DL Short Course Notes

Anri Lombard

2023

Table of Contents

LangChain for LLM Application Development

Introduction

The creator of LangChain is Harrison Chase, a former Harvard student.

LangChain is:

1. an open-source framework for building LLM applications with either Python or JavaScript
2. focussed on composition and modularity

It makes the process of making modular components for specific use cases easier.

Models, Prompts, and Parsers

Models refer to the LMM models that are used to generate text.

Prompts refer to the inputs that are passed into the model.

- Prompt templates allow for the reuse of good prompts to do specific tasks. Langchain has a set of template prompts that can be used.

Parsers refer to the output from the model that needs to be parsed into a format that can be used downstream.

- Prompt templates also support output parsing.
- Output is initially given as a long string, but can be parsed into a dictionary with LangChain.

Memory

Store past conversations in memory so that it has a more conversational flow.

- The way it stores memory is with ConversationBufferMemory.
- The LLM itself is stateless (it does not remember anything from the past). Wrapper code gives the full conversation so far as context to the LLM so it could generate a response. As a conversation gets longer the context gets longer and becomes expensive to provide. LangChain solves this with ConversationBufferMemory, which specifies in a parameter how many conversation steps should be remembered.

Memory Types

1. **ConversationBufferMemory**: allows for storing of messages and then extracts the message in a variable.
2. **ConversationBufferWindowMemory**: keeps list of interactions of the conversation over time and only uses last k interactions.
3. **ConversationTokenBufferMemory**: keeps buffer of recent interactions in memory and uses token length rather than number of interactions to determine when to flush interactions.
4. **ConversationSummaryMemory**: creates summary of the conversation over time.

Chains

Combines LLM with prompt.

- chain.run formats the prompt and feeds it to the LLM, then returns the result of the chain.
- The simple sequential chain is where there are multiple chains with a single input and single output each.
- For multiple inputs and/or outputs
- A router chain routes input to the correct chain

Question and Answer

LLMs can only inspect a few thousand words at a time. This is why we need to use embeddings and vector stores.

A **Vector store** is a database of vectors that can be queried for similar vectors.

An **Embedding** is a vector representation of a piece of text which allows us to compare it to other pieces of text.

A **Vector database** is a database of vectors that can be queried for similar vectors.

The **Stuff Method / Stuffing** is a method to simply stuff all data into a prompt as context to pass to the language model.

- Pros: It makes a single call to the LLM, then the LLM has access to all the data at once.
- Cons: Very large or many documents may cause a prompt to exceed the context length.

Additional methods:

1. Map_reduce: takes many chunks and their respective questions, then passes those to another language model to summarize the answers into a final answer.
    a. Very fast, but requires many calls
2. Refine: also takes the chunk approach, but builds upon the previous answer iteratively.
    a. Not as fast since answers are not independent, but rather depends on result of previous calls.
3. Map_rerank: single call for each document, then score the answers and pick the highest score as the final answer.

Evaluation

Applications using LLMs need to be evaluated with some criteria to test their accuracy.

One very effective way to do this is LLM-assisted evaluation (see notebook).

Agents

Langchain "agents" refer to a specific component of the Langchain system. An agent is essentially a stateless wrapper around an agent prompt chain, such as MRKL, and it plays a crucial role in the functioning of the system. The agent is responsible for formatting tools into the prompt, as well as parsing the responses that are obtained from the chat model. It receives user input and returns a response that corresponds to an "action" to take and a corresponding "action input"[1].

Furthermore, in some applications, an agent can be used when there's a need for not just a predetermined chain of calls to language models (LLMs) or other tools, but potentially an unknown chain that depends on the user's input. The agent has access to a suite of tools, and it can decide which, if any, of these tools to call based on the user input.

Conclusion

LangChain, as explored in this course by deeplearning.ai, empowers us to swiftly construct and deploy impressive applications powered by large language models (LLMs). It leverages the capabilities of state-of-the-art LLMs, combines them with unique functionalities like chains, agents, and custom prompts, and presents a streamlined interface for rapid development. While the course provides a solid introduction, it only scratches the surface of LangChain's full potential. There is a vast array of features and capabilities that remain to be discovered and harnessed, which could revolutionize the way we build and interact with LLM applications.

How Diffusion Models Work

Introduction

Diffusion models are a type of generative model used in machine learning. They are used to generate new samples from a learned data distribution. The term "diffusion" comes from the way these models work, which is similar to the process of diffusion in physics.

Here's a high-level overview of how diffusion models work:

1. **Data Distribution**: The model starts with a dataset, which represents a certain distribution of data. This could be anything from images of faces to text documents.
2. **Noise Addition**: The model then adds noise to this data, gradually transforming the data distribution into a known distribution, usually a standard Gaussian distribution. This process is akin to the diffusion process in physics, where particles spread out from a high concentration area to a low concentration area until a uniform distribution is achieved.
3. **Noise Reduction**: To generate new samples, the model starts with a sample from the known distribution (the Gaussian), and then gradually reduces the noise, transforming the sample back into the data distribution. The model learns to perform this noise reduction process through training.
4. **Generation of New Samples**: The end result is a new sample that should resemble the original data distribution. For example, if the model was trained on images of faces, it should be able to generate new images of faces that it has never seen before.

Diffusion models have been used to achieve state-of-the-art results in various tasks, such as image generation and text-to-speech synthesis. They are particularly notable for their ability to generate high-quality, diverse samples.

Intuition

1. **Forward Process (Data to Noise):** We start with a dataset of sprites. Each sprite can be thought of as a point in a high-dimensional space, where each dimension corresponds to a pixel's color value. The distribution of these points is the data distribution. The diffusion model then adds noise to these points, gradually transforming the data distribution into a standard normal distribution. This is akin to blurring the sprites until they become unrecognizable blobs of noise.
2. **Training the Model:** During training, the model learns how to perform this noise addition process. More specifically, it learns a series of transformations that can gradually add noise to the sprites to transform them into the standard normal distribution. The model is trained using a method called maximum likelihood estimation, which adjusts the model's parameters to maximize the likelihood of the observed data given the model.
3. **Reverse Process (Noise to Data):** Once the model is trained, it can generate new sprites by reversing the noise addition process. It starts with a sample from the standard normal distribution (a blob of noise), and then gradually reduces the noise, transforming it back into the data distribution. The model uses the transformations it learned during training to perform this noise reduction process. The end result is a new sprite that should resemble the original data distribution.
4. **Generating Novel Sprites:** By starting with different samples from the standard normal distribution, the model can generate a variety of different sprites. Because the model learned the data distribution from the original sprite dataset, the generated sprites should resemble the types of sprites in that dataset, but they will be novel in the sense that they are not exact copies of the sprites in the dataset.

Sampling

Here's a simplified version of the sampling process in a diffusion model:

1. **Initialize:** Start with a random sample from the standard normal distribution. This will be a high-dimensional vector where each dimension corresponds to a pixel's color value in the sprite. Let's call this vector **z**.
2. **Iterate:** For a certain number of steps (let's say **T** steps), do the following:
    1. **Noise Reduction:** Apply the reverse of the noise addition process to **z**. This involves applying a transformation that was learned during training, which reduces the amount of noise in **z**. Let's call this transformed vector **z'**.
    2. **Resampling:** Draw a new sample from a normal distribution that has **z'** as its mean. This new sample becomes the new **z** for the next step.
3. **Output:** After **T** steps, the vector **z** should have been transformed from a sample from the standard normal distribution into a sample from the data distribution. This is your generated sprite.

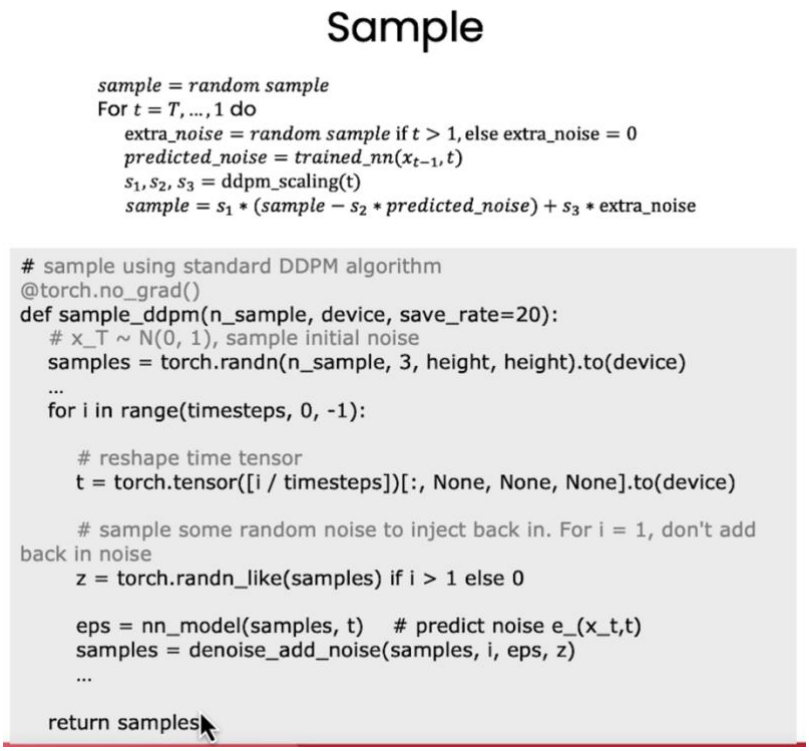

Figure 1: Sharon's algorithm for sampling

DDPM: denoising diffusion probabilistic models (arxiv.org/pdf/2006.11239.pdf)

Describing the **ContextUnet** function in the notebook:

- This code defines a U-Net architecture with context and time embeddings. U-Net is a type of convolutional neural network that is often used for image segmentation tasks. It has a symmetric encoder-decoder structure, which allows it to capture both local and global information about the input image.

  Here's a breakdown of the code:
    - The `ContextUnet` class is a subclass of PyTorch's `nn.Module` class, which means it represents a neural network module. It has several instance variables that represent different layers of the network.
    - The `__init__` method initializes these layers. The `in_channels` parameter is the number of channels in the input image, `n_feat` is the number of feature

maps in the intermediate layers, `n_cfeat` is the number of context features, and `height` is the height of the input image.
    - The `forward` method defines the forward pass of the network. It takes an input image `x`, a time step `t`, and an optional context label `c`, and returns the output of the network.
    - The network consists of an initial convolutional layer, a down-sampling path (which reduces the spatial dimensions of the input), an up-sampling path (which increases the spatial dimensions back to the original size), and a final convolutional layer that maps the output to the same number of channels as the input.
    - The down-sampling path and up-sampling path are each composed of several layers, including convolutional layers, normalization layers, activation functions, and up-sampling or down-sampling operations.
    - The network also includes embeddings for the time step and context label, which are learned during training. These embeddings are added to the feature maps at different stages of the up-sampling path, allowing the network to incorporate information about the time step and context into its output.
    - The `forward` method also includes some logic for handling the case where the context label is not provided. In this case, it creates a tensor of zeros as a placeholder for the context label.
    - The output of the network is a tensor with the same number of channels as the input image, which could be used for tasks like image segmentation or image-to-image translation.
- This U-Net architecture is a powerful model for image processing tasks, and the addition of context and time embeddings allows it to handle more complex tasks that involve additional information beyond just the input image.

Why we add additional noise scaled based upon what time step we are at, at each step:

Neural Network

U-Net is a type of convolutional neural network (CNN) that was originally developed for biomedical image segmentation. It's named U-Net because of its U-shaped architecture.

The U-Net architecture consists of two parts: an encoder (or contracting path) and a decoder (or expanding path), which give it the "U" shape.

1. E**ncoder**: The encoder part of the U-Net is similar to a traditional CNN. It consists of a series of convolutional layers and pooling layers. The convolutional layers are used to extract features from the input image, while the pooling layers are used to reduce the spatial dimensions of the image,

increasing the network's field of view. As you go deeper into the encoder, the spatial dimensions of the feature maps decrease, but the number of feature maps increases, allowing the network to learn more complex features.

2. **Decoder**: The decoder part of the U-Net is used to upsample the feature maps back to the original input size. This is typically done using transposed convolutions or upsampling operations. The decoder also includes skip connections from the encoder to the decoder. These skip connections pass the feature maps from the encoder directly to the decoder, which helps the network to recover the spatial information that was lost during the downsampling process in the encoder.
3. **Final Layer**: The final layer of the U-Net is a convolutional layer that maps the output of the decoder to the desired number of output channels. For example, in a segmentation task, the number of output channels would be equal to the number of classes.

The U-Net architecture is particularly well-suited for segmentation tasks because it combines the ability of a CNN to extract complex features from an image with the ability to maintain spatial information about where those features are located. The skip connections allow the network to use both local and global information about the image, which can help to improve the quality of the segmentation.

An **embedding** is a learned representation of some input data in a lower-dimensional space. They are often used to convert categorical data, such as words or labels, into a form that can be processed by a neural network. A U-Net takes in additional information in the form of time step and context label embeddings. These embeddings are learned during training and are used to incorporate additional information into the network's output.

Here's how it works:

1. **Time Step Embedding:** The time step embedding is a learned representation of the time step. It's created by passing the time step through a fully connected layer, which is a type of layer that can learn to map its input to any output. The learned time step embedding is then reshaped and added to the feature maps at different stages of the up-sampling path.
2. **Context Label Embedding:** The context label embedding is a learned representation of the context label. Like the time step embedding, it's created by passing the context label through a fully connected layer. The learned context label embedding is then reshaped and added to the feature maps at different stages of the up-sampling path.

By adding these embeddings to the feature maps, the network is able to incorporate information about the time step and context into its output. This can be useful for tasks where the output depends not only on the input image, but also on some additional context or time information.

The UNet can take in more information in the form of embeddings

- Time embedding: related to the timestep and noise level.
- Context embedding: related to controlling the generation, e.g. text description or factor (more later).

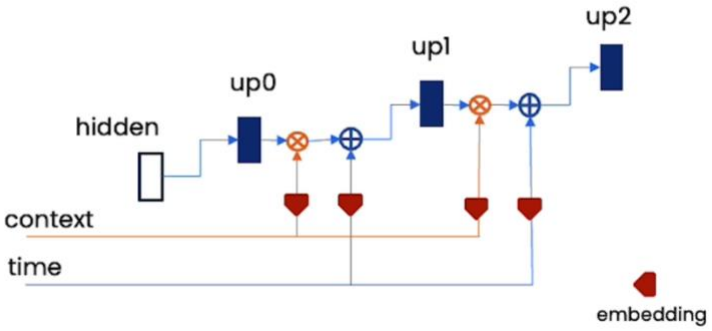

Figure 2: Embeddings in a U-Net

Training

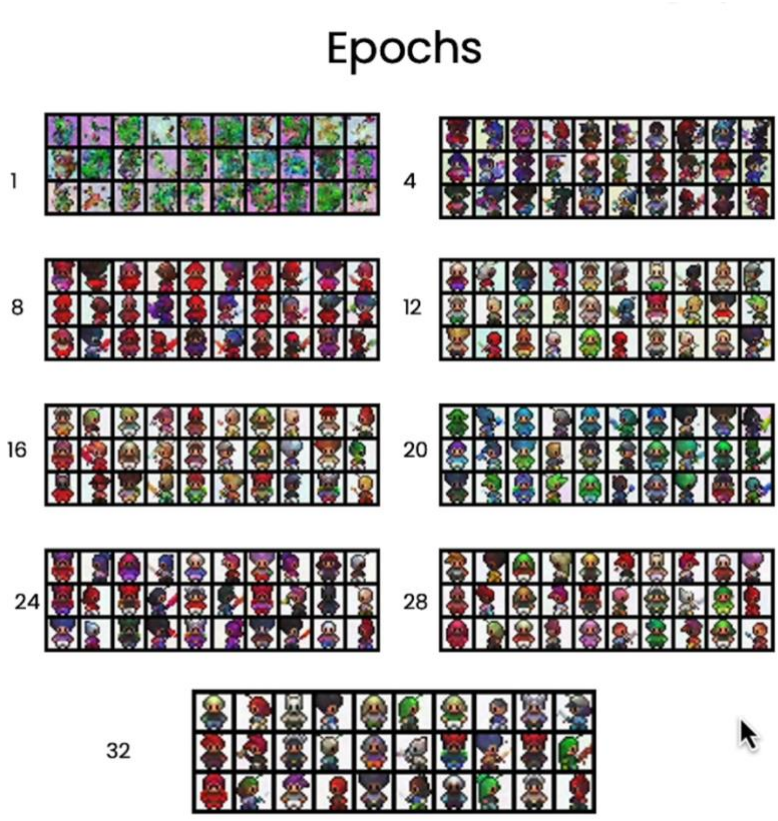

Figure 3: More epochs lead to more refined predictions

- Sample training image.
- Sample timestep t. This determines the level of noise.
- Sample the noise.
- Add noise to image.
- Input this into the neural network. Neural network predicts the noise.
- Compute loss between predicted and true noise.
- Backprop & learn!

```
for ep in range(n_epoch):

    # linearly decay learning rate
    optim.param_groups[0]['lr'] = lrate*(1-ep/n_epoch)

    pbar = tqdm(dataloader, mininterval=2 )
    for x, _ in pbar:   # x: images
        optim.zero_grad()

        # perturb data
        t = torch.randint(1, timesteps + 1, (x.shape[0],)).to(device)
        noise = torch.randn_like(x)
        x_pert = perturb_input(x, t, noise)

        # use network to recover noise
        pred_noise = nn_model(x_pert, t / timesteps)

        # loss is mean squared error between the predicted and true noise
        loss = F.mse_loss(pred_noise, noise)
        loss.backward()
        optim.step()
```

Figure 4: Algorithm for training neural network

Controlling

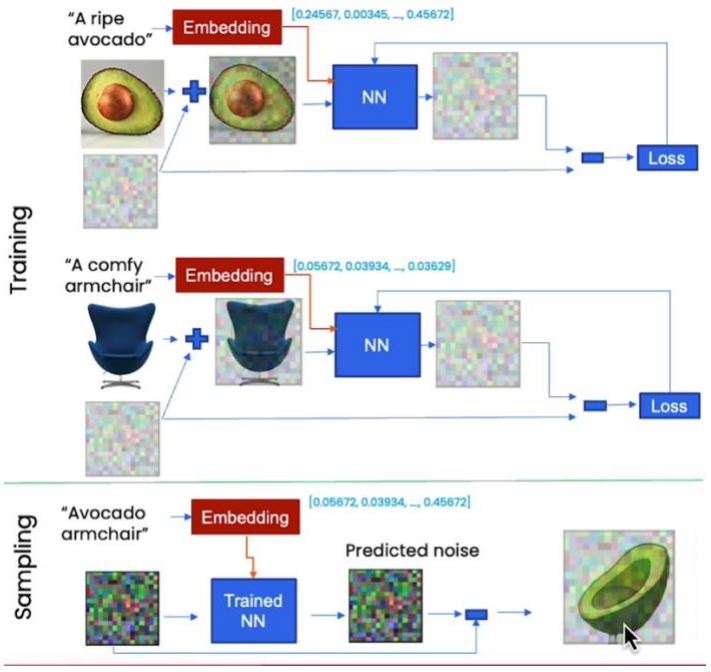

Figure 5: Adding context is useful for controlling generation

Speeding Up

A much faster sampling method than DDPM is DDIM: 2010.02502.pdf (arxiv.org)

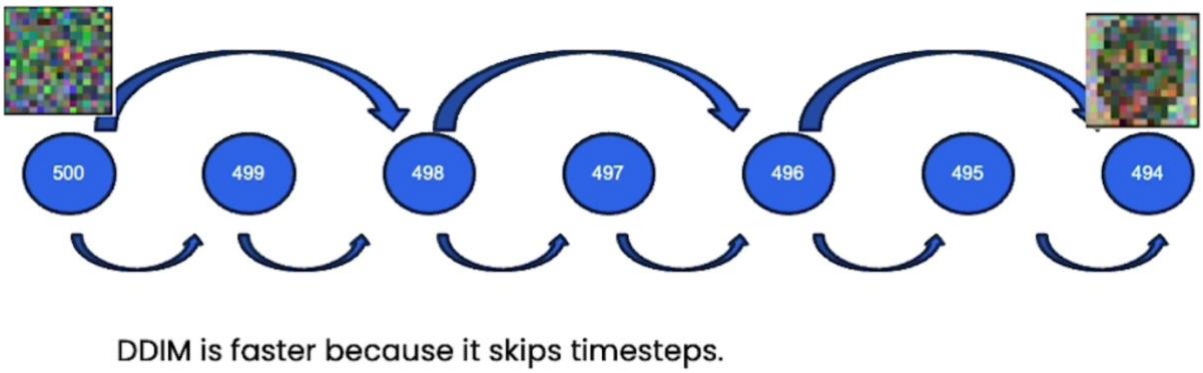

Figure 6: DDIM breaks the Markov assumption since it removes the randomness and is deterministic

The **Markov assumption** refers to the idea that each step in the diffusion process is independent of the previous steps, given the current state. In other words, the future state of the system depends only on the current state and not on how it got there.

Traditional diffusion models follow this Markov assumption. They generate new samples by starting with a sample from the noise distribution and then applying a series of independent noise-reduction steps. The noise at each step is typically sampled from a random distribution, which introduces randomness into the process.

DDIM, on the other hand, breaks this Markov assumption. Instead of applying a series of independent noise-reduction steps, it uses a deterministic process to generate new samples. This is achieved by using a denoising function that depends on the entire trajectory of the process, not just the current state. This removes the randomness from the process and makes it deterministic.

The advantage of DDIM is that it can generate high-quality samples more efficiently than traditional diffusion models. By taking into account the entire trajectory of the process, it can make more informed decisions about how to reduce the noise at each step, which can lead to better results.

Although DDIM is significantly faster, the results may be of a lower quality than DDPM that runs at 500 steps. It has been found that DDIM performs better for steps lower than 500, although DDPM outperforms DDIM at more than 500 steps.