Data 188 Introduction to Deep Learning
Spring 2026 Eric Kim

Discussion 11

This discussion covers masked autoencoders and recommendation systems.

1. Supervised vs. Self-Supervised Learning

(a) As discussed in this class, what is the difference between a supervised learning task vs. a self-supervised learning task?

Give an example of a supervised learning task and an example of a self-supervised learning task for both the computer vision and NLP domain.

Solution: A supervised learning task is one where we have explicit labels for each input, generally where the labels come from some external source (ex: human labelers).

For a self-supervised task, the labels instead come directly from the data.

An example of supervised learning for both the computer vision and the NLP domain would be classification. An example of self-supervised learning for computer vision would be the "fill in the image patch" task, and an example of self-supervised learning for the NLP domain would be the "fill in the blank" (aka "cloze") task.

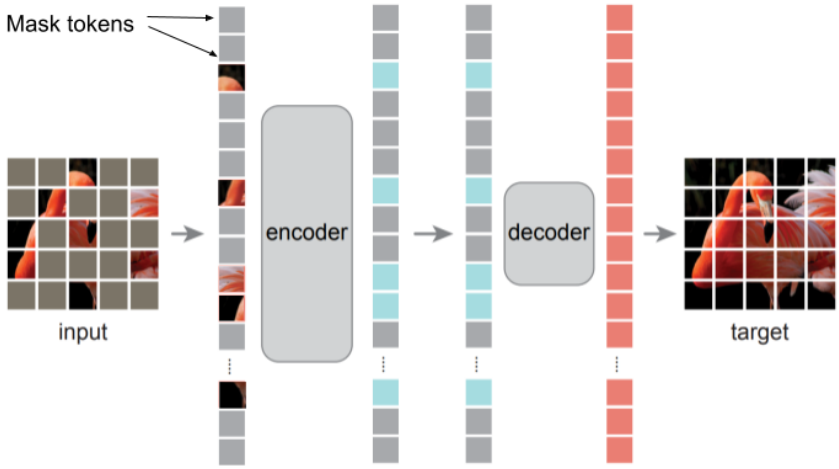
(b) Suppose we had an audio dataset consisting of millions of songs from a diverse range of genres and artists. Describe a self-supervised learning task to learn a strong audio-clip embedding representation using an audio encoder model (say, an audio-clip embedding of duration 5 seconds).

Solution: We can apply a "fill in the blank" task for an audio auto-encoder, where we take a region of audio (say, a 5 second clip), randomly blank out part of it (say, 1 second), and have the autoencoder recover the missing clip.

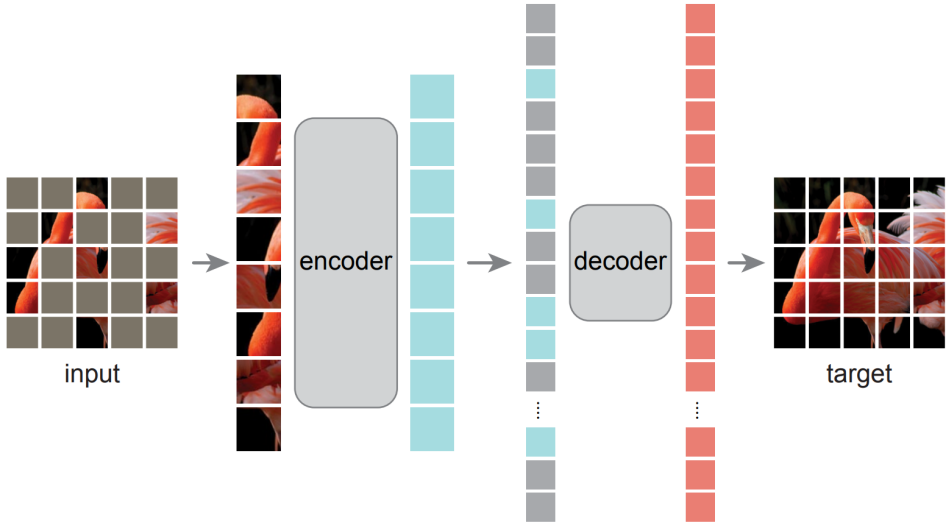
2. Encoder and Mask Tokens

In the masked autoencoder (MAE) architecture, recall that we employ the "fill in the image patch" self-supervised task, using a transformer encoder-decoder as our autoencoder.

Consider the following realization of this idea, where we pass all image patches to the encoder (i.e., both pink and gray boxes):



Compare it to the following approach that the MAE paper actually took:



Why do you think the authors chose to omit the masked image patches to the encoder? Recall that the authors found that an aggressive masking percentage (masking 75% of the image) performed well.

Solution:

By reducing the encoder input sequence by 75%, we significantly reduce the encoder computation cost (recall that transformer computation cost is quadratic in input sequence length).

This approach also avoids a train/test input drift. If we pass the masked image patches (i.e., mask tokens) to the encoder during training, then the encoder model will learn that most of its inputs are mask tokens and adjust accordingly.

However, during test time, we're shifting tasks: we're passing completely-unmasked images and doing, say, ImageNet-1k image classification. Now, the encoder is seeing NO mask tokens, which is a huge shift in input distribution, so it would perform worse on the classification task if we use the approach in the first diagram.

3. Distributed Training

It's rumored that GPT-4 has on the order of 1.8 trillion model parameters. If we stored them as the float16 data type (2 bytes per parameter), this would require 3600 GB GPU memory.

For comparison, in 2026 one of the most powerful GPU servers that AWS EC2 offers is the p5en.48xlarge, that (with 8 H200 GPU's) has a combined total of 1128 GB GPU memory, and costs around $554K per year to rent. Suppose we don't have access to these machines, and we have to instead settle for machines that only have 384 GB GPU memory (ex: g6e.48xlarge).

Which distributed training technique would be most appropriate for us to train a GPT-4 model: Distributed Data Parallel (DDP), or Fully Sharded Data Parallel (FSDP)?

Solution: FSDP. Since 3600 GB exceeds the amount of GPU memory available on a single machine, we must split ("shard") the model parameters across multiple machines.

4. Two-Stage Recommendation System

Recall that, during lecture, we covered a two-stage recommendation system architecture, where we first have a lightweight retrieval component ("candidate generation"), followed by a heavier-duty ranking component that ultimately produces the final results.

Why do we need this two-stage architecture? Suppose we discarded the first stage ("candidate generation"), and used my ranking model to rerank my entire corpus. Why is this usually a bad idea? When might this be OK?

Solution: The two-stage architecture is needed for scalability. For many tech companies, their item corpus is on the order of billions, so it's highly impractical to run an ML ranking model on every single item in the corpus for every user query.

Discarding the candidate generator means rather than passing in hundreds (or thousands) of candidates to the ML ranker, we're now passing in billions of candidates, which will take forever to finish processing. This is a bad idea.

However, if your item corpus is very small (say, a few hundred items), then you can get away with skipping the candidate generation stage.

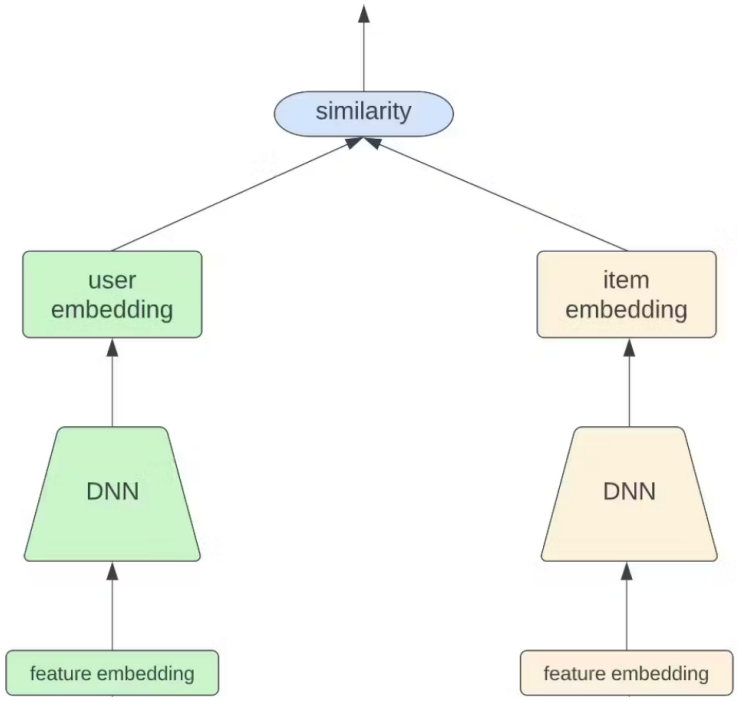
5. Two Tower Models

Suppose we manage a photosharing app where users engage with photos (like, comment, share) that other users have uploaded.

Assume that the app has been running long enough (and with enough users) that we have plentiful user engagement data.

We would like to train a model that, given a User and a query image, outputs the probability that the User will like the image. We want our User and Item embeddings to be 64-dimensional. Use the cosine similarity function as our similarity function.

Describe how to build a two-tower model that achieves this. Be specific about the model architecture details. How would you generate the training dataset? Use this diagram from lecture as reference:



Assume we have the following User features: `int age`, `int account_age`, `string country`, `string gender`, `string favorite_photo_category`.

Solution: The two-tower model would have a User tower and an Item (image) tower.

To featurize ("tensorize") the User features, we'd convert the categorical features (ex: `country`, `gender`, `favorite_photo_category`) into one-hot encoded vectors. We'd also preprocess the numerical features (`age`, `account_age`) using either normalization or bucketing.

Then, we'd pass this User features tensor to an MLP, say a 4-layer MLP, where the final output dimension is `d = 64`.

To featurize the Item (image), we would pass the image to either a ConvNet or a Visual Transformer (ViT). The output Item embedding would depend on the model type: ConvNet: as discussed in lecture, use the embedding after the pooling layer, but before the Linear layer(s) producing the logits.

Visual Transformer: either use the CLS token embedding (if available), or an aggregation/pooling of all output encoder tokens.

Given that the image model (ConvNet, ViT) may not output its embedding as 64-dimensional (ex: ResNet-34 outputs the embedding as a 512-dimensional embedding), we can add another Linear projection layer to project to our desired 64-dimensional embedding size.

Finally, the User and Item embeddings are passed to the similarity function (cosine similarity), and the similarity score is passed to the BinaryCrossEntropyLoss.

To generate the training data, we scrape user engagement data to generate pairs like:
Positive examples: when User clicked on Item $\rightarrow$ (User, Item)
Negative examples: randomly sample Items not engaged with $\rightarrow$ (User, Item)

Contributors:

- Eric Kim, Andria Xu.

