Research Project

Application of Diffusion Probabilistic Models on MRI Brain Scan Generation

Student: Patrick Siegfried Hiemsch
Supervisor: Anders Eklund

January 5, 2023

1 Introduction

This work is the summary of my research activities in the context of the Research Project course (732A76) at Linköping University as part of the masters program "Statistics and Machine Learning".

In the following, after a little overview over the work, the research objectives will be discussed.

1.1 Overview

Denoising diffusion probabilistic models (DDPMs) are a type of generative model that has been recently proposed as a new method for image generation tasks.

The latest released models like Dall-E 2 [8], Midjourney [9] or Stable Diffusion [7] which all leverage this new class of generative models show promising results in the synthesis of high-resolution images from text prompts.

The availability of sufficient amounts of training data, especially in the Computer Vision field, is crucial for the performance of machine learning models. To help alleviate this bottleneck when working with images e.g. in the context of classification, researchers use different data augmentation techniques like random rotations or mirroring as regularization. This often helps to reduce overfitting, allowing the model to generalize better on unseen data by using the scarce data more efficiently.

In this context researches recently also experimented with more sophisticated data augmentation methods. For example Sandfort et al. [2] show in their paper that using synthetically created pictures in the training process of a segmentation network for medical images can improve the performance of the model.

Since DDPMs show comparable results when applied to image generation tasks [5], in this research project we will investigate their suitability in the context of their application as data augmentation tools. Therefore we will train a noise-to-image DDPM on brain MRI scans as a possible data generation candidate for improving brain tumor segmentation models.

The data we will use are 2-D slices from patients' the 5-channel MRI brain volumes included in the BraTS (Brain Tumor Segmentation) 2020 dataset [10].

We will start by first training a 2-channel model, using the T1CE-channel and the according segmentation mask. We will then proceed with a multi-channel model that will be trained on all five available channels included in the BraTS data volumes.

1.2 Research objectives

As explained before, the aim of this research project is to train a noise-to-image DDPM that will be fitted to the distribution of the training data, the (2-channel/5-channel) MRI brain scans, with the goal of using this fitted model to sample synthetic images that originate from the same/a very similar distribution.

The following steps will be covered as part of the research project:

1. Find an appropriate implementation of a DDPM architecture, that is suitable for the task at hand
2. Adjust the implementation to the training of images with variable number of channels
3. Conduct multiple experiments with different hyperparameter settings for the training of a 2-channel model
4. Conduct multiple experiments with different hyperparameter settings for the training of a 5-channel model
5. Analysis of sampled images

2 Theoretical Background

As described by Ho et al. [3], diffusion models are a form of latent variable model. The data, in most applications a corpus of images, is assumed to represent a sample X_0 from an unknown, high-dimensional distribution q, where every single example x_0 is assumed to be drawn from $q(x_0)$, so $x_0 \sim q(x_0)$.

The basic idea of these models is to first gradually add Gaussian noise to the image in a *forward process* of T-steps until the image consists entirely of noise. The learning objective of the model is to then learn how to reverse this process, so that a tensor of Gaussian noise can be plugged into it and noise will gradually be removed in a *reverse process*.

Figure 1 shows an illustration of both processes based on the latent model formulation $p_\theta(x_0) = \int p_\theta(x_{0:T})\, dx_{1:T}$, where x_0 is the original data and $x_1, \ldots, x_T$ represent latent variables with the same dimensionality as x_0.

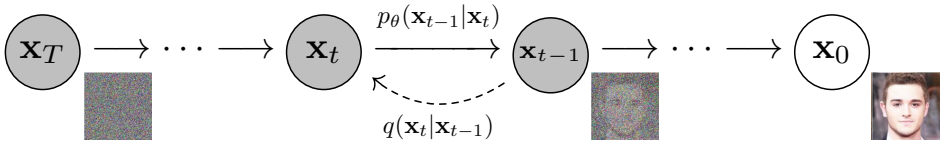

Figure 1: Illustration of diffusion process

The *forward process* can be described by a Markov chain that adds Gaussian noise to the tensor from the step before following a variance schedule $\beta_1, ..., \beta_T$. It can be formulated as the joint distribution of the latents $x_1, \ldots, x_T$ conditioned on the data x_0, so $q(x_{1:T}|x_0)$. This joint can be factorized as follows:

$$q(x_{1:T}|x_0) \coloneqq \prod_{t=1}^{T} q(x_t|x_{t-1}), \qquad q(x_t|x_{t-1}) \coloneqq \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

The *reverse process* can be formulated as a Markov chain as well, with learned Gaussian transitions that starts at the first tensor consisting of Gaussian noise $p(x_T) = \mathcal{N}(x_T; 0, I)$. The joint $p_\theta(x_{0:T})$ can also be factorized in the following way:

$$p_\theta(x_{0:T}) \coloneqq p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t), \qquad p_\theta(x_{t-1}|x_t) \coloneqq \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

Here $\mu_\theta(x_t, t)$ and $\Sigma_\theta(x_t, t)$ represent the learned mean and variance of the reverse process. In the original paper by Ho et al. [3] the variance Σ_θ is kept constant. But as Nichol et al. [6] show in their work, learning the variance as well together with the mean can be beneficial for the quality of samples.

To model the reverse process, a U-Net backbone architecture is used with the integration of self-attention in both underlying papers [3, 6].

Presenting the exact equations and procedures underlying the training process of the DDPM implementations used in this project are beyond the scope of this work. A good starting for a deeper understanding of the models is the work by Ho et al. [3] and the improvements made by OpenAi summarized in their paper by Nichol et al. [6] followed by Dharwani et al. [5]. But for the sake of completeness, a short introduction to the loss function will be provided in the following.

In the original paper, the variational lower-bound (VLB) on negative log-likelihood is optimized [3]:

$$\mathbb{E}[-\log p_\theta(x_0)] \leq \mathbb{E}_q\left[-\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}\right] = \mathbb{E}_q\left[-\log p(x_T) - \sum_{t\geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}\right] =: L$$

With a few simplifications, Ho et al. [3] propose to use a simplified variant of the VLB as the appropriate loss term:

$$L_{\text{simple}}(\theta) := \mathbb{E}_{t,x_0,\epsilon}\left[\left\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\right\|^2\right]$$

Here ϵ_θ refers to the noise predicted by the trained network. Ho et al. [3] found, that the training works best when predicting the noise $\epsilon_\theta(x_t, t)$ instead of modeling $\mu_\theta(x_t, t)$ directly.

Nichol et al. [6] argue that for a further improvement of the DDPMs' likelihood in [3], which is used as one of the main metrics to compare different likelihood based models in the context of image generation tasks, the variance Σ_θ needs to be learned as well. Since the loss represented by L_{simple} is independent of Σ_θ and the gradient thus does not contain any loss signal for the variance, they propose a new loss function:

$$L_{\text{hybrid}} = L_{\text{simple}} + \lambda L_{\text{vlb}}$$

The exact calculations of these losses can be found in [6].

The authors also show in their work that the linear noise schedule proposed in [3] works well for high-resolution images (e.g. 128x128 or 256x256) but that a cosine-schedule can be beneficial for lower resolution images [6].

Another important finding especially in the context of practicability that Nichol et al. [6] present is the number of sampling steps needed to create high-quality results. Sampling one example from a model trained using 4000 diffusion steps can take up to several minutes even while running computations on GPU. But as figure 2 suggests, fewer steps for the sampling phase could be sufficient as well. The plot shows the Fréchet Inception Distance (FID) measuring the similarity of generated and real images [1] for different models and sampling steps (25, 50, 100, 200, 400, 1000, 4000). We can observe that after 200 steps the FID does only improve minimally for models trained with the L_{hybrid} which is why the authors suggest using 250 steps for sampling. In general this can be achieved by adjusting the sampling variances to this timestep respacing. The corresponding equations can be found in [6].

This process improves sampling efficiency and time immensely, which is why we will also use it in this work.

There is also another technique for increasing sampling speed for DDPMs, the DDIM algorithm [4]. But as Nichol et al. [6] showed in their worked, this algorithm is only beneficial when using less than 50 sampling steps.

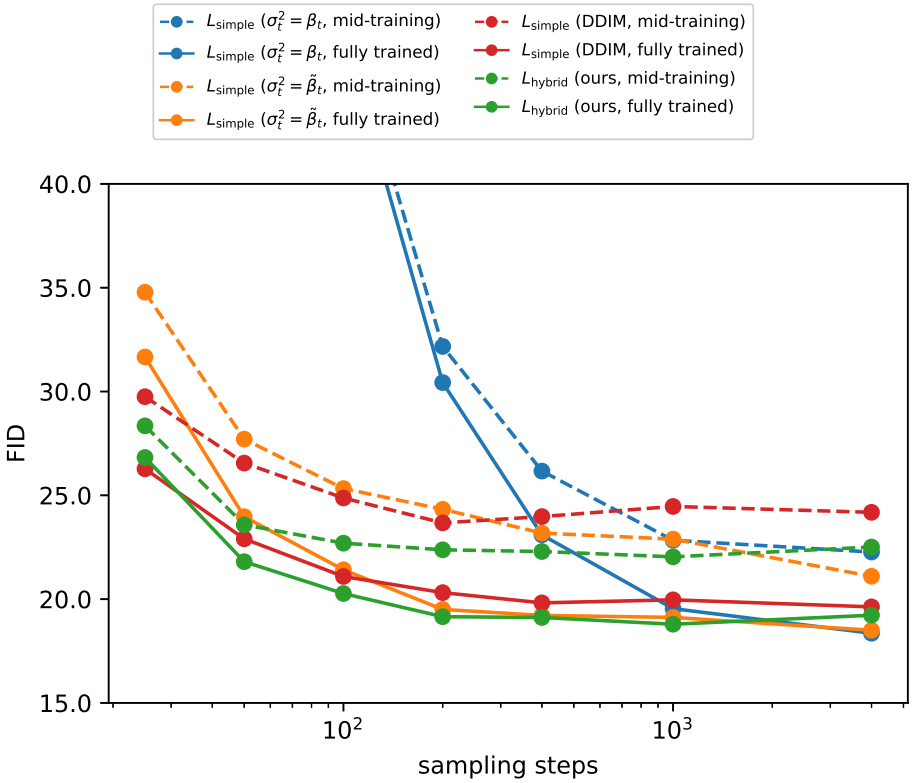

Figure 2: FID versus number of sampling steps, for models trained on ImageNet 64×64. All models were trained with 4000 diffusion steps. [6]

3 Methods

3.1 Data

As described in the introduction, the data used for training of the different DDPMs in this work is based on the BraTS 2020 dataset [10] that can be found here.

The BraTS dataset is a collection of MRI scans and related data for brain tumor segmentation. It was used in the Multimodal Brain Tumor Segmentation Challenge in 2020. The dataset consists of MRI scans from patients with gliomas, which are a type of brain tumor. Each MRI scan is accompanied by a set of manual segmentation labels that indicate the location of the tumor in the scan. Besides the Segmentation labels, each patients data consists of four MRI volumes that were acquired using four different modalities:

1. **T1-weighted (T1)**: This modality is sensitive to the density of different tissues in the body and is often used to visualize the anatomy of the brain.
2. **T1-weighted with contrast enhancement (T1Gd)**: This modality is similar to T1, but a contrast agent is injected into the patient's bloodstream to highlight certain tissues. This can help to more clearly visualize abnormalities such as tumors.
3. **T2-weighted (T2)**: This modality is sensitive to the water content of tissues and is often

used to visualize abnormalities in the brain such as tumors, inflammation, and edema.

4. **Fluid-attenuated inversion recovery (FLAIR)**: This modality is similar to T2, but it is specifically designed to suppress the signal from normal brain tissue, making it easier to visualize abnormalities.

Since the goal of this work is to train a 2D noise-to-image DDPM, the volume data of 369 patients was taken and split into 2D slices to be able to use them in the described context. On average 64 slices were taken per patient, leading to a corpus of 23,478 grayscale images for each of the 5 channels.

As a first approach we will just use the T1Gd channel to create a first one-channel greyscale DDPM, then add the segmentation mask as a second channel to train a two-channel model, followed by experiments to create a DDPM on the full five channels included in the dataset.

Note that following the labels in the dataset, the T1Gd channel will be called T1CE from now on.

3.2 Model Implementation

The model implementation used throughout this entire work is based on the code published by OpenAI in their github repository, which is the code used to produce the results mentioned in their 2020 paper [6].

An even more recent implementation also by OpenAI, which is based on the code mentioned before can be found here. This repository is the basis for the models described in [5]. The code for unconditional training, so training of DDPMs without any class labels, is very similar to the implementation used for this work. Only when training class-conditional models, so DDPMs that differentiate between classes when learning the sampling distributions, the newer repository seems to be more relevant.

3.2.1 Structure overview

The "improved diffusion" package provided by OpenAi contains the PyTorch implementation of the DDPM definition in their connected paper [6].

It allows the training and sampling of noise-to-image DDPMs and also includes scripts for applying super-resolution which are out of scope for this work.

The implementation is tailored towards the training of RGB images, which will need to be adjusted for the application horizon in this work. Training is started by providing the **image_train.py** script with the respective directory where images are stored and different other user specific parameters that are explained in the next subsection.

The training process on a high level can be summarized as follows: Inside the **image_train.py** script, first the data loader is created which yields a new batch of images from the specified directory together with their label (for class-conditional training). Since the images are yielded and not

returned, there will be no stop signal "StopIteration", so the loader yields new batches infinitely. The preprocessing of the data inside the custom Dataset class is described in Ho et al. [3] as follows:

> "We assume that image data consists of integers in $\{0, 1, ..., 255\}$ scaled linearly to $[-1, 1]$. This ensures that the neural network reverse process operates on consistently scaled inputs starting from the standard normal prior $p(x_T)$." [3, p. 4]

Inside the **TrainLoop** class, a new batch is fetched and one forward and backward pass is performed with the `run\_step()` function. Important to note is that the underlying implementation enables microbatching, meaning that if a batch does not fit in the GPU all at once, so computing the gradients in parallel for each of the images in the batch and taking one gradient step exceeds the internal memory, microbatching can be used to still take advantage of the stability of higher batch sizes. This is done by splitting the full batch size specified into smaller microbatches of user provided size, which will fit into the GPU all at once. This way the gradients of the full batch of images are used for one gradient update.

There are four implemented loss types, that can be used:

1. MSE
2. Rescaled MSE
3. KL-loss (VLB loss)
4. Rescaled KL-loss

The one that will be used by the models we train is the MSE-loss since the use_kl parameter is False by default and we set rescale_learned_sigmas to False (in the later experiments). This represents the L_{hybrid} loss mentioned above, so when learn_sigma is set to True, the L_{vlb} depending on the learned variance is included too.

The other three losses are a rescaled version of the MSE loss and a normal and rescaled version of the KL-loss, which will only use the term referring to the VLB, so L_{vlb}, for the loss computations. The exact computations of the target and denoising model output to compute the MSE-loss can be found in [3].

For each of the N inputs in the (micro)batch the loss for a certain diffusion step t is computed, which is drawn either uniformly over the diffusion interval $t \in \{1, ..., T\}$ or sampled using importance sampling based on previous losses. This makes timesteps with high loss values more likely to being sampled and updated accordingly. More information regarding this process can be found in the paper [6]. Using the computed list of losses for each image, the weighted average loss is computed. If not specified differently, each loss is weighted similarly as unbiased importance sampling. Only if importance sampling is used, this mean differs from the standard unweighted mean since the according importance weights are used.

An overview over the general training and sampling algorithm can be found in figure 3.

The computed loss is then propagated backwards through the network. When using microbatching, a gradient update is not performed immediately but the propagated loss for each of the weights is

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Figure 3: Training and Sampling algorithm overview

rather accumulated for the whole batch over all microbatches. This procedure makes sense since the same microbatchsize is used in each iteration until the full batch is processed and the mean of individual sample means of the same sample size is the same as the mean for the whole population. So computing the mean of the microbatch loss means will yield the mean loss for each batch. But when inspecting the implementation for the package it looks like the mean of the loss computed for each microbatch is propagated backwards without weighing it by the number of microbatches used. The weight used in the snippet in figure 4 seems to only include the importance sampling weights. If this is actually true, then this procedure would lead to an accumulation of losses rather than taking the mean over all the microbatch means. This should be investigated further in order to make sure that the right loss signals are used.

```
        loss = (losses["loss"] * weights).mean()
        log_loss_dict(
            self.diffusion, t, {k: v * weights for k, v in losses.items()}
        )
        if self.use_fp16:
            loss_scale = 2 ** self.lg_loss_scale
            (loss * loss_scale).backward()
        else:
            loss.backward()
```

Figure 4: Snippet of `forward_backward()` function of TrainLoop class

After the loss computation for the whole batch, a gradient step is taken using the AdamW algorithm. This change is then also propagated to the EMA model, which is a second model, whose parameters are updated according to an exponential moving average with default rate of 0.9999. This model is characterized by the authors to produce better results than the normal model, when of course trained for a sufficient amount of time. The parameters of the model will put a high weight on the historic EMA-weights (which start off as random) but will also integrate a small part of the most recent model weights (in the default case 0.01%). This seems counterintuitive at first when thinking of common EMA formulations for time series in finance for example found here, where the smoothing parameter represents the ratio of the new parameters that is used for the update of the old EMA values (so $(1-\alpha) \cdot \text{EMA}_{t-1}$ remain). But in the implementation found in the repository, the EMA update is calculated according to the following formula:

$$\text{EMA}_t = (1-\alpha) \cdot \text{Par}_t + \alpha \cdot \text{EMA}_{t-1}$$

Here EMA_t stands for the EMA model parameters at time step t and Par_t represents the current model parameters. The default value of 0.9999 thus implements a very slow update of model

weights, which is why the first EMA models will always be just noise as the experiments will show.

The implementation can be found in the two code snippets in figure 5 and 6.

```python
def update_ema(target_params, source_params, rate=0.99):
    """
    Update target parameters to be closer to those of source parameters using
    an exponential moving average.

    :param target_params: the target parameter sequence.
    :param source_params: the source parameter sequence.
    :param rate: the EMA rate (closer to 1 means slower).
    """
    for targ, src in zip(target_params, source_params):
        targ.detach().mul_(rate).add_(src, alpha=1 - rate)
```

Figure 5: `Update_ema()` function

```python
def optimize_normal(self):
    self._log_grad_norm()
    self._anneal_lr()
    self.opt.step()
    for rate, params in zip(self.ema_rate, self.ema_params):
        update_ema(params, self.master_params, rate=rate)
```

Figure 6: Invocation of EMA-update

The script then logs the progress in the specified log interval and saves models according to the save interval. In the log file, the user gets access to the L_2-norm of all model gradients, the batch-loss, 4 loss time quartiles, the MSE (which is redundant to the loss when using MSE as loss function), the 4 time step quartiles for the MSE and the VLB including its quartiles. These quartiles in general are referring to loss/MSE/VLB values connected to one of the four time step quartiles in the interval $t \in \{1, ..., T\}$ (e.g. q0 stands for loss/MSE values in the first 25% of timesteps).

Sampling can be done in the same way as training by using the sampling script and providing it with a valid model checkpoint and model and diffusion flags.

3.2.2 Model parameters

There are a lot of different parameters that can be given to the model for training and/or sampling. Some of the essential (hyper)parameters will be briefly explained in the following. Others are optional and already have some good defaults that were used for the experiments in this work.

Some of the parameters mentioned here are not directly described in the repository by OpenAi but can be found by closer code inspection and are used for the experiments as they are practical for training and sampling.

The following tables show the most important model, diffusion, training and sampling parameters that should be considered (without particular order). In general, the same model and diffusion parameters should be specified during training and sampling.

Parameter	Default	Explanation
`log_interval`	10	After how many steps log file should be updated
`save_interval`	10,000	After how many steps model (incl. EMA + optimizer state) should be saved
`out_dir`	3	Output directory for model checkpoints & log-files
`resume_checkpoint`	""	Path to model (not EMA) whose training is to be continued
`lr`	1e-4	Learning rate
`batch_size`	1	Batch size to use during training
`microbatch`	-1	Use microbatches, that fit into GPU at once, if only limited amount of compute available (if -1 disabled)
`ema_rate`	"0.9999"	comma-separated list of EMA values, for which EMA models should be saved
`schedule_sampler`	"uniform"	If "uniform", no importance sampling, if "loss-second-moment" importance sampling enabled
`use_kl`	False	If True, L_{vlb} will be used instead of L_{hybrid}

Table 1: Training parameters

Parameter	Default	Explanation
`in_channels`	3	Number of input channels of the images
`image_size`	64	Size of the images
`num_channels`	128	Base channel count for U-Net model
`num_res_blocks`	2	Number of residual blocks per downsample
`num_heads`	4	Number of attention heads in each attention layer
`learn_sigma`	True	Whether to learn the variance Σ_θ
`use_scale_shift_norm`	True	Whether to use a FiLM-like conditioning mechanism
`attention_resolutions`	"16,8"	Collection of downsample rates at which attention will take place. May be a set, list, or tuple. For example, if this contains 4, then at 4x downsampling, attention will be used.
`class_cond`	False	Whether model is class conditional

Table 2: Model parameters for reverse process U-Net model

Parameter	Default	Explanation
`diffusion_steps`	1000	Number of time steps used in diffusion process
`noise_schedule`	"linear"	Type of noise schedule, "cosine" is alternative (adds noise slower)
`rescale_learned_sigmas`	True	Whether to use rescaled MSE loss (if not VLB term can possibly hurt MSE term)
`rescale_timesteps`	True	If True, pass floating point timesteps into the model so that they are always scaled like in the original paper (0 to 1000).

Table 3: Diffusion parameters

Parameter	Default	Explanation
`batch_size`	16	Batch size to use during sampling
`num_samples`	10000	Number of samples to generate
`timestep_respacing`	""	Whether to use rescaled timesteps (i.e. less diffusion steps) for sampling
`model_path`	""	Path to the model to use for sampling
`sample_dir`	"Samples"	Directory to store samples
`use_ddim`	False	Whether to use DDIM algorithm for sampling [4]

Table 4: Sampling parameters

The parameter `in_channels` shall be highlighted again since it is important for the training of models for images with arbitrary number of channels and not only RGB images. More detailed explanations can be found in the subsection about this topic.

3.2.3 Deployment

For the deployment of the training and sampling scripts, the provided scripts in the original repository are used as a basis. They are embedded into a `.py` file, where the different parameters are specified in flags and the `image_train.py` or `image_sample.py` files are invoked. These experiment specific scripts are then run using a defined shell-script `run_script.sh`, which uses nohup to train and sample the models in the background. The outputs of the processes are then temporarily stored in the `output.txt` file. Also, a file `PID_current.txt` will be created that contains the current process ID, so that training can be stopped easily with the `kill_pid.sh` script (but unfortunately sometimes the corresponding GPU process needs to be killed manually).

3.2.4 Custom BraTS Dataset/-loader

To be able to use the BraTS images for training, each channel that should be included must be loaded separately from the according directory. To enable this in a flexible way and to make it easily integratable with the infrastructure of the model at hand, a custom `Dataset` and `DataLoader` were developed. They can be found inside the file `BRATS_dataset.py`.

The original infrastructure uses the `ImageDataset` class, that loads images (incl. labels for classes), converts them to RGB, resizes them if necessary and normalizes them to $[-1, 1]$. It returns the transposed image as [Channels x Size x Size] together with a dictionary for the class label. This function is then invoked inside the `load_data()` function, which wraps the created Dataset in the PyTorch `DataLoader` with the given batch size, `shuffle=True` and `drop_last=True`. Once called, the `load_data()` function then acts as an infinite generator that yields a new batch of images (and labels) each time the `next()` operator is applied on it inside the training loop. Interesting in this

context is, that this generator returns images randomly (when the `DataLoader` is set with enabled shuffling) but epoch-wise. So if an image is included in a batch it will not be sampled again until all other images in this epoch were chosen once. For the next epoch, the images get then shuffled again so the order of the batch indices is different for each epoch.

To follow this structure for the BraTS data, first a custom dataset `load_channels_from_dir()` was developed. This dataset-subclass takes a list of image directories, where each image directory itself is a list with the paths to the different channels to be included in the training. When retrieving a new item, the according image channels are loaded separately, normalized similar to the procedure in the original implementation, converted to a tensor and then stacked together in one tensor of size [Channels x Size x Size]. Returned is then this stacked tensor and an empty dictionary for the labels (since in the context of this work, class unconditional models are trained), to follow the same structure as the original dataset which also returns these two objects. Then this dataset is invoked in the `load_data()`, similar to how it was described before.

Inside the `BRATS_dataset.py` the dataset directory needs to be specified and the list of channels to be included during training needs to be saved in the `CHANNELS`. The possible channels are `['FLAIR', 'Seg', 'T1', 'T1CE', 'T2']`. This is also the order in which channels will be returned by the loader inside the tensor.

As a final step, the original `load_data()` function is removed as an import and replaced by the BraTS pendant inside the `image_train.py` script.

3.2.5 Variable channel training

To actually be able to use the batches returned by the custom BraTS dataloader in the model, a few adjustments need to be made to the code.

By more detailed inspection of the code it becomes obvious that the training process is hard coded to be tailored to three channel RGB images. Even when using just grayscale images, the images will be transformed into RGB style with three redundant channels.

To adjust the code to variable number of channels, first the input of the U-Net model needs to be changed. To achieve this the parameter `in_channels`, that was previously hard coded to 3 in U-Net, was added as an adjustable parameter in the `create_model_and_diffusion()` in the `script_util.py` file (with default 3 in `create_model_and_diffusion_defaults()`). This parameter is then propagated all the way down to the `UNetModel` constructor method.

This fixes the input issue. Now only the output channels also need to be adjusted. They were set to 3 or 6 for the case when $\Sigma_\theta(x_t, t)$ is trained as well. To make them follow the number of input channels, they were simply set to `in_channels` or 2*`in_channels` respectively.

Why twice the amount of input channels are needed when $\Sigma_\theta(x_t, t)$ is trained is still unclear but as the experiments will show, setting these parameters for the output channels result in exactly in the intended behaviour.

Now with these adjustments, the `in_channels` parameter just needs to be provided as one of the model flags inside the training call to enable training of arbitrary number of channels.

3.3 Hardware

All experiments were conducted on computers provided by LiU. For the first experiments, models were trained on NVIDIA GeForce RTX 3060 Ti cards with 8 GB VRAM.

To increase stability of the training process by using higher batch sizes, the later experiments for the two and five channel models were performed on NVIDIA GeForce RTX 3090 cards with 24 GB VRAM.

4 Results

During the time of the research project different experiments on different GPU architectures and with various parameter adjustments were conducted in relation to the described (altered) code base. The following shows a brief overview over the included experiments:

All training and sampling scripts referenced in the following that were used for the experiments can be found in the `ExperimentScripts` folder in the repository accompanying this work.

1. **Introduction Experiments**: Training of first model based on the CIFAR-10 dataset, sampling of LSUN-bedroom checkpoint
2. **1-Channel-Model**: Model trained on T1CE greyscale images
3. **2-Channel-Model**: Model trained on T1CE images including Segmentation mask
4. **5-Channel-Model**: Model trained on all five channels

4.1 Introduction Experiments

As a starting point an unconditional model was trained on the CIFAR-10 dataset based on the example included in the original repository. Since the model was trained on a lot less training steps, the results are far from optimal compared to the ones in [6] but work as a proof of concept. Some samples of this run can be found in figure 12.

Also, the checkpoint for the LSUN-bedroom dataset model found in the repository under the following link was used to create some diffused bedroom examples. Some examples are shown in figure 13.

4.2 1-Channel-Model

The first step to approach the BraTS model was to try to implement a noise-to-image DDPM that could sample a single MRI channel, the T1CE channel.

Before the custom BraTS dataset was implemented, a first experiment was run based on the original architecture which still transformed greyscale images into RGB images with three redundant channels.

For this experiment and in general for all following experiments the parameter setup of the unconditional LSUN-bedroom model that the authors trained in [6] is used. It seems particularly fitting since the image size is exactly the same (256x256) and it is also an unconditional setup. Only slight adjustments were made.

Experiment 1

The first described model in Experiment 1 was trained for 100K steps (which equals around 5 full epochs considering the 23,478 images) on the small GPU, which only worked with a batch size of 1.

Experiment 2

The second one channel model in Experiment 2 used the custom BraTS `DataLoader` and was trained for 230K steps, so around 10 epochs, also on batch size one.

The results of these runs can be found in the repository files (locally on lnx00276) but are not included in this report since they just represent a first experiment whose results were not that impressive.

4.3 2-Channel-Model

In the context of 2-Channel-Models, the T1CE channel together with the segmentation mask was chosen as inputs to the model.

Experiment 1

In the first experiment, the model was trained for 250K steps (around 10 epochs) using the stated hyperparameters connected to the unconditional LSUN-bedroom training with batch size 1 since it was still conducted on the small GPU.

Experiment 2

For Experiment 2, the bigger GPU was used, which allowed an increase of the batch size to 6. The training was run for 40 epochs (160K steps). This increase in batch size led to a significant stabilization of the loss since 6 images were used for the gradient updates instead of 1 which led to very noisy loss calculations.

Experiment 3

In Experiment 3 the batch size was kept high at 7 but the learning rate was decreased from $1e-4$ to $1e-6$. The higher learning rate was used during the LSUN-bedroom training which led to high quality image samples, which is why for this work we tried to keep as close to this proven stable setup as possible.

But since only that processing of 7 images fitted in the GPU in parallel and the authors of [6] used a batch size of 128, it seemed to be a valid approach to lower the learning rate. A casual discussion about this can be found under this link.

In general, the training of the LSUN-bedroom model whose checkpoint was used to create the images in figure 13 was run for 1.2 million steps on a batch size of 128, so the model saw around 153.5 million images during training (in other runs they used 64 or 256 as batch size but always processed between 120-155 million images). Their dataset consists of around 50K pictures which indicates 3072 full epochs for the training.

This means that to replicate this training behavior, 3072 full epochs would need to be taken (even though the distribution of the brain scans seems to be less complex and diverse which could allow the model to pick it up more quickly). Applied to the BraTS dataset and this learning rate, this corresponds to $(23,478x3072)/7 = 10.3M$ steps. Even though the training was run for several days, only 220K steps were reached. But the results, which can be found in the repository, still seemed of reasonable quality.

The training took around 9.68 s to process 100 images (2hrs (113 min) for 10K steps so 7x10,000=70,000 pics, so (113x60)x100/70000=9.68s/100pics)

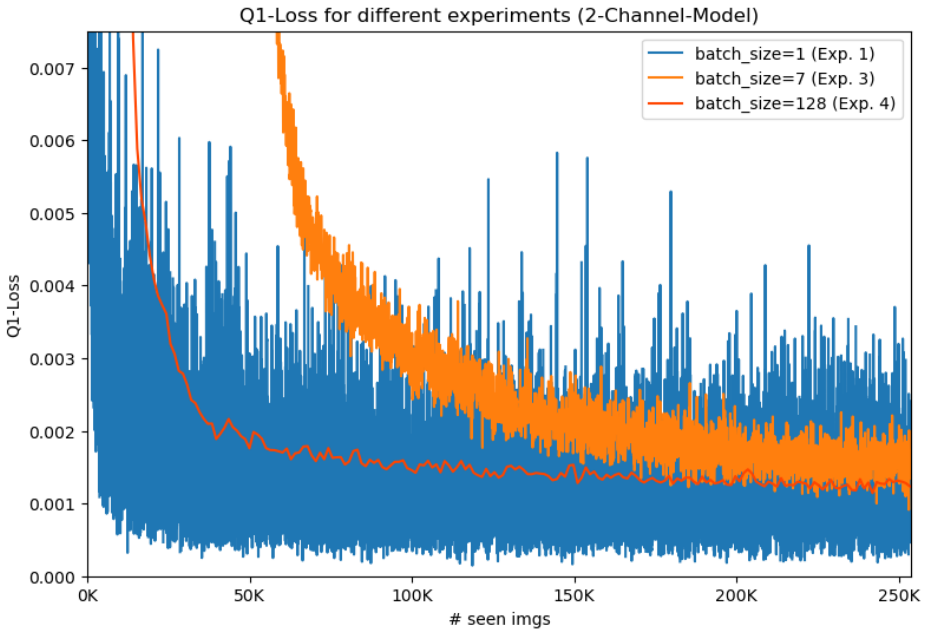

Figure 7: Q1-Loss comparison for different batch sizes

Experiment 4

The last experiment was conducted by using the microbatching technique explained above which allowed the training on a higher batch size even though it does not fit in the GPU all at once. By using this, we were able to replicate the LSUN-bedroom training setting entirely by increasing the batch size to 128 with a microbatch of 7 (learning rate was set back to $1e-4$).

This lead to an even more stable training which can be seen in the comparison of the loss in the seocond quartile of the diffusion process (so the loss of time steps 250 to 500) Experiments 1, 3 and 4 in figure 7.

In general this model showed the best results and was run for the longest time (60K steps on batch size 128 equalling around 7.7 million images seen so 328 epochs). Even though microbatching was used, the model did not suffer any increased processing time per image (Detailed: Run for first 10,000 steps from 25.12. 20:11 to 27.12. 03:51 24,7,40min=31h40min = 1900 minutes for 10,000*128 pics, so around 9 secs per 100 pics).

All of the figures related to this model will be based on the 60K steps model checkpoint(s) (besides the development of results over steps of course).

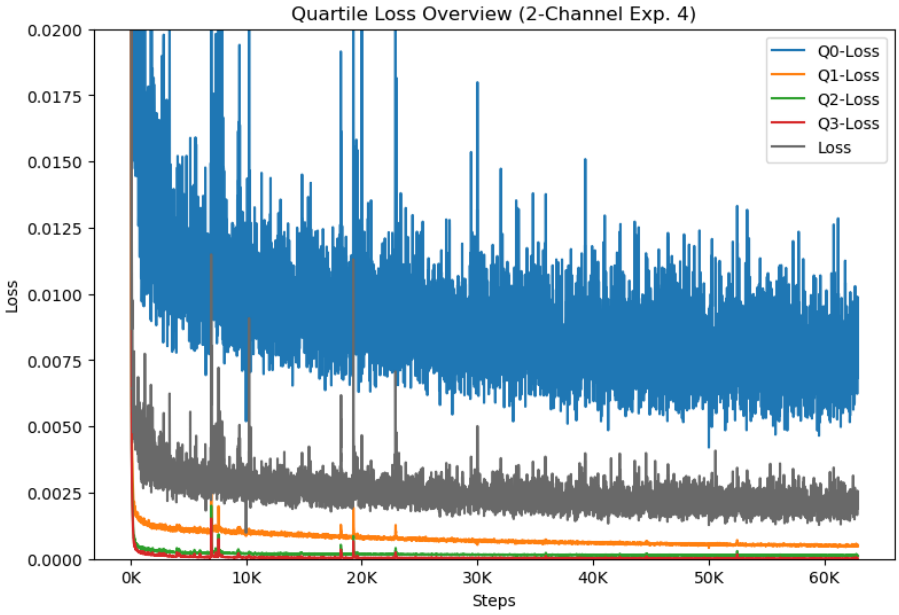

Figure 8: Loss curve for all four timestep quartiles

For the sampling procedure timestep respacing from 1000 to 250 steps was used, which results in a massive improvement of sampling time per image. Figure 9 shows two results from a batch of images sampled with respacing. Figure 10 shows two results from a batch sampled with the full 1000 diffusion steps. The batches contain 42 images each which were generated in parallel, filling up almost all GPU memory. The respaced batch took 285 seconds, so around 6,8 seconds per image. The full diffusion images on the other hand took 1131 seconds (=26,9 seconds per image). So the respacing resulted in a x4 speedup.

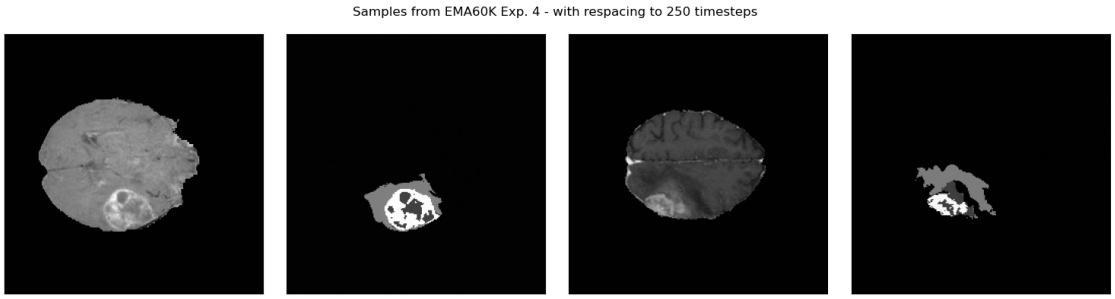

Figure 9: Respaced 60K-EMA samples Experiment 4 (250 steps)

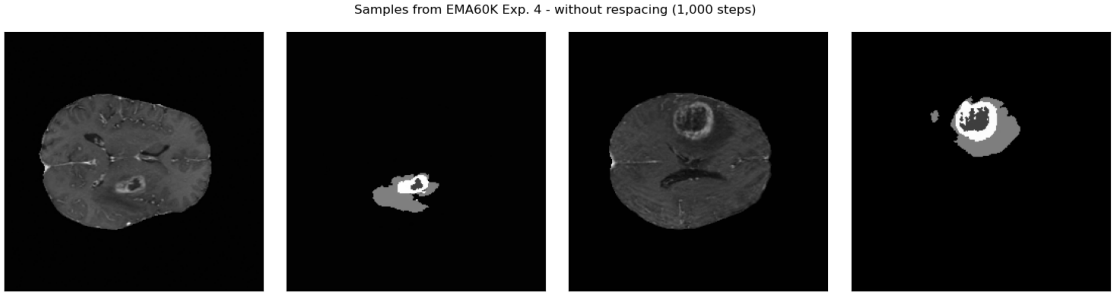

Figure 10: Non-Respaced 60K-EMA samples Experiment 4 (1000 steps)

Figure 8 shows an overview of the different timestep quartiles. We can see that steps in the first part of the diffusion process have much higher loss signals, which also makes sense since they correspond to the first steps where most of the real image is still present in comparison to later time steps which almost entirely consist of noise [6] (where there is nothing to learn).

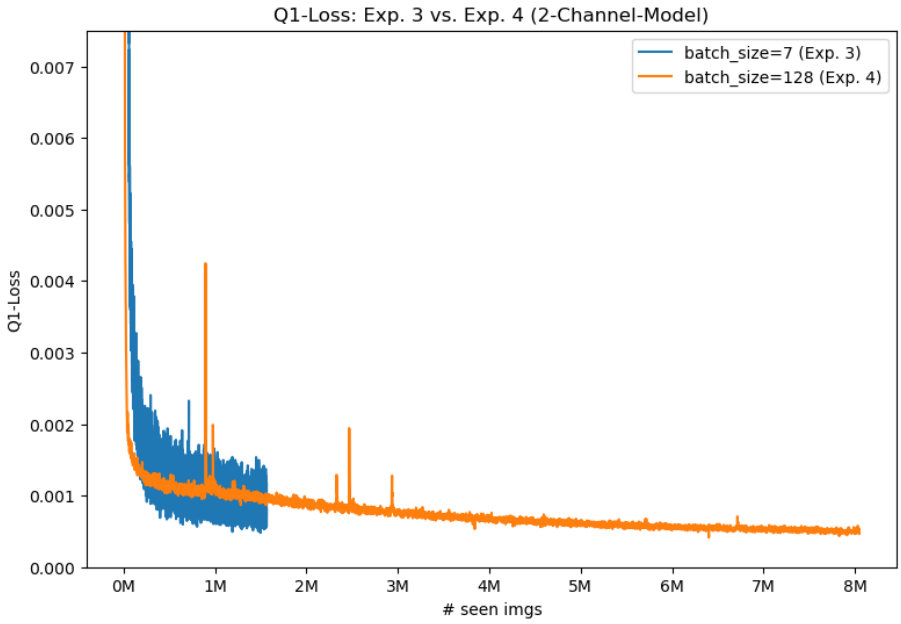

Figure 11: Q1-Loss curve comparison of Experiment 3 and 4

Figure 11 shows a comparison of the shorter training of experiment 3 vs. experiment 4 based on

the seen images during training.

Example of the samples of the EMA-model and normal model of this experiment can be found in figure 14 and figure 15.

In general, in all of the models trained during the research work, a big part of the generated samples (around 10-30%) include just noise channels, especially for the Segmentation channel. This issue should also be addressed and investigated more deeply in further research.

The evolution of the (EMA-)model over the training steps can be seen in figure16 and figure 17, where a few random samples form the batches per step-checkpoint are visualized.

4.4 5-Channel-Model

The 5-Channel-Model experiment was conducted on the same bigger GPU with the same parameter settings as Experiment 4. It was run for 45K steps with batch size 128. Interestingly, even though the learning curve suggests that the model still learns, performance of samples decreased after 30K steps. Maybe the model needs to be run for a far longer time span or adjustments to hyperparameters or architecture (e.g. importance sampling, attention resolution, ...) should be made.

Some of the results of sampling from the 30K model can be found in figure 18.

5 Conclusion

In general the model class and the specific implementation of the DDPMs used in this work proved themselves as powerful tools to pick up and sample from the distribution of input image data with various number of channels.

The results, especially the ones of the long multi-day run of the fourth experiment for the 2-Channel-Model were of high quality. But one of the problems is, that the resulting images were only judged based on their perceived quality. To really get a good estimate of how well the models were able to pick up the distribution of the data, e.g. a measure like the FID score should be used to test this.

Also, in further research it should be investigated and tested whether the models are really generalizing (so generating sample MRI scans of new brains) or whether they are overfitting the data and just replicating images.

In general, a more in depth tuning of the vast amount of hyperparameters could be beneficial to improve results, especially with regard to the 5-Channel-Models. By increasing computational resources and distributing training on multiple GPUs, more steps could be taken in the same time and the amount of epochs that the model in the paper [6] was run could be replicated (328 epochs in experiment 4 vs. 3072 for LSUN-bedroom).

If the created images actually help a cancer segmentation model as a form of sophisticated data

augmentation to improve segmentation performance needs to be tested in further research.

References

[1] Martin Heusel et al. "Gans trained by a two time-scale update rule converge to a local nash equilibrium". In: *Advances in neural information processing systems* 30 (2017).

[2] Veit Sandfort et al. "Data augmentation using generative adversarial networks (CycleGAN) to improve generalizability in CT segmentation tasks". In: *Scientific reports* 9.1 (2019), pp. 1–9.

[3] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6840–6851.

[4] Jiaming Song, Chenlin Meng, and Stefano Ermon. "Denoising diffusion implicit models". In: *arXiv preprint arXiv:2010.02502* (2020).

[5] Prafulla Dhariwal and Alexander Nichol. "Diffusion models beat gans on image synthesis". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 8780–8794.

[6] Alexander Quinn Nichol and Prafulla Dhariwal. "Improved denoising diffusion probabilistic models". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 8162–8171.

[7] *Stable Diffusion Version 2*. original-date: 2022-11-23T23:59:50Z. Jan. 2023. URL: `https://github.com/Stability-AI/stablediffusion` (visited on 01/05/2023).

[8] *DALLE 2*. en. URL: `https://openai.com/dall-e-2/` (visited on 01/05/2023).

[9] *Midjourney*. en. URL: `https://midjourney.com/home/?callbackUrl=%5C%2Fapp%5C%2F` (visited on 01/05/2023).

[10] *Multimodal Brain Tumor Segmentation Challenge 2020: Data*. URL: `https://www.med.upenn.edu/cbica/brats2020/data.html` (visited on 01/05/2023).

A LSUN & CIFAR samples

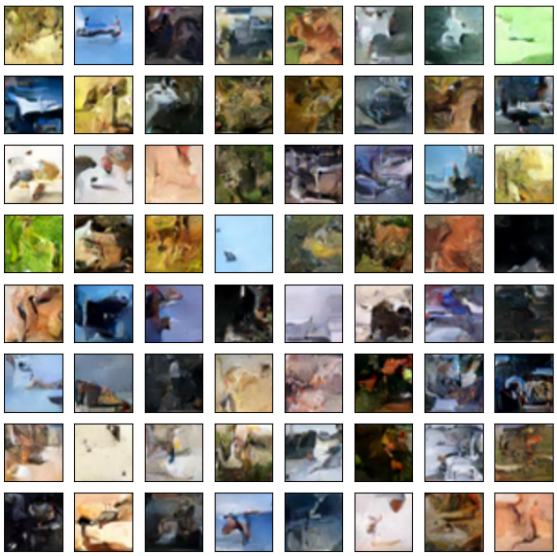

Figure 12: CIFAR-10 samples

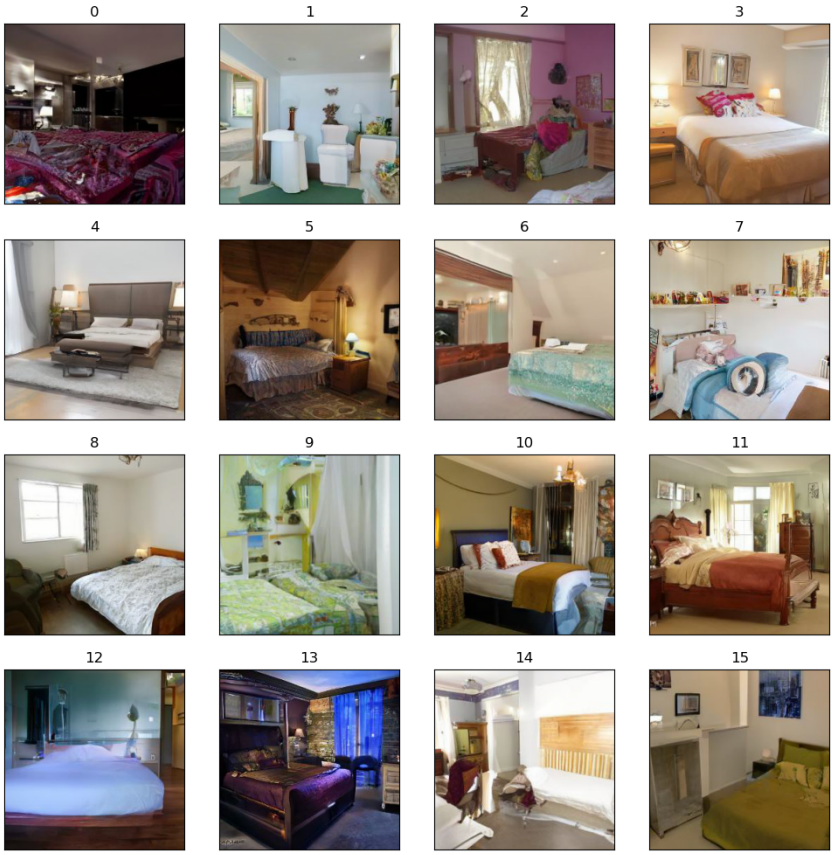

Figure 13: LSUN-bedroom samples

B 2-Channel-Model samples

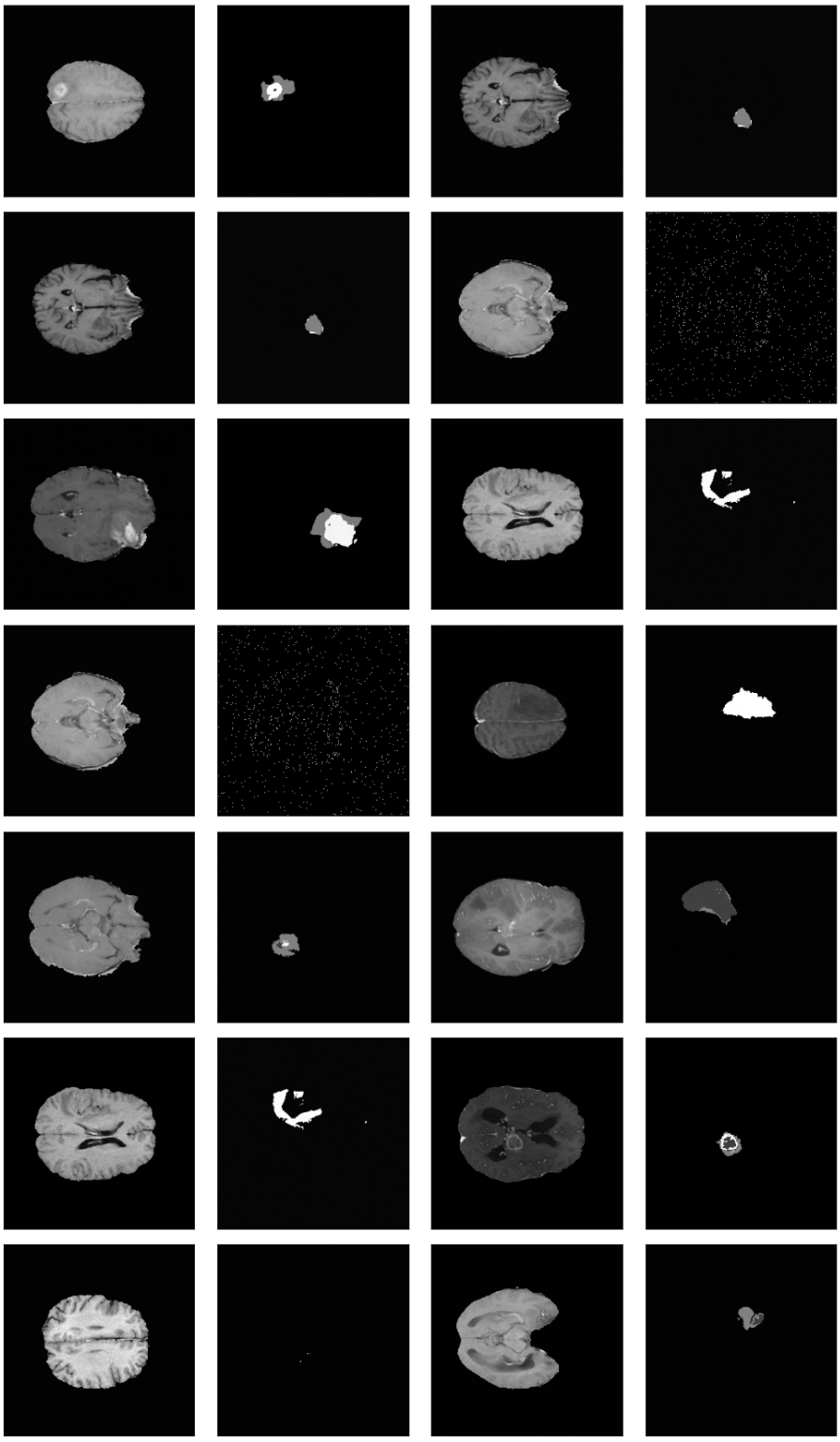

Figure 14: Samples of 60K EMA 2-Channel-Model

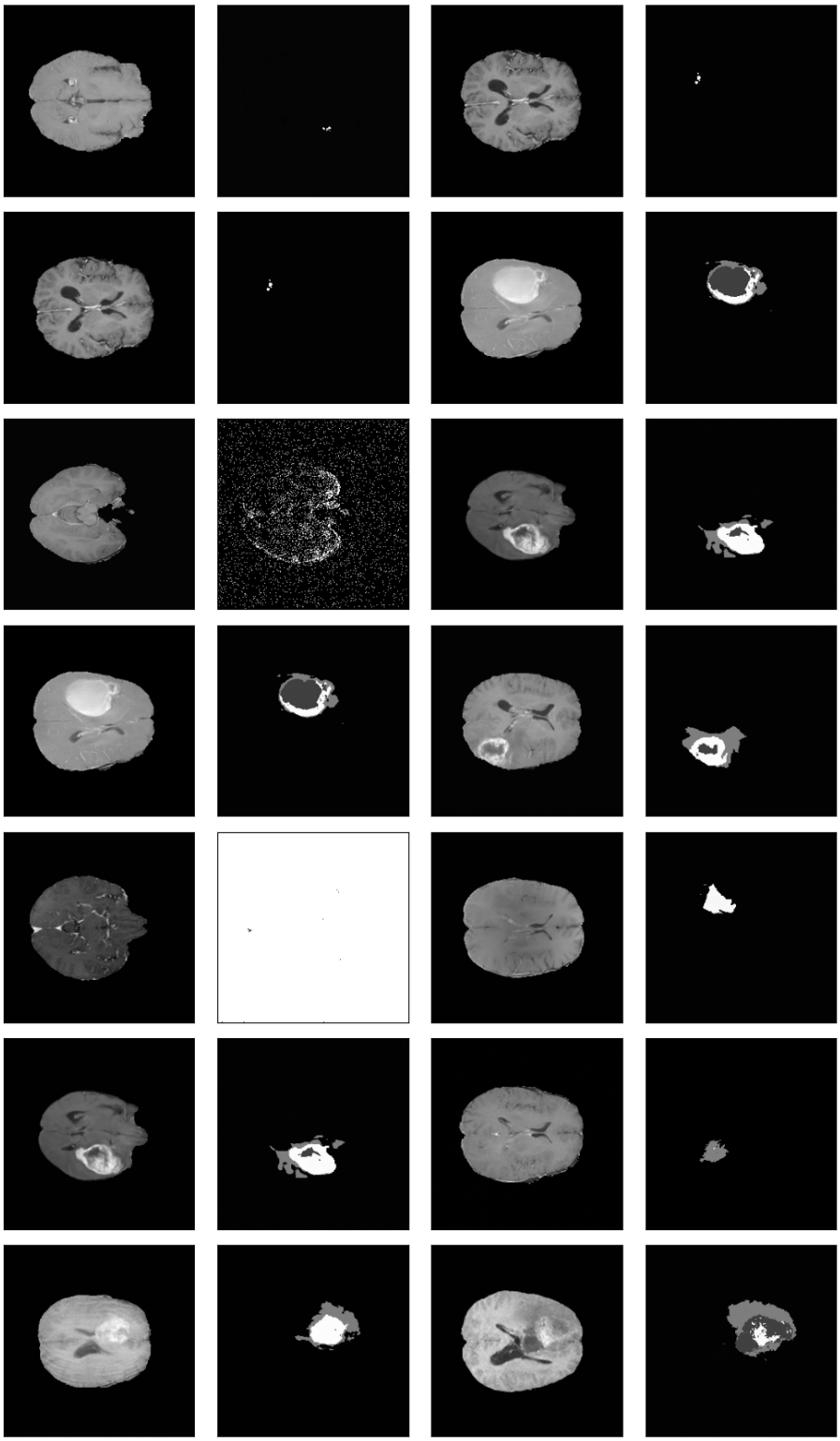

Figure 15: Samples of 60K Model 2-Channel-Model

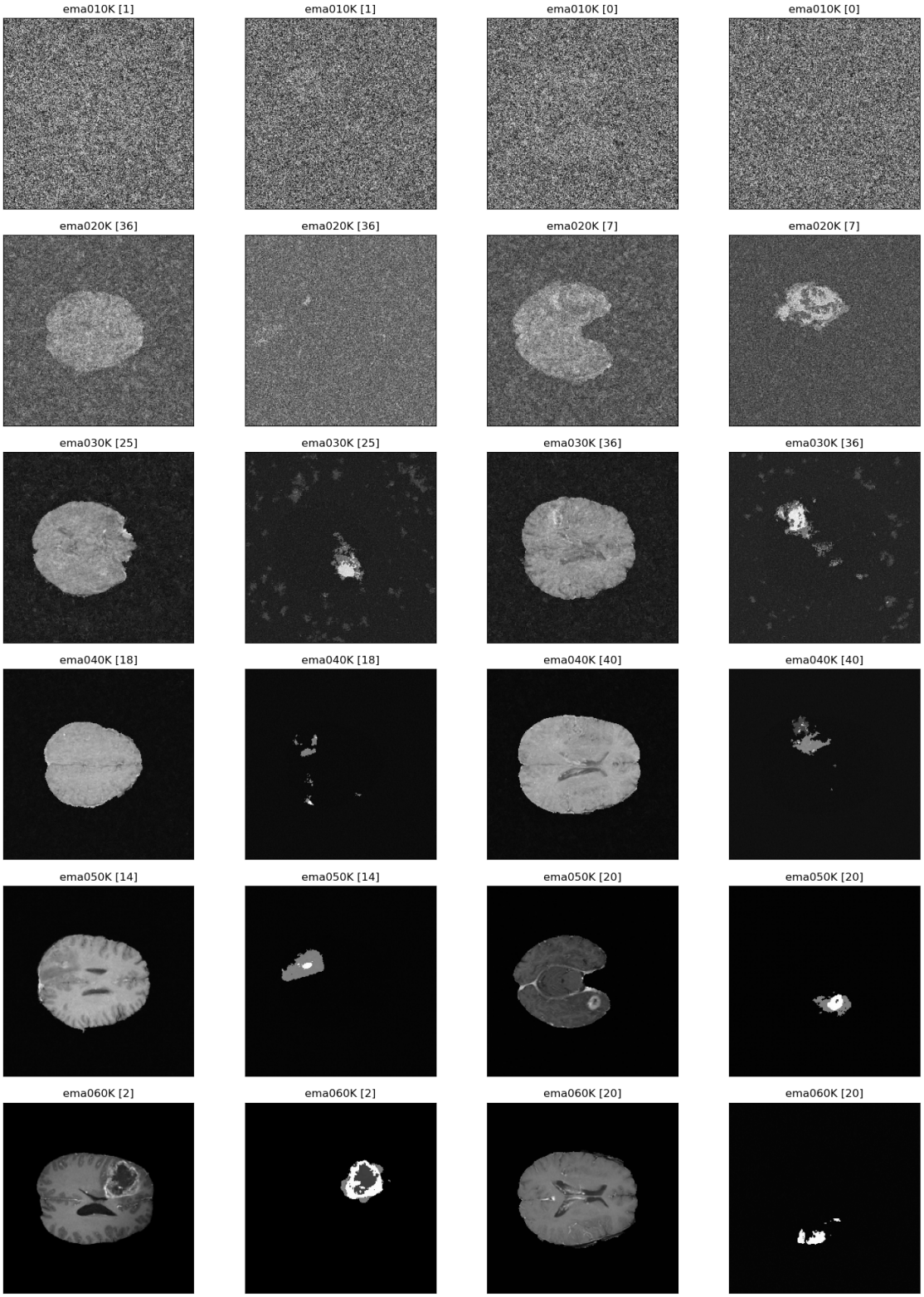

Figure 16: Experiment 4 model evolution over time

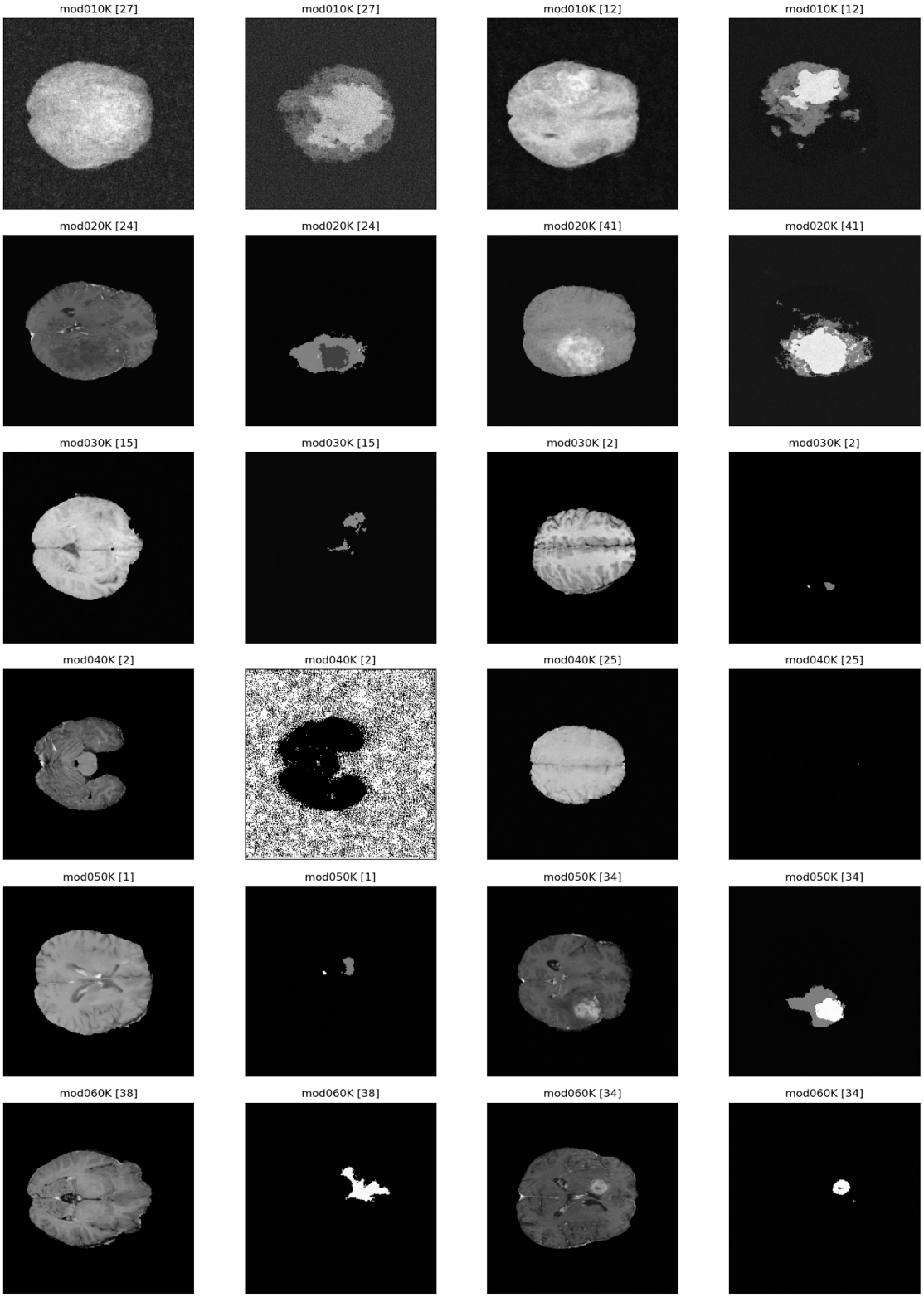

Figure 17: Experiment 4 model evolution over time

C 5-Channel-Model samples

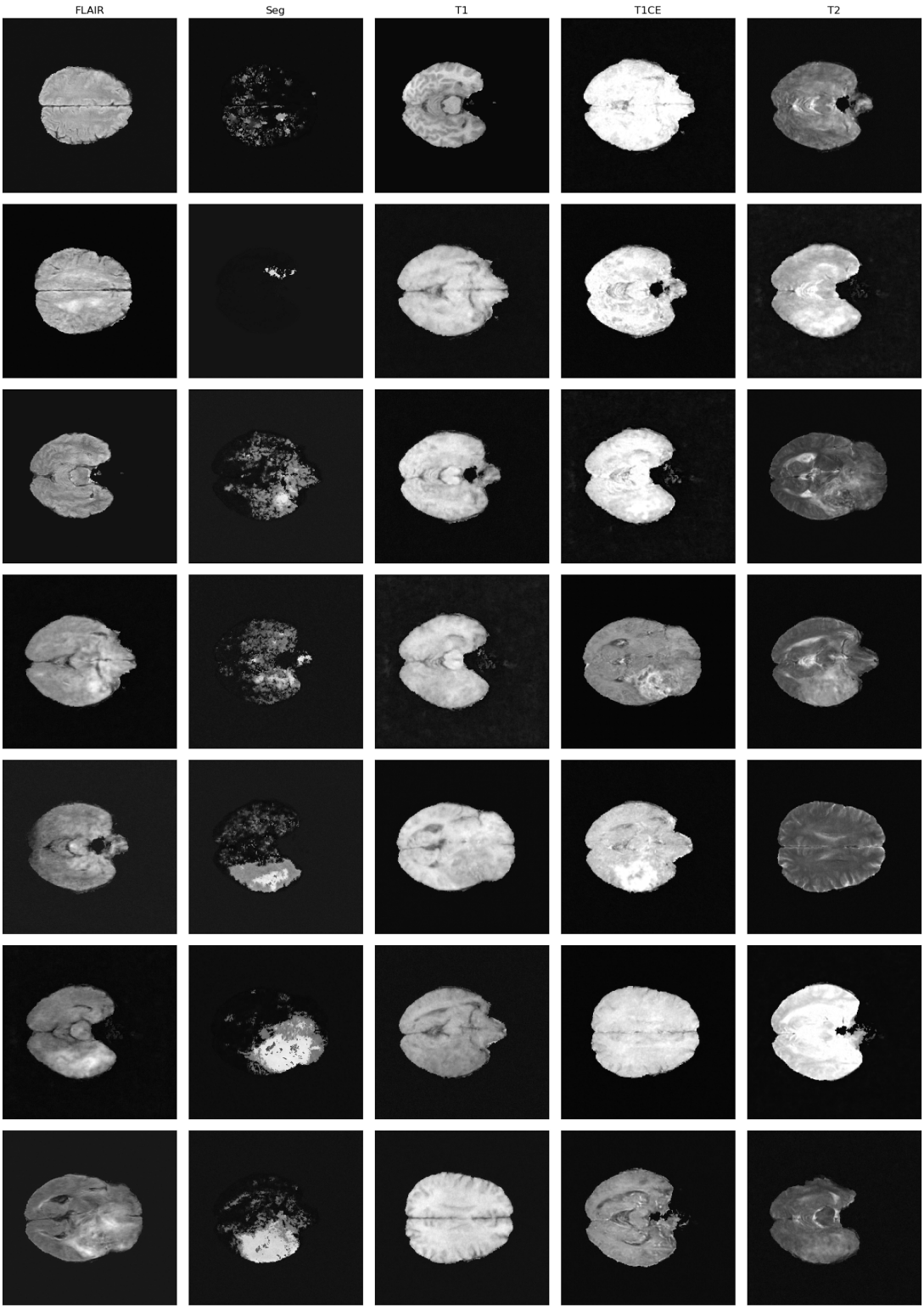

Figure 18: Samples from the 5-Channel 30K model checkpoint