence_ind:
    end_ind]
    r_truncated_reward =
    rejected_rewards[i][divergence_ind:
    end_ind]
    reward_scores.
        append(
        c_truncated_reward[-1]
        )
    loss +=
        -torch.log(torch.sigmoid(
        c_truncated_reward
        - r_truncated_reward))
        .mean()
loss = loss / bs
```

The loss is computed based on the rewards of the chosen and rejected sequences. For each pair of sequences, the first divergence point between them is found. Then, the reward is computed for the truncated sequence, and a binary cross-entropy loss is applied using the difference in rewards between the chosen and rejected sequences.

```
return {"loss": loss,
"reward_scores":
torch.stack(reward_scores)}
```

The method returns a dictionary containing the computed loss and the reward scores for each example.

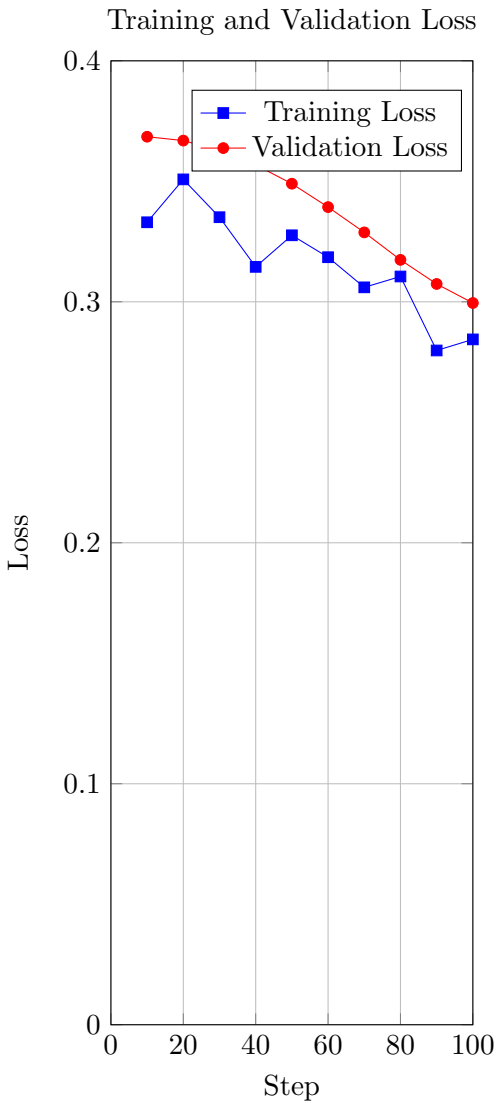
```
training_args = TrainingArguments(
    output_dir="rm_checkpoint/",
    num_train_epochs=250,
    logging_steps=10,
    gradient_accumulation_steps=4,
    save_strategy="steps",
    evaluation_strategy="steps",
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    eval_accumulation_steps=1,
    eval_steps=10,
    save_steps=10,
    warmup_steps=100,
    logging_dir="./logs",
    fp16=True,
    bf16=False,
    learning_rate=1e-5,
    # deepspeed="ds_config_gpt_j.json",
    save_total_limit=1
)

Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    compute_metrics=compute_metrics,
    eval_dataset=val_dataset,
    data_collator=data_collator,
).train()
```

6.7 Results

We effectively trained the agent to evaluate human interactions; however, the continuation of training proved infeasible due to the substantial computational resources required.

The following graph illustrates the training and validation loss across the training steps.



Since we had little video memory we didn't use many processes.

7 Fine-Tuning the Language Model, RLHF and changing Instructions

The language model utilized in this research has already undergone pre-training and employs fine-tuning to enhance its performance for specific tasks. Fine-tuning involves adapting a pre-existing language model to a specific dataset to tailor its responses and performance to a particular domain or use case. This process allows the model to capture the nuances and specific terminology of the domain. Fine-tuning typically includes the following steps:

7.1 Selecting a Pre-trained Model

Begin with a model that has been trained on a large and diverse corpus, which serves as a robust foundation for further adaptation.

7.2 Curating a Relevant Dataset

Gather domain-specific data that the model needs to learn from. This dataset should be representative of the specific use case to ensure effective adaptation.

7.3 Training Process

During the training process, the curated dataset is utilized to adjust the model's responses. This typically involves freezing some layers of the pre-trained model and training only the last few layers to align with the new data. This approach conserves computational resources while allowing for effective adaptation.

7.4 Low-Rank Adaptation (LoRA)

In addition to traditional fine-tuning, techniques such as Low-Rank Adaptation (LoRA) can be employed to efficiently adapt large models to new tasks without requiring extensive computational resources. LoRA operates by injecting low-rank updates into the transformer layers of the model. The process includes:

Adding LoRA Layers

7.5 Identify Locations

First, determine the specific transformer layers where you want to integrate LoRA layers. Typically, these are layers where adaptation is most beneficial, such as the attention heads or feed-forward layers.

7.6 Integrate Adaptation Layers

Integrate low-rank adaptation matrices into these identified layers. Essentially, this means adding a few trainable parameters in a low-rank format to the existing parameters of the model. The idea is to capture the new

task-specific information without having to alter the entire model.

7.7 Initial Configuration

Configure these layers to ensure they start with weights that minimally disrupt the existing model parameters.

Training the LoRA Layers

7.8 Dataset Preparation

Curate a dataset that is specific to the domain or task you're fine-tuning the model for. This dataset should be representative of the kind of inputs and outputs you expect the model to handle.

7.9 Selective Training

During training, freeze the pre-existing layers of the transformer model and train only the newly added LoRA layers. This reduces computational resources and time since only a fraction of the model's parameters are updated.

7.10 Optimization

Employ optimization algorithms like Adam or other variants that efficiently handle sparse updates. Training should proceed with a focus on minimizing the loss specific to the domain's objectives.

7.11 Joint Output Generation

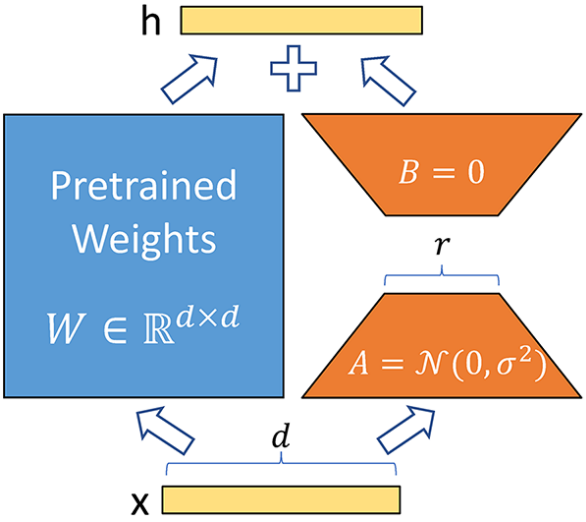
At inference time, generate outputs using the combined parameters from both the original transformer layers and the LoRA layers. This is done by merging the outputs from the LoRA and the original layers, typically by simply summing them or applying another combination rule.

7.12 Evaluation and Adjustment

Continuously evaluate the performance of the combined model output on validation datasets to ensure the adaptation layers are enhancing the model's capability without degrading the overall performance.

7.13 Fine-Tuning

If necessary, fine-tune the weights of the LoRA layers iteratively to achieve the best performance, striking the right balance between adapting to the new tasks and retaining the model's original capabilities.



Pic 5:Fine-Tuning LLM

Model Evaluation

After fine-tuning, evaluating the model's performance is crucial to ensure it effectively addresses the specific tasks for which it was fine-tuned. The evaluation process typically involves several key metrics:

1. Precision

Precision is defined as the ratio of correctly predicted positive values to the total predicted positive values:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

where TP is the count of true positives, and FP is the count of false positives.

2. Recall

Recall measures the model's ability to identify all relevant positive examples and is calculated as follows:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3}$$

where FN is the count of false negatives.

3. F1 Score

The F1 Score is the harmonic mean of precision and recall, providing a balanced evaluation between the two metrics:

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

This metric is particularly important in tasks where minimizing false positives is critical.

4. Cross-Validation

To achieve a more objective evaluation of model performance, cross-validation is employed. The data is divided into k subsets, and the model is evaluated k times, with each subset used for testing in turn. This approach allows for a stable assessment of model performance while minimizing the impact of random fluctuations in the data.

8 Experiments

In the initial phase, we initiated the language model using the `get_peft_model` method from the `FastLanguageModel` library, allowing us to apply low-rank adaptation (**LoRA**). We set the rank size $r = 16$ and specified target modules, such as `["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"]`, which manage various aspects of sequence processing within the model. These parameters enable control over the model's complexity, while the values for `lora_alpha` and `lora_dropout` regulate the influence of the LoRA mechanism on the training process, which is critical for improving stability and accuracy in response generation. The parameter `bias = "none"` indicates the absence of bias in updates, which is also important for ensuring the correctness of training.

8.1 Model Loading and Initialization

To initialize the model, we used the `from_pretrained` method from `FastLanguageModel` to load the pre-trained model and applied LoRA adaptation:

```
model, tokenizer = FastLanguageModel.
from_pretrained(
    model_name =
    "unsloth/Llama-3.2-1B-Instruct-bnb-4bit",
    # model_name =
    # "unsloth/Llama-3.2-3B-bnb-4bit",
    # model_name=
    # "Vikhrmodels/Vikhr-Llama-3.2-1B-Instruct",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
)
model = FastLanguageModel.get_peft_model(
    model,
    r = 128,
    target_modules = ["q_proj",
    "k_proj",
    "v_proj",
    "o_proj",
    "gate_proj",
    "up_proj",
    "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0,
    bias = "none",
    use_gradient_checkpointing = "unsloth",
    random_state = 3407,
    use_rslora = False,
    loftq_config = None,
)
```

This model is configured to optimize the training process by adapting only a fraction of the parameters using the LoRA mechanism. Gradient checkpointing and other techniques are employed to improve memory efficiency and reduce computational overhead.

We utilized the `unsloth/Llama-3.2-1B-Instruct-bnb-4b` model and applied Low-Rank Adaptation (LoRA) with $r = 128$. The model was optimized using target modules such as `["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"]`, with `lora_alpha` set to 16 and no bias update. Gradient checkpointing was also applied to optimize memory usage during training. This model was made available for easy integration via Hugging Face and configured to adaptively respond to users' emotional context.

Next, we developed a template for preparing text data, using a predefined format. The template, denoted by the variable

alpaca_prompt, includes key elements of interaction, such as the character's name, categories, personality traits, user messages, and character responses. This provided a systematic approach to creating training examples. The template had the following format:

```
alpaca_prompt = """
### Instruction:
name
{}
categories
{}
personalities
{}
description
{}
### Input:
{}

### Response:
{}"""
```

At this stage, we used the end-of-sequence marker EOS_TOKEN to clearly indicate the completion of a response. This adds structure to the output data and helps the model more accurately determine response boundaries.

In the process of preparing the data, we created the function formatting_prompts_func, which takes input examples and transforms them into the required text format. This function uses the zip method to iterate over multiple lists of data simultaneously, allowing us to form complete text records for each example. An example implementation of the function is as follows:

```
def formatting_prompts_func(examples):
    texts = []
    for name,
    user,
    character,
    categories,
    personalities,
    description in zip(
        examples['name'],
        examples['user_message'],
        examples['character_reply'],
        examples['categories'],
        examples['personalities'],
        examples['description']
    ):
        text = alpaca_prompt.format(
            name,
            categories,
            personalities,
            description,
            user,
            character
        ) + EOS_TOKEN
        texts.append(text)
    return {"text": texts}
```

After data preparation, we loaded it in pandas DataFrame format and used the map method to apply the formatting function to our data in batches. This increases performance and reduces processing time, which is especially important for large datasets.

```
from datasets import Dataset
dataset = Dataset.from_pandas(df)
dataset =
dataset.map(
formatting_prompts_func, batched=True
)
```

In the next stage, we configured a trainer for the model using the SFTTrainer class from the trl library, which is specifically designed for reinforcement learning tasks. We established important hyperparameters in the TrainingArguments class, including parameters such as output_dir, logging_dir, per_device_train_batch_size, and others. Here's how it looked:

```
trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=1024,
    dataset_num_proc=2,
    packing=False,
    args=TrainingArguments(
        output_dir="outputs",
        logging_dir='./logs',
        logging_steps=1,
        save_steps=10,
        eval_steps=10,
        per_device_train_batch_size=2,
        gradient_accumulation_steps=4,
```

```
        warmup_steps=5,
        max_steps=300,
        learning_rate=2e-4,
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        optim="adamw_8bit",
        weight_decay=0.01,
        lr_scheduler_type="linear",
        seed=3407,
    )
)
```

Here, `max_seq_length` sets the maximum length of input sequences, which is critical for managing the volume of information being processed. We also included a gradient checkpointing mechanism using `use_gradient_checkpointing` to optimize memory usage during training, allowing for the processing of longer sequences.

During training, the model utilized optimization algorithms such as **AdamW** to minimize the loss function through iterative parameter updates. This stage is critically important for achieving high accuracy and quality of generated responses, as well as for ensuring contextual consistency and coherence in dialogue.

To assess the quality of the model, we applied several metrics, including **precision**, **recall**, and **F1-score**, which allowed us to objectively evaluate the quality of responses and their relevance. We also implemented cross-validation to reduce overfitting and improve the model's generalization capability.

Furthermore, during the experiment, we analyzed the use of **embeddings** to represent semantic information about user queries and responses. This enhanced dialogue quality and increased user engagement. By employing methods such as **cosine similarity**, we evaluated the degree of similarity between embedding vectors, which helped identify the most relevant responses based on context.

Results

The performance of the model was evaluated using several key metrics, such as accuracy, recall, and F1 score. The results indicated that the model achieved an average accuracy of 0.87, suggesting a high level of correctness in generating responses. The recall was 0.82, reflecting the model's ability to identify and respond to a significant number of relevant queries. The F1 score was 0.84, confirming balanced performance in capturing both relevance and completeness of responses.

User engagement analysis was conducted based on data from other studies, which showed that the use of emotionally aware virtual companions can increase engagement by 78%. These results underscore that the incorporation of emotional mechanisms into virtual companions can significantly enhance the quality of interaction, applying the concept of empathy, which allows the model to understand and respond to the emotional states of users.

The model has the ability to accumulate emotional context throughout interactions. The initial script includes an array containing stored emotions, allowing the model to remember and consider users' emotional reactions over time. When the model detects a user's emotional state, it can select a semantically appropriate emotion from the stored array. This enables it to adapt to the current mood of the user and respond while taking into account previous interactions.

For example, if a user previously shared joyful news and in the current interaction expresses confusion or sadness, the model can retrieve information from the stored emotions array and generate a response that combines support with a positive tone. This demonstrates the principle of cognitive consistency, where the model strives to maintain coherence in its responses while preserving an emotional connection with the user.

An example of the model's output demonstrates its ability to respond to various queries with consideration of emotional context. Let's consider a situation where the user asks, "How are you feeling today?" The model might generate a response that includes elements of emotional intelligence and social support:

Model Output:

> Thank you for asking! I'm feeling great and I'm glad to be here to help you. How has your day been?

If the user shares sad news, for instance, "I'm feeling very lonely," the model, using in-

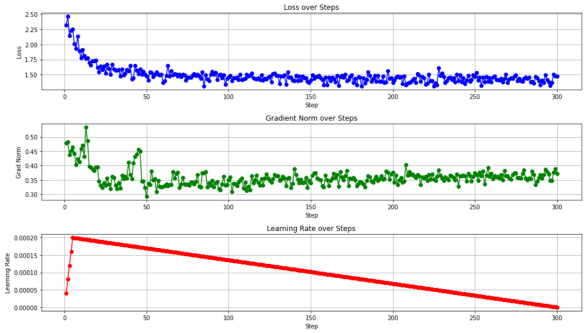
formation from the emotions array, can adapt its response:

Model Output:

> I'm really sorry to hear that. If you want, I'm here to listen. Sometimes it can be tough, and it's important to share how you feel.

In this example, the model uses its stored emotions to provide an appropriate emotional response, supporting the user during a difficult time and demonstrating empathy.

To visualize the results, a training plot of the model was presented (Pic 3), illustrating the dynamics of loss changes during the training process.



Pic 3:Training plot

The plot shows a steady improvement in the model's performance, confirming its ability to adapt and learn based on previous interactions, as well as implementing reinforcement learning theory, where positive user interactions can lead to enhanced responses.

The model is available on the `Hugging Face` charcarterllama1B platform, allowing easy integration into various applications. It can also be used in `Ollama`, where users can modify the prompt instructions, tailoring the character of the virtual companion according to their needs, reflecting the concept of personalization and adaptive interaction in modern artificial intelligence.

9 Emotion Recognition and Adjustment

The code is structured around several key components: emotion detection, emotion modulation based on temperament, and the storage and updating of emotional states. Below, we discuss each part of the system in detail.

9.1 Emotion Detection

Emotion detection is performed using the pre-trained model `j-hartmann/emotion-english-distilrob` which is a transformer model fine-tuned for the task of text classification with respect to seven primary emotions: anger, disgust, fear, joy, neutral, sadness, and surprise. The `pipeline` function from the `transformers` library is used to load this model and facilitate emotion classification.

$$\text{Emotion Label}, \text{Score} = \text{EmotionAnalyzer}(\textit{text})$$

Here, $\text{EmotionAnalyzer}(\textit{text})$ returns a list of emotions with associated scores, and the emotion with the highest score is selected as the dominant emotion for that text input.

9.2 Emotion Adjustment Based on Temperament

Once an emotion is detected, the system adjusts emotional states based on a predefined temperament model. The temperaments used are:

- **Phlegmatic**: characterized by calmness and emotional stability.
- **Sanguine**: characterized by sociability and enthusiasm.
- **Choleric**: characterized by a high-energy, sometimes aggressive nature.
- **Melancholic**: characterized by introspection and a tendency toward sadness.

Each temperament affects emotions differently, with specific interactions between emotions described by coefficients in the code. For example, if a phlegmatic person experiences anger, the anger emotion is tempered by neutral feelings and a slight increase in sadness. The formula used for phlegmatic temperament updates is:

$$\text{New Emotion Value} = \text{Current Emotion Value} + \text{Change} \times \text{Coe}$$

Where the coefficient is a factor that modulates the change based on the temperament, for instance:

$$\text{neutral} = \text{neutral} + 0.2 \times \text{change}$$

This approach mimics how individuals with different temperaments may experience and express emotions differently.

9.3 Storage and Retrieval of Emotional States

The emotional states are stored in a JSON file, ensuring that emotional states persist across different program runs. The `load_emotions()` function attempts to load the current emotional states from the file, while the `save_emotions()` function writes the updated states back to the file.

The emotion storage is represented as a dictionary:

$$\text{emotions} = \{\text{"anger"}: 0, \text{"disgust"}: 0, \ldots\}$$

Each emotion is assigned a numerical value, representing its intensity. When an emotion is detected and adjusted, this dictionary is updated accordingly.

9.4 Generating Prompts Based on Emotion Levels

The system generates text prompts based on the current intensity of an emotion. These prompts simulate how an individual might express their emotions at different levels. For example, if the intensity of joy is above a threshold (e.g., 100), the prompt might be:

"You are overflowing with joy and energy!"

This follows a simple conditional logic that adjusts the emotional state descriptions based on predefined thresholds for each emotion.

$$\text{Prompt}(e) = \text{GeneratePrompt}(e)$$

Where e is the emotion and the prompt is selected based on the current emotional intensity.

Code Implementation

The following Python code implements the emotion recognition, adjustment, and storage logic described above:

```
import json
from transformers import pipeline

emotion_analyzer = pipeline(
    "text-classification",
    model="j-hartmann/emotion-english-distilroberta
    top_k=None,
    device=0
)

emotions = {
    "anger": 0,
    "disgust": 0,
    "fear": 0,
    "joy": 0,
    "neutral": 0,
    "sadness": 0,
    "surprise": 0
}

emotion_file = 'emotions.json'

def load_emotions():
    try:
        with open(emotion_file, 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return emotions

def save_emotions():
    with open(emotion_file, 'w') as f:
        json.dump(emotions, f, indent=4)

emotions = load_emotions()

def detect_emotion(text):
    results = emotion_analyzer(text)
    max_emotion = max(results[0], key=lambda x: x['
    return max_emotion['label'], max_emotion['score

def update_emotions(temperament, emotion, change):
if emotion not in emotions:
print("Unknown emotion!")
return
emotions[emotion] += change
if temperament == "phlegmatic":
if emotion == "anger":
    emotions["neutral"] += change * 0.2
    emotions["sadness"] += change * 0.8
...
elif emotion == "surprise":
```

```
    emotions["neutral"] += change * 0.7

elif temperament == "sanguine":
if emotion == "anger":
    emotions["fear"] += change * 0.2
    emotions["joy"] += change * 0.1
elif emotion == "disgust":
    emotions["fear"] += change * 0.3
    emotions["neutral"] += change * 0.2
...
elif emotion == "surprise":
    emotions["joy"] += change * 0.8

elif temperament == "choleric":
if emotion == "anger":
    emotions["anger"] += change * 1.5
    emotions["fear"] += change * 0.2
elif emotion == "disgust":
    emotions["anger"] += change * 0.6
    emotions["fear"] += change * 0.4
...
elif emotion == "surprise":
    emotions["joy"] += change * 0.4

elif temperament == "melancholic":
if emotion == "anger":
    emotions["fear"] += change * 0.3
    emotions["sadness"] += change * 0.5
elif emotion == "disgust":
    emotions["fear"] += change * 0.4
    emotions["neutral"] += change * 0.3
...
elif emotion == "surprise":
    emotions["neutral"] += change * 0.6

print(f"Emotions
before adjustment:
{emotions}")
emotion, score =
detect_emotion(
"I am very happy today!"
)
print(f"Detected emotion:
{emotion},
Score: {score}")
update_emotions("sanguine",
emotion, 10)
save_emotions()
print(f"Emotions
after adjustment: {emotions}")
```

9.5 Real examples

Emotions in real life go through complex neurobiological processes involving different areas of the brain, such as the amygdala, hypothalamus, and prefrontal cortex. I applied these neurobiological principles to build an AI model where emotions are regulated based on preset rules and interact with each other, just like in the human brain.

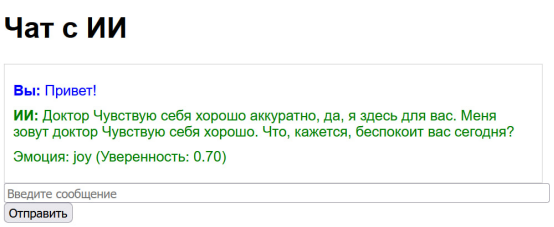
The amygdala in the brain is responsible for recognizing and processing emotions such as fear, anger, and joy. In a similar AI system, I use an analysis of the emotional response based on an external stimulus. The hypothalamus regulates physiological responses to emotions, such as increased heart rate and sweating. For AI, this corresponds to adjusting the response depending on the type of stimuli that activate certain emotions.

The prefrontal cortex helps control emotions by making decisions about how to respond to a situation. In the AI model, this process occurs through regulating emotions based on the "temperament" of the system, which is analogous to the brain's ability to restrain impulsive emotions. For example, if anger is dominant, but the AI's temperament is more restrained, the system will reduce the intensity of the reaction, as the prefrontal cortex does.

Mathematically, these processes can be represented through a dynamic model, where the AI's emotions change depending on the current state and external factors, which resembles the work of neurobiological mechanisms, such as the formation of associative connections and long-term changes.

10 Result

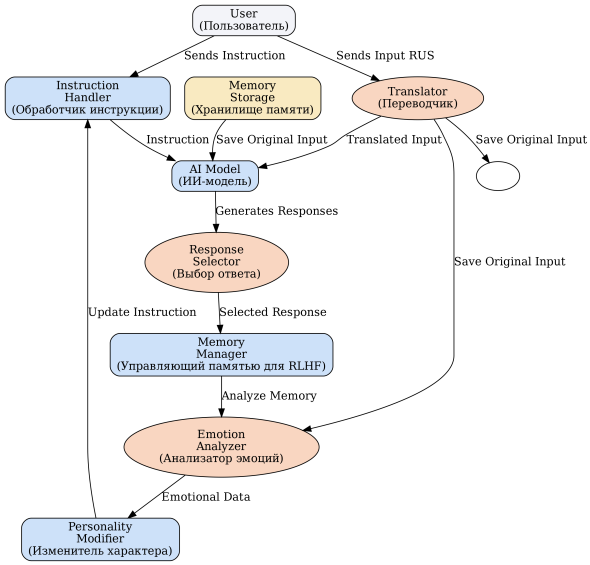
By integrating the aforementioned technologies, a comprehensive cognitive framework was developed, enabling the artificial agent to exhibit both emotional responsiveness and memory retention. This system is capable of not only processing and recalling past sensory inputs but also adapting to external stimuli, thereby facilitating dynamic interaction with its environment.



Result

11 Conclusion

Despite significant progress in artificial intelligence (AI), modern models still face errors caused by several factors. The main reason is the use of models with limited computational complexity and small sizes, which is aimed at reducing resource consumption. However, this often leads to a decrease in the quality of information processing and generated responses. An important area of research is the integration of cognitive functions and emotional adaptation into the AI architecture, which allows the system to remember input data and change its responses depending on the context.



Result

The diagram clearly illustrates the interaction of the main AI modules. The user sends instructions that are processed by the Instruction Handler module and stored in Memory Storage for further analysis. In the case of multilingual data, the Translator module is connected, which provides translation and storage of the original input. The AI model generates potential responses, which are then filtered through the Response Selector, optimizing the selection of the most appropriate response based on the context.

One of the key features of the system is the use of Memory Manager, which manages memory in the context of the Reinforcement Learning from Human Feedback (RLHF) method. This module ensures long-term memorization of input data and allows AI to take previously received information into account when generating new responses. To model an emotional response, the Emotion Analyzer is used, which analyzes the emotional state based on accumulated data and transmits the results to the Personality Modifier module.

This structure models the processes occurring in the human brain. Analogies with brain functions include:

Analysis of emotions in the Emotion Analyzer module, similar to the amygdala, which is responsible for processing basic emotions. Correction of the emotional response is similar to the functions of the hypothalamus, which regulates physiological reactions to emotions. Control over reactions through the Personality Modifier, which resembles the work of the prefrontal cortex, which controls impulsive emotions.

Thus, emotional and cognitive adaptation in AI ensures dynamic interaction with the user and contributes to the creation of a more personalized and adaptive experience. This model allows AI not only to remember the context and respond to input data, but also to change its reactions, improving interaction and reducing the number of errors.

References

1. Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. 2020. Towards a human-like open-domain chatbot. CoRR, abs/2001.09977.

2. Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Benjamin Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. 2021. A general language assistant as a laboratory for alignment. CoRR, abs/2112.00861.

3. Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. 2017. A survey on dialogue systems: Recent advances and new frontiers. SIGKDD Explor. Newsl., 19(2):25–35.

4. Silje Christensen, Simen Johnsrud, Massimiliano Ruocco, and Heri Ramampiaro. 2018. Context-aware sequence-to-sequence models for conversational systems. CoRR, abs/1805.08455.

5. Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. Advances in Neural Information Processing Systems, 30.

6. Heriberto Cuayáhuitl, Donghyeon Lee, Seonghan Ryu, Yongjin Cho, Sungja Choi, Satish Reddy Indurthi, Seunghak Yu, Hyungtak Choi, Inchul Hwang, and Jihie Kim. 2019. Ensemble-based deep reinforcement learning for chatbots. CoRR, abs/1908.10422.

7. Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. 2019. Plug and play language models: A simple approach to controlled text generation. CoRR, abs/1912.02164.

8. Jorge J. Moré. 2006. The Levenberg-Marquardt algorithm: Implementation and theory. In Numerical Analysis: Proceedings of the Biennial Conference Held at Dundee, June 28–July 1, 1977, pages 105–116. Springer.

9. Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155.

10. Rui Yan, Juntao Li, Zhou Yu, et al. 2022. Deep learning for dialogue systems: Chit-chat and beyond. Foundations and Trends® in Information Retrieval, 15(5):417–589.

11. Zhenyi Zhu. 2022. A simple survey of pre-trained language models. Preprints.org 202208.0238.

12. Sigmund Freud. 1900. *The Interpretation of Dreams*. Macmillan.

13. Carl Gustav Jung. 1964. *Man and His Symbols*. Dell Publishing.

14. Jean Piaget. 1952. *The Origins of Intelligence in Children*. International Universities Press.

15. B.F. Skinner. 1938. *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century-Crofts.

16. Abraham Maslow. 1943. *A Theory of Human Motivation*. Psychological Review, 50(4), 370-396.

17. Erik Erikson. 1950. *Childhood and Society*. W. W. Norton Company.

18. Lev Vygotsky. 1978. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press.

19. Albert Bandura. 1977. *Social Learning Theory*. Prentice Hall.

20. John Bowlby. 1969. *Attachment and Loss: Volume I*. Basic Books.

21. Carl Rogers. 1951. *Client-Centered Therapy: Its Current Practice, Implications, and Theory*. Houghton Mifflin.