Assignment 1
COL775: Deep Learning. Semester II, 2023-2024.
Due Date: March 30, 2024

March 16, 2024

PART 1

1 ResNet over Convolutional Networks and different Normalization schemes

Residual Networks (ResNet) [5] present a very simple idea to introduce identity mappings via residual connections. They are shown to significantly improve the quality of training (and generalization) in deeper networks. We covered the core idea in class. Before starting this part of the assignment, you should thoroughly read the ResNet paper. Specifically, we will implement the ResNet [5] architecture, and study the effect of different normalisation schemes, viz. Batch Normalization [6], Instance Normalization [12], Batch-Instance Normalization [12], Layer Normalization [2], and Group Normalization [13] within ResNet. We will experiment with a dataset on Indian Birds species classification.

1.1 Image Classification using Residual Network

This sub-part will implement ResNet for Image Classification.
Dataset: link

1. You will implement a ResNet architecture from scratch in PyTorch. Assume that the total number of layers in the network is given by 6n+2. This includes the first hidden (convolution) layer processing the input of size 256×256. This is followed by n layers with feature map size 256×256, followed by n layers with feature map size 128×128, n layers with feature map size given by 64 × 64, and finally a fully connected output layer with r units, r being number of classes. The number of filters in the 3 sets of n hidden layers (after the first convoluational layer) are 16, 32, 64, respectively. There are residual connections between each block of 2 layers, except for the first convolutional layer and the output layer. All the convolutions use a filter size of 3 × 3 inspired by the VGG net architecture [10].

Whenever down-sampling, we use the convolutional layer with stride of 2. Appropriate zero padding is done at each layer so that there is no change in size due to boundary effects. The final hidden layer does a mean pool over all the features before feeding into the output layer. Refer to Section 4.2 in the ResNet paper for more details of the architecture. Your program should take n as input. It should also take r as input denoting the total number of classes.

2. Train a ResNet architecture with n = 2 as described above on the given dataset. Use a batch size of 32 and train for 50 epochs. For dataset, r = 25. Use SGD optimizer with initial learning rate of 10^{-4}. Decay or schedule the learning rate as appropriate. Feel free to experiment with different optimizers other than SGD.

3. Train data has 30,000 images while validation has 7500 images each image is of different resolution. Report the following statistics / analysis:

   - Accuracy, Micro F1, Macro F1 on Train, Val and Test splits.
   - Plot the error, accuracy and F1 (both macro and micro) curves for both train and val data

2 Impact of Normalization

The standard implementation of ResNet uses Batch Normalization [6]. In this part of the assignment, we will replace Batch Normalization with various other normalization schemes and study their impact.

1. Implement from scratch the following normalization schemes. They should be implemented as a sub-class of torch.nn.Module.

   (a) Batch Normalization (BN) [6]

   (b) Instance Normalization (IN) [12]

   (c) Batch-Instance Normalization (BIN) [7]

   (d) Layer Normalization (LN) [2]

   (e) Group Normalization (GN) [13]

2. In your implementation of ResNet in Section 1.1, replace the Pytorch's inbuilt Batch Normalization (nn.BatchNorm2d) with the 5 normalization schemes that you implemented above, giving you 5 new variants of the model. Note that normalization is done immediately after the convolution layer. For comparison, remove all normalization from the architecture, giving you a No Normalization (NN) variant as a baseline to compare with. In total, we have 6 new variants (BN, IN, BIN, LN, GN, and NN).

3. Train the 6 new variants on the dataset, as done in Section 1.1

4. As a sanity check, compare the error curves and performance statistics of the model trained in Section 1.1 with the BN variant trained in this part. It should be identical (almost).

5. Compare the error curves and performance statistics (accuracy, micro F1, macro F1 on train / val / test splits) of all six models.

6. **Impact of Batch Size:** [13] claim that one of the advantages of GN over BN is its insensitivity to batch size. Retrain the BN and GN variants of the model with Batch Size 8 and compare them with the same variants trained with Batch Size 128. Note that reducing the batch size will significantly increase the training time. To reduce the time taken, run it for 50 epochs and early stop based on validation accuracy.

7. **Visualize model's thinking:** An important part of any model training is analysing why a model predicts any particular label for a given input. In this sub-part we'll use Grad-CAM [9] for producing visual-explanations from our ResNet models. Grad-CAM computes the gradient of input feature map with respect to the specified output class. Higher the gradient value at particular position indicates it's importance for predicting particular class. For more information please read Grad-CAM paper[9]. For this assignment use the Grad-CAM's implementation from this package.

    - Consider following 7 classes for visualization: Cattle Egret , Coppersmith Barbet, Indian Peacock, Red Wattled Lapwing, Ruddy Shelduck, White Breasted Kingfisher, White Breasted Waterhen. Use the best performing model from all the variants trained above, visualize the gradient maps (superimposed on image) of 5 correctly and 5 incorrectly classified images for each of the above class from the validation set. Gradient needs to be computed with respect to the correct class and for the output feature map from last $2n$ layers (where feature map size is 64x64). These gradient maps need to be mean aggregated to give final 64x64 gradient map. The given package does all of these by default and you just need to mention the model layers.
    - Analyse the gradient maps and report your observations.

PART 2

Text to Math program

In Math Word Problem solving[3], given a mathematical problem specified in text, the goal is to find the solution to the problem by applying mathematical reasoning on the input text. See Figure 1 for an example.

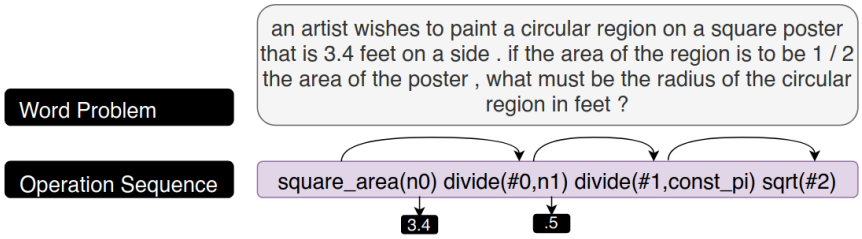

Figure 1: Sample data

Answering math word problems is a difficult task, and requires logical reasoning over implicit or explicit quantities expressed in text. In this assignment, our goal is develop an automatic *converter*, which would convert the textual narrative into executable program. One of the ways to generate executable programs from their textual description is to make use of **Encoder-Decoder** architecture. For this part of assignment, we will make use of seq2seq architectures to convert input text into the corresponding symbolic form. We will be using *operation representation language* provided in [1] to represent the executable program.

Note that training recurrent architectures like LSTMs can be slightly tricky. Therefore, look into various tricks that people have employed for this purpose. Some of them can be found in [11]. Try to optimize your training pipeline as much as possible.

2.1 Data

Dataset and checker file link
The folder contains the train, test and dev json files along with a checker file. Each json contains a $Program$, $linear\_formula$ and $answer$ fields for each example. The train split contains 19791 samples, dev contains 2961 and test contains 1924 samples. We would be generating the $linear\_formula$ given the $Problem$, the example of same is given in 1.

2.2 Metrics

Evaluation metrics are critical for any system to be evaluated. For this task we will use 2 different metrics.

- **Exact Match :** If all the predicted sequence matches the ground-truth sequence exactly it's 1 else 0.
- **Execution Accuracy :** If the executed answer of your predicted sequence is within 2% of true answer then the score is 1 else 0. You are provided

with the an evaluation script in the dataset. You must run this script on generated outputs to get the evaluation scores. The script will do the substitution of variables in the predicted sequence and will also execute the sequence to generate answer. Please read the checker file for more details.

2.3 Models

For your experiments you are supposed to train the following models.

2.3.1 Architectural experiments

1. A Seq2Seq model with GloVe embeddings [8], using an Bi-LSTM encoder and an LSTM decoder. For details of an LSTM refer [4]
2. A Seq2Seq+Attention model with GloVe embeddings, using an Bi- LSTM encoder and an LSTM decoder.
3. A Seq2Seq+Attention model using a pre-trained frozen BERT-base-cased encoder and an LSTM decoder
4. A Seq2Seq+Attention model with pre-trained BERT-base-cased encoder and an LSTM decoder, where the BERT encoder can now be fine-tuned along with the remaining network.

   You must report the following:

   - Loss curves on the training and dev set
   - Exact Match Accuracy and Execution Accuracy (see evaluation metrics) on the dev/test set.
   - All hyper-parameter settings used in your experiments.
   - Any findings observed from any potential grid search performed
   - Any other insights that you gained from the training procedure and the outputs of your model.

In this part, we will implement the models with **teacher forcing** with a ratio of **0.6** during training. During inference we will use **beam search** with **k = 10** to generate output. You must implement beam search in python/pytorch from scratch.

2.3.2 Effect of Teacher Forcing probability

Train the second model (Seq2Seq+Attention BiLSTM-LSTM) again with teacher forcing probabilities in $\{0.3, 0.6, 0.9\}$. Report the exact match accuracy and token match accuracy on test data with beam size $k = 10$. Comment your observations.

2.3.3 Effect of Beam Size

For the fourth model (Seq2Seq+Attention with fine-tuned Bert) trained with teacher forcing probability 0.6, generate the output for test data with beam sizes $\{1, 10, 20\}$. Report the exact match accuracy and token match accuracy on test data. Comment your observations.

References

[1] Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms, 2019.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[3] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[7] Hyeonseob Nam and Hyo-Eun Kim. Batch-instance normalization for adaptively style-invariant neural networks, 2019.

[8] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[9] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, October 2019.

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[11] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[12] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization, 2017.

[13] Yuxin Wu and Kaiming He. Group normalization, 2018.

Assignment 2
COL775: Deep Learning. Semester II, 2023-2024.
Due Date: ~~May 6, 2024~~ May 7, 2024

April 5, 2024

The dataset for this assignment can be found from `/scratch/cse/dual/cs5190448/A2_dataset` on HPC.

PART 1

1 Object-Centric Learning With Slot Attention

Neural models are considered as black box methods since they often work with raw data as input, and implement complex function manipulations over them, to get the final output, without much interpretability of what happens at intermmediate steps. For image related tasks, recent literature [**?**] has discovered that having object-centric representation in latent space not only improves the interpretability but also improves the performance of models. Slot attention [3] is one of the methods to learn object centric representations from images. In this assignment you will implement slot attention from scratch.

1.1 Slot Attention

Slot attention was proposed in the paper [3]. And is defined as follows: "The Slot Attention module maps from a set of N input feature vectors to a set of K output vectors that are refer to as slots. Each vector in this output set can, for example, describe an object or an entity in the input. Slot Attention uses an iterative attention mechanism to map from its inputs to the slots. Slots are initialized at random and thereafter refined at each iteration t = 1 . . . T to bind to a particular part (or grouping) of the input features." Read the original slot attention paper [3] for more information. A pseducode for algorithm is shown in figure 1.

- More formally given image $x \in \mathbf{R}^{H \times W \times 3}$, a CNN encoder $\mathcal{E}$ gives feature maps $\mathcal{E}(x) = z \in \mathbf{R}^{\sqrt{N} \times \sqrt{N} \times D_{inputs}}$.

Algorithm 1 Slot Attention module. The input is a set of N vectors of dimension $D_{\texttt{inputs}}$ which is mapped to a set of K slots of dimension $D_{\texttt{slots}}$. We initialize the slots by sampling their initial values as independent samples from a Gaussian distribution with shared, learnable parameters $\mu \in \mathbb{R}^{D_{\texttt{slots}}}$ and $\sigma \in \mathbb{R}^{D_{\texttt{slots}}}$. In our experiments we set the number of iterations to $T = 3$.

Input: $\texttt{inputs} \in \mathbb{R}^{N \times D_{\texttt{inputs}}}$, $\texttt{slots} \sim \mathcal{N}(\mu,\ \mathrm{diag}(\sigma)) \in \mathbb{R}^{K \times D_{\texttt{slots}}}$
Layer params: $k,\ q,\ v$: linear projections for attention; `GRU`; `MLP`; `LayerNorm` (x3)
  $\texttt{inputs} = \texttt{LayerNorm}\,(\texttt{inputs})$
  for $t = 0 \ldots T$
    $\texttt{slots\_prev} = \texttt{slots}$
    $\texttt{slots} = \texttt{LayerNorm}\,(\texttt{slots})$
    $\texttt{attn} = \texttt{Softmax}\,(\frac{1}{\sqrt{D}} k(\texttt{inputs}) \cdot q(\texttt{slots})^T,\ \texttt{axis='slots'})$ # norm. over slots
    $\texttt{updates} = \texttt{WeightedMean}\,(\texttt{weights=attn} + \epsilon,\ \texttt{values=}v(\texttt{inputs}))$ # aggregate
    $\texttt{slots} = \texttt{GRU}\,(\texttt{state=slots\_prev},\ \texttt{inputs=updates})$ # GRU update (per slot)
    $\texttt{slots} \mathrel{+}= \texttt{MLP}\,(\texttt{LayerNorm}\,(\texttt{slots}))$ # optional residual MLP (per slot)
  return `slots`

Figure 1: Pseudo Code for slot attention

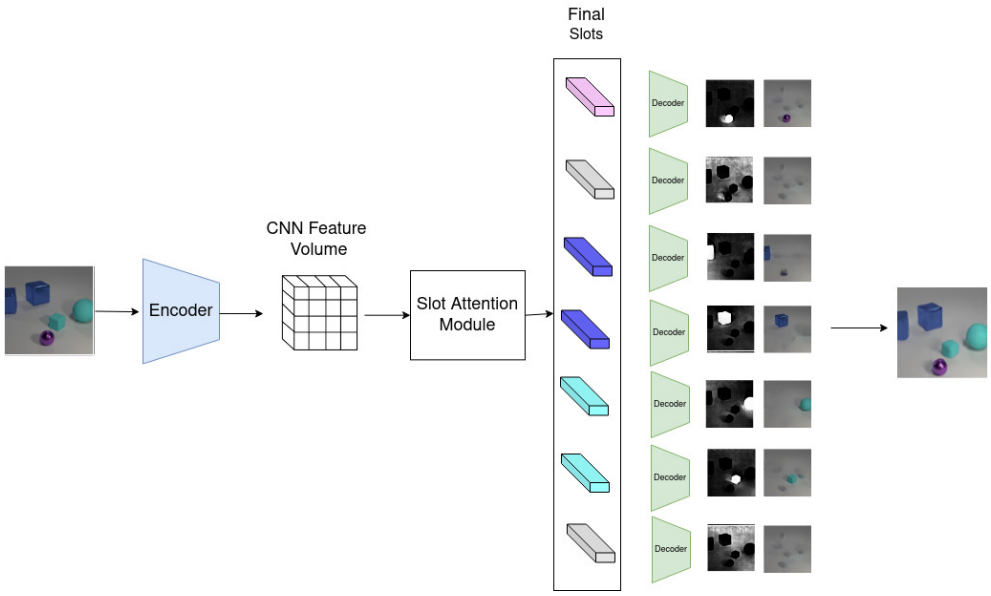

Figure 2: High level diagram

- A 2D positional encoding $P \in \mathbf{R}^{\sqrt{N} \times \sqrt{N} \times D_{inputs}}$ is added to z,

$$z' = z + P$$

- z' is flattened spatially to get N input vectors. These are passed through `MLP` to get final $\texttt{inputs} \in \mathbf{R}^{N \times D_{inputs}}$

$$\texttt{inputs} = \texttt{MLP}(\texttt{Flatten}(z'))$$

- Then K slot vectors, $\texttt{slots} \in \mathbf{R}^{K \times D_{slots}}$ are initialized by sampling them from normal distibution $\mathcal{N}(\mu, \text{diag}(\sigma))$, where both μ and σ are learnable. Optionally on complex datasets it has been found beneficial to use learnable initialization of slots, that is you initialize slots with learnable K vectors rather than sampling. [10]

- Then `slots` attend to `inputs` itertively for T times and updated `slots` at the end of each iteration. For detailed steps refer to pseudo code 1.

- Each of K vectors from `slots` are decoded with **Spatial Broadcast Decoder** [9], $\mathcal{D}$ to produce 4 channeled image. Denote $\texttt{output}_i = \mathcal{D}(\texttt{slots}[i])$, note that $\texttt{output}_i \in \mathbf{R}^{H \times W \times 4}$

- To get the final image, first K masks are generated one for each slot by taking softmax for each pixel of first channel, across slots.

$$\texttt{masks}_i = \texttt{Softmax}(\{\texttt{output}_j[:, :, 0]\}_{j=1}^{K})_i$$

- The final reconstructed image is obtained as

$$\hat{x} = \sum_{i=1}^{K} \texttt{masks}_i \cdot \texttt{content}_i$$

  where $\texttt{content}_i = \texttt{output}_i[:, :, 1:]$

- You backpropagate on the image reconstruction loss for batch of size B

$$\mathcal{L}(\theta) = \frac{1}{BHW} \sum_{i=1}^{B} (x^{(i_b)} - \hat{x}^{(i_b)})^2$$

These are spatially flattened to get N input vectors $\texttt{inputs} \in \mathbf{R}^{N \times D_{inputs}}$.

1.2 Experiments

You're provided with a sub-sampled CLEVRTex dataset which comprises of synthetic images of objects with different attributes, locations and properties. Train the Slot-Attention based network as shown in 2 on this data and visualize the generated masks and the reconstructed images. **We recommend using similar hyper-parameters to [3] [10]. However you're free to experiment with other values as well.** Use number of slots $K = 11$ for this experiment.

- Report the Adjusted Rand Index (ARI) score between the ground-truth and predicted object segmentation masks on the val split.
- Plot the train and val image reconstruction loss vs epochs. To save training time you may choose validate after certain number of epochs rather than every epoch.
- **Compositional Generation**: Create a slot library using the training data, apply K-means clustering with K being the number of slots in the trained model. Sample new slots from each of the cluster to generate an image with compositional attributes. Generate N_{val} such images, where N_{val} is number of validation images and report the clean-fid [4] metric using the validation images as ground truth.

PART 2

2 Slot Learning using Diffusion based Decoder

In this part you'll implement a conditional diffusion model to decode the image conditioned on the slots, to replace Spatial Broadcast Decoder and again learn the object-centric slots keeping the hyper-parameters for the encoder and other modules same.

2.1 Diffusion Models

Diffusion models[2] are a class of generative models which learns the data-distribution through a fixed noising process and a learned reverse process to denoise the image. In this part we'll focus on a sub-category of diffusion models known as Latent space Diffusion model (LDM)[5] which trains the model in the latent space of a pretrained Variational AutoEncoder(VAE)[7] to lower the computational requirement by compressing the input space. The LDM is trained to generate the image latent (obtained by pretrained VAE) conditioned upon text but here you will condition it upon the slots. We provide the checkpoint of a pre-trained VAE and the inference script here.

2.2 Implementation

Training a slot conditioned diffusion model requires the following components, you're required to implement them from scratch in pytorch. You can read more about the implementation in [10]

2.2.1 Slot Encoder

This module is same as in 1.1. Given image x and trainable CNN based encoder $\mathcal{E}$, $\mathcal{E}(x)$ will give input feature vectors and slot attention module will give final `slots` given the input features of x.

2.2.2 Diffusion Decoder

Given an image $x \in \mathbf{R}^{H \times W \times 3}$ and it's corresponding $\texttt{slots} \in \mathbf{R}^{K\ D_{slots}}$, the goal of the diffusion model is to generate latent $z \in \mathbf{R}^{P \times P \times C}$, which is encoding of x through $z = \texttt{VAE}(x)$. Formally if $\mathcal{D}$ denotes diffusion model then $\mathcal{D}(x, \texttt{slots}) = \hat{z}$. Note that during training we train slot encoder and diffusion model **jointly** and `VAE` need to be **freezed**.

- **Forward-Process :** The training latents z (since we're working on LDM) are gradually corrupted with Gaussian noise with a variance schedule (β_1 ... β_T), implement a linear schedule which starts at $\beta_1 = 0.0015$ and ends at $\beta_T = 0.0195$ with 1000 steps, i.e $T = 1000$

- **Reverse-Process :** In the reverse process the diffusion model generates latent $\hat{z}$ given `slots`. It does so by denoising the $z_0 \sim \mathcal{N}(I, 0)$ iteratively for $T = 1000$ steps. Architecturally Diffusion models use a modified Unet architecture [6] to learn the denoising of latent z_{t-1}, to generate z_t. The Unet will comprise of a Residual Block (used for downsampling and upsampling) and a Transformer Block (used for slot conditioning) as shown in figure 3 which will be used in both downsampling and upsampling parts of UNet. A detailed architecture for the Unet Model can be found 5

  - *Residual Block*: Figure 4 shows detailed architecture of residual block. Note that residual block is used in both downsampling and upsampling part of UNet. To reduce the learnable parameters we can replace the first convolution layer of the residual blocks with different non parametric operations depending on whether the residual block is used for downsampling or upsampling.
    - *Downsample ResBlock:* In case of the downsampling replace the first convolution layer with average pooling with stride.
    - *Upsample ResBlock:* In case of upsampling replace first convolution layer with interpolation operation. [1]
    - In neither of case (that is output and input have same shape) keep the convolution layer with same padding.
  - *Transformer Block*: To condition the denoising on `slots`, cross attention with `slots` layers are added to UNet. These are called transformer block. With similar architecture to transformer decoder in [8]. This takes the slots from the slot-attention model and the intermediate latent of UNet and applies self-attention, followed by cross-attention as shown in figure 4. These attentions are multi-headed.

- **Decoding (Generation):** Implement Ancestral Sampling which is just iterative denoising of z_t to generate z_{t+1}. Start with a random noise z_0 of shape same as latent, iteratively run the noise through the unet from $T = 1000$ to $T = 0$. Refer to Algorithm 2 in [2]

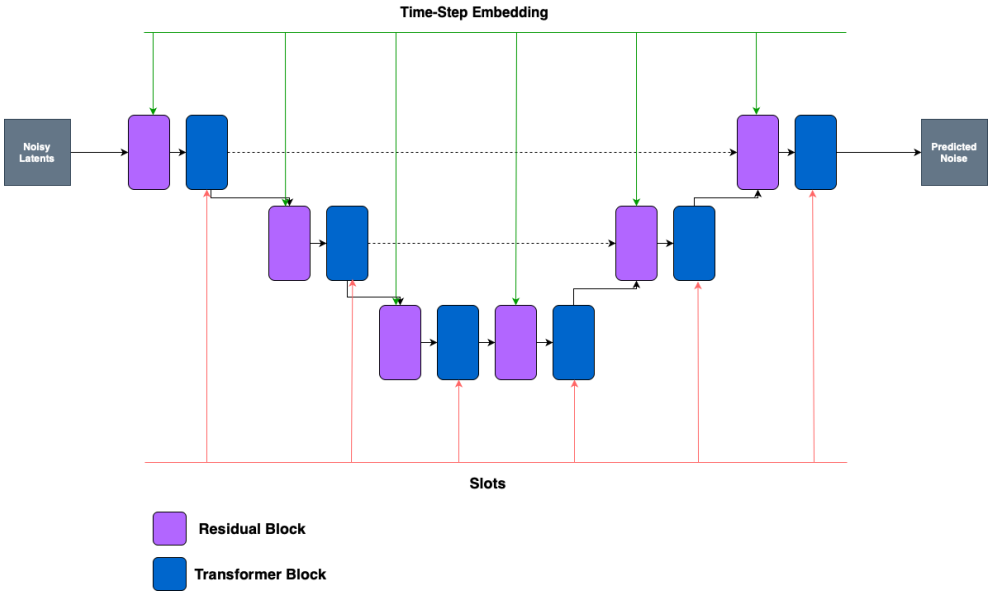

Figure 3: Overview of Unet Model

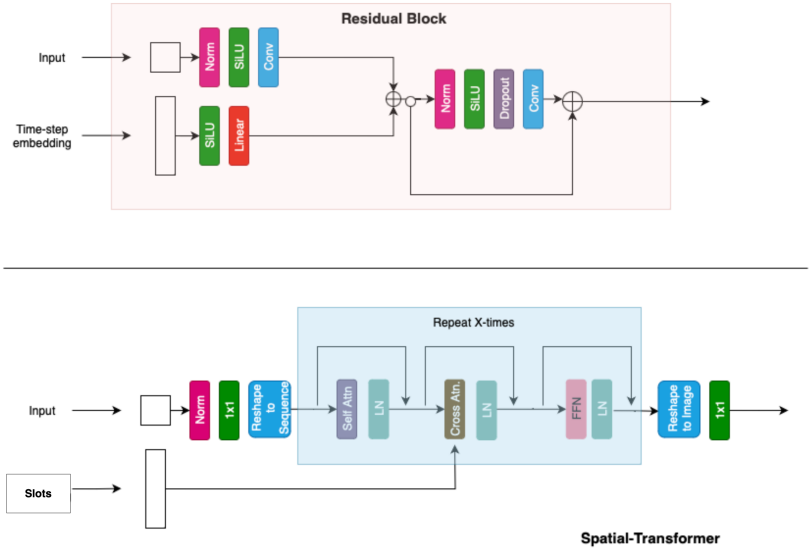

Figure 4: Detailed Block Architecture

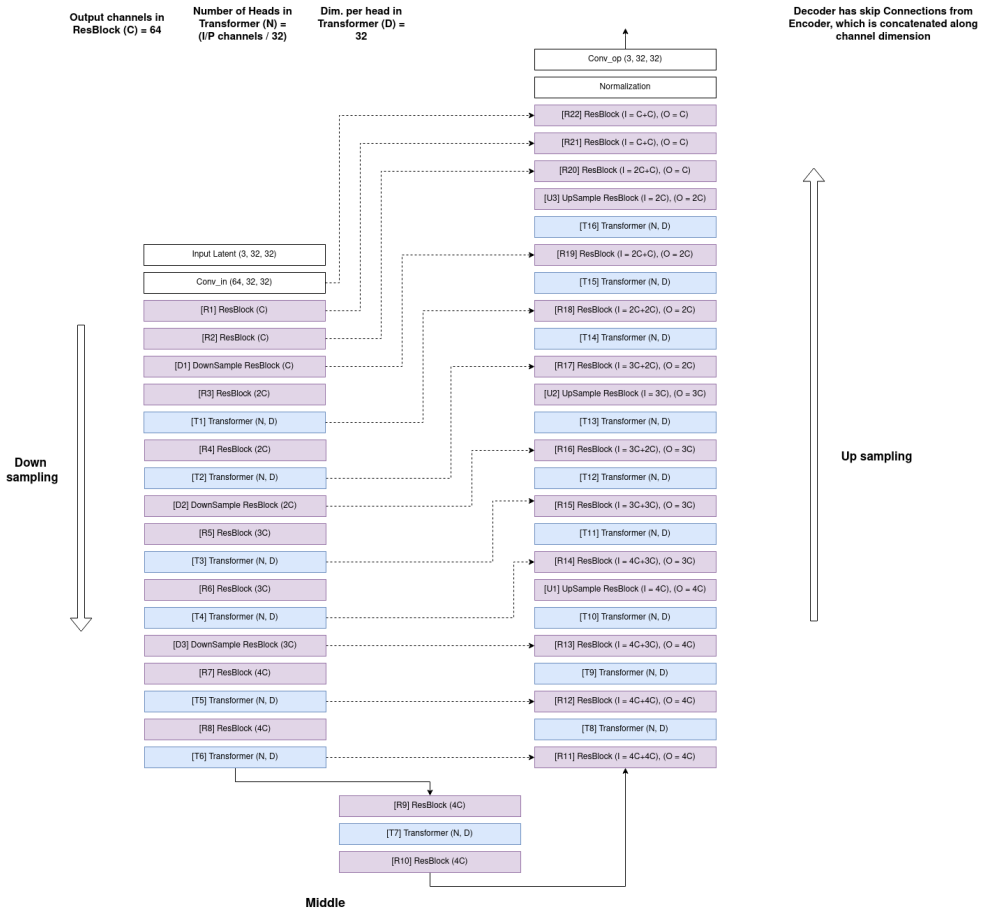

Figure 5: Detailed Block Architecture

2.3 Experiments

Repeat the experiments from part 1 on same ub-sampled CLEVRTex dataset. For hyperparameters we **strongly advice** you to start with this model mentioned above. Also refer to slot diffusion paper [10] for unmentioned hyperparameters. Note that there are no slot-masks for reporting ARI, instead we will use slot-input attention maps $\texttt{attn} \in \mathbf{R}^{\sqrt{N} \times \sqrt{N}}$ (line number 7 in 1) as proxy to slot mask. You need to resize this attention maps to $H \times W$.
To use provided checkpoint and code of `VQE` use the following code snippet: The image input to `VAE` need to be 128×128 dimension and normalized in $[-1, 1]$.

```
import VAE from vae_
import torch
vae = VAE()

#load the checkpoint
# (replace with your checkpoint path)
ckpt = torch.load('vae_checkpoint.pth')
vae.load_state_dict(ckpt)

#to encode an image
#images: [Batch size x 3 x H x W]
z = vae.encode(images)

#to deocde z
#z: [Batch size x 3 x 32 x 32]
rec_images = vae.decode(z)
```

References

[1] MS Windows NT kernel description. https://pytorch.org/docs/stable/generated/torch.nn.functional.interpolate.html. Accessed: 2010-09-30.

[2] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

[3] Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention, 2020.

[4] Gaurav Parmar, Richard Zhang, and Jun-Yan Zhu. On aliased resizing and surprising subtleties in gan evaluation. In *CVPR*, 2022.

[5] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[7] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning, 2018.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[9] Nicholas Watters, Loic Matthey, Christopher P. Burgess, and Alexander Lerchner. Spatial broadcast decoder: A simple architecture for learning disentangled representations in vaes, 2019.

[10] Ziyi Wu, Jingyu Hu, Wuyue Lu, Igor Gilitschenski, and Animesh Garg. Slotdiffusion: Object-centric generative modeling with diffusion models, 2023.