How to Modify ControlNet Input: A Step-by-Step Guide for Input Modification, from 3 channels input to 5 (or more)

TL;DR

In this blog, we'll guide you through the technical process of modifying ControlNet's architecture to handle more complex inputs, such as concatenated masked images and masks. By following this step-by-step guide, you will learn how to adapt ControlNet's existing structure to support inputs beyond the default settings, allowing for greater flexibility in your diffusion model projects. This is a highly technical blog, and if you follow it, you'll be able to modify the number of channels in your ControlNet from 3 to 5 (or to any number you like). We'll cover everything from the rationale behind these architectural changes to detailed code modifications. Whether you're new to ControlNet or looking to fine-tune it for a specific use case, this blog will equip you with the necessary knowledge and tools to customize ControlNet effectively.

For more details - you are welcome to join our Discord channel

Disclosure

I'm currently the VP of Generative AI Technology at Bria. With a Ph.D. in Computer Vision under my belt and years of experience in generative AI, I've had the privilege of working extensively in this field. My expertise spans from training models (both fine-tuning and building from scratch) to writing pipelines and conducting practical, theoretical, and applied research across various areas of computer vision. While my experience strongly influences the content of this blog, I always strive to present information as objectively as possible, acknowledging any potential biases along the way.

Introduction

This guide aims to provide a comprehensive walkthrough on how to modify ControlNet to accommodate more complex input types, such as masked images and custom channels. Whether you're a developer, researcher, or simply interested in pushing ControlNet's

boundaries, this guide will equip you with the necessary technical understanding to make precise architectural changes. By the end of this process, you'll be able to adapt ControlNet for specific use cases, learning how to implement these modifications step by step. If you're interested in diving deeper into the theory behind ControlNet, be sure to check out this blog post.

A Few Words About ControlNet

ControlNet is a platform for training diffusion models that generate images based on both textual descriptions and a conditioning image.
ControlNet comes “off-the-shelf,” ready to handle inputs like depth maps, edge maps (think Canny), pose diagrams,color grids, and a variety of other models and templates you can find in Hugging Face .

Now, I’m guessing you landed on this blog because you want to use ControlNet, and you probably have a very specific goal in mind.
If the community has already created the ControlNet model you need—awesome! Just grab it and start experimenting. But what if the ControlNet model you need doesn't exist yet?
What if you want to use a specific conditioning image that no one’s thought of before?

https://i.giphy.com/media/v1.Y2lkPTc5MGI3NjExbmtjeTI4M3RmYnViZTI2Y3FvMmlhdDljanI3MGIlZ2JkbzhpcGQxZCZlcD12MV9pbnRlcm5hbF9naWZfYnlfaWQmY3Q9Zw/QBA3nFqgBkFB9EwieD/giphy.gif

What if you want to use a specific conditioning image that no one’s thought of before?

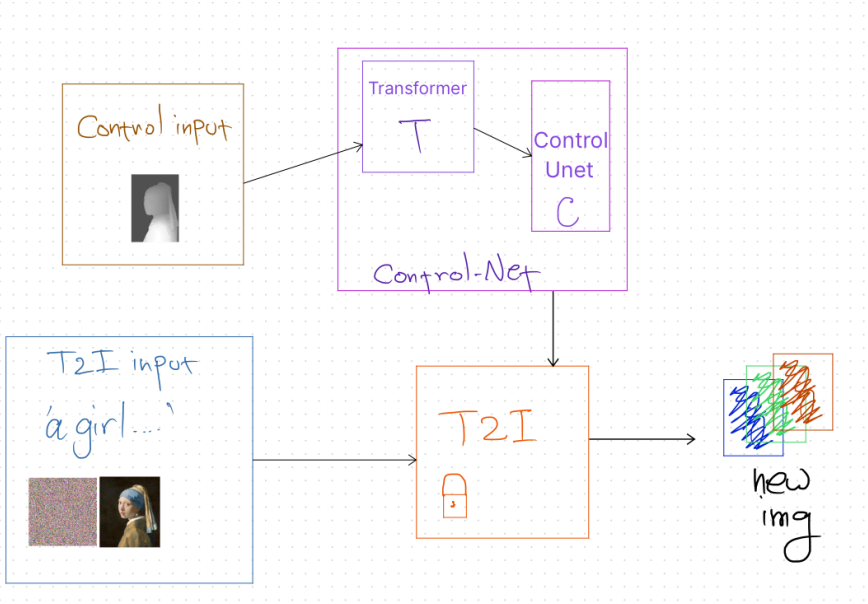
Ah, here’s where the plot thickens—but this is where it gets really interesting (and fun!). This is your chance to get creative and work some magic!
What’s the catch? Well, to do complex things, you need to understand the ControlNet platform deeply. Sure, things might get a bit complex, but that’s the fun part, right?

Whether you're here for a complex deep dive or just a quick win, this blog has got you covered.

Technical Basic Details About ControlNet You Should Know Before Getting Started

The ControlNet platform consists of four main components:

1. **Input Component to the Foundation Text-to-Image (T2I) Model** (blue) – This component passes along a noisy image and text prompt to the foundation model.
2. **Foundation T2I Model** (orange) – This model receives the noise and text (and tensors from the control net) processes them, and generates a new image as output.
3. **Input Component to the Control Model** (brown)– This component passes a conditioning image, such as a depth map or an edge map (like Canny), to the control model.
4. **Control Model Component** (purple) – This part processes the conditioning image through internal layers (of the Transformer) and produces a tensor. This tensor is then passed through the Control-UNet layers and integrated directly into the convolution and attention layers of the Foundation T2I model.



Four main components consisting the ControlNet platform

When it comes to technology, I love to dig deep—exploring the inner workings of these models, popping the hood, peeking behind the scenes, exploring the inner workings of these models, tearing apart the engine (have I run out of metaphors yet?). I've dedicated an entire blog to this kind of exploration, focusing more on the theory behind ControlNet. It offers a different perspective that helps explain the reasoning behind the changes we're about to make.

But this blog? It's going to be more technical. Here, I'll guide you through how to actually tweak ControlNet to fit your needs, and we'll dive into how to implement changes to replace backgrounds in images.

So, let's roll up our sleeves, grab the wrench (or keyboard), and get technical. Let's begin!

Why change the architecture?

Well, if you've landed on this blog, you probably already have a reason in mind. But let me give you a few examples. Sometimes the product you're developing requires input types that the basic ControlNet isn't designed to handle. For instance, instead of just a single-channel input, you may want to combine multiple inputs—like a color image plus a mask, or a combination of a color image, a depth map, and an edge map. These inputs are more complex, often involving

four or more channels (I'm not talking about connecting multiple ControlNets; I'm talking about changing the input structure for a single ControlNet!). To handle such inputs, you'll need appropriate data for training and adjustments to the platform and code to accommodate this new structure.

For our example, let's focus on something concrete. I'll show you how to tweak the ControlNet platform to accommodate an input of five channels like we did for the inpainting use case in this blog. This is what I did with it, but the possibilities are endless. You could train models and develop products for all kinds of applications with a similar approach.

Ready-to-Use Code and Models from Bria

Before we jump into the technical details, it's worth mentioning that at Bria, we've already implemented these changes. Our team has prepared ready-to-use code with all the necessary modifications, and we've trained control-net models (with unique input size) on large, legally-sourced datasets. If you want to fine-tune these models, you can simply download them from Bria's website in Hugging Face and get started right away. Please note that the models are completely free for academic use, just fill out this form and we will send them to you

As for Diffusers—you're probably familiar with it. It's the go-to library for working with diffusion models, so I won't bore you with the details. Because it's such a standard in the community, we at Bria made sure all our models are fully compatible with Diffusers. Whether you're developing or running models in production, it's the backbone we rely on to make your work easier.

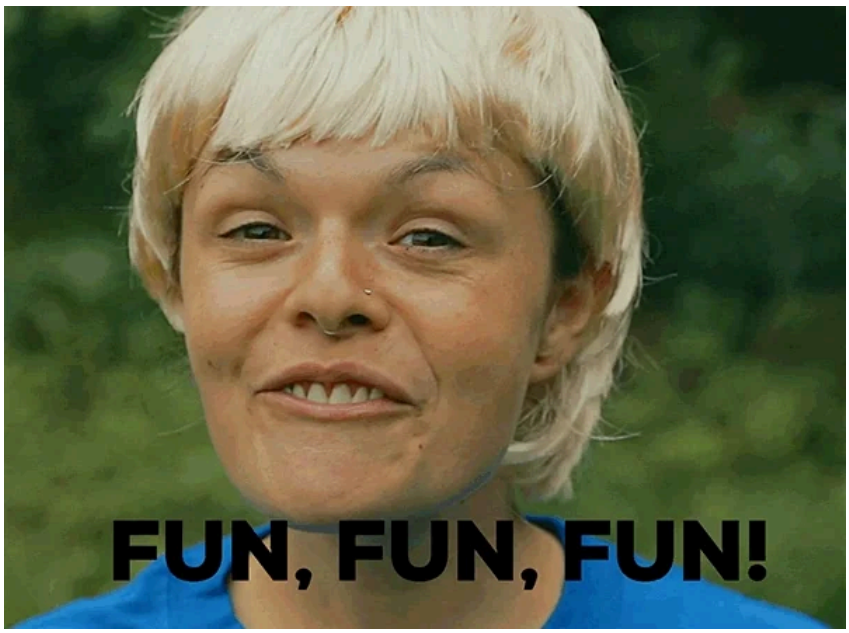
Technical Part

What's Going On in the ControlNet Training Code?

ControlNet Training Code

Let's start by taking a look at the script provided by Diffusers for **training** ControlNet: train_controlnet_sdxl. So, what's actually happening here? This script is designed to train the ControlNet model using `diffusers` library. Before we jump in and make changes, let's take a

moment to understand what the code is doing—because, trust me, you don't want to dive in without knowing what's under the hood (debugging is fun... said no one ever).



debugging is fun... said no one ever

https://giphy.com/gifs/ZO7Y7ssNfYEN4LQYCj

Basically, this script trains the ControlNet model—pretty standard stuff. You've got the usual suspects: importing the necessary libraries, setting up some parameters, configuring the optimizer and scheduler, and handling the data with the DataLoader.

Since this is a ControlNet training script, it also loads the various models you'll need, like the foundation model, VAE, and of course, ControlNet itself. The VAE is the standard one you're used to, and for the ControlNet-UNet, you've got options: you can either use a sub-copy of the foundation model or bring in a control-specific UNet and fine-tune that as suggested here.

Then comes the main event—the training loop. This is where the script adds noise to the latent variables, uses ControlNet for conditioning, and calculates the loss to guide the control net model's learning (or at least give it a good push in the right direction).

Controlnet.py

The training script calls the `ControlNetModel`, which defines the architecture and functionality of the model, including the components needed to handle conditioning images and incorporate

them into the image generation process. When `ControlNetModel` is imported into the training script, it loads this architecture of the models.

Essentially, the architecture code defines the model, while the training script performs the actual model training, making them work together. I know I'm not breaking any new ground here for most of you, but it's important to be aligned here - because we will change these 2 scripts

The `ControlNetModel`, also defines key classes in ControlNet, including `ControlNetConditioningEmbedding`, which processes conditioning images and integrates them into the model.

ControlNetConditioningEmbedding:

This class is a part of the `ControlNetModel` and it is responsible for implementing a **small** neural network that converts conditioning images into a latent format that the model can learn from and use during image generation. It plays a crucial role in embedding conditions into the model for training.. In the diagram attached above, I called it "The Transformer" because it transfers the condition input from the pixel space to the latent space

The Modular Architecture Behind ControlNet

The code contains the full architecture of the ControlNet model, from the basic definition of the different models to how they work together to form the entire "control net platform". ControlNet is an extension of the foundation Diffusion model that allows the use of conditioning images to influence the image generation process, enabling more precise control over the final result.

The architecture is modular, enabling the integration of different building blocks, flexibility in choosing various embedding methods, and the ability to customize layers to meet the specific requirements of any application.

In essence, this code provides the foundation for all the necessary modules for ControlNet and serves as the overall architecture to manage the diffusion process and the associated conditions, aiming to generate images that align with the given input conditions.

Technical Part 2: What Needs to Be Changed

In this section, we will explain the reasoning and logic behind the necessary changes in the ControlNet code to accommodate more complex inputs, such as masked images. Understanding these concepts will help you better grasp the modifications we'll implement later. For a more detailed explanation of the theory behind these changes, you can refer to this blog post. The detailed technical implementation of these changes will follow in Technical Part 3.

What we did was update the code to fit our specific needs—specifically, to change the size and type of input.

Now, we're going to do two things:
First, I'll provide a link to the updated ControlNet Python file where these changes have already been implemented. You can simply take the script I'm sharing here, replace it with your existing architecture script, and it should work pretty smoothly.

Second, I'll explain the changes in detail. These explanations will help you implement the modifications yourself, if you're able to follow along with the steps in this blog. This is actually the preferable method, since I don't know which version of ControlNet you're working with or what updates may have been made since I implemented these changes.

What We're Changing:

1. **Change the input size**
2. **Modify stride size**
3. **Add a custom embedding layer**
4. **Create custom data**
5. **Encode the custom data with VAE**
6. **Update the control image**
7. **Data mormalization**

Breaking It Down: Why We're Making These Changes

Overview of ControlNet Modifications

In this section, we'll provide a brief overview of the general modifications needed to adapt ControlNet to handle more complex inputs, such as masked images and masks. These changes will allow ControlNet to process multi-channel inputs and enable more flexibility in its applications. We'll also explain why these changes are necessary and how they improve the model's performance. This overview will set the stage for the detailed technical steps that follow.

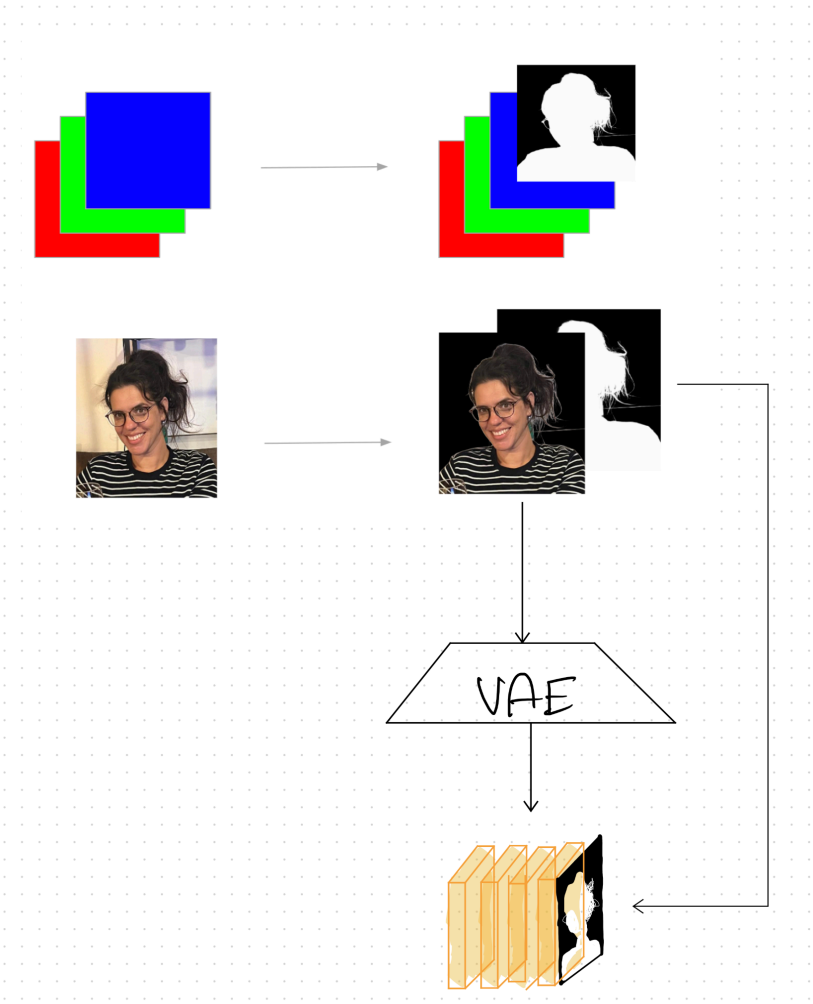
1.Why Change the Input Size?

We want to change the input size because (in our example) we'll be inputting a *masked image+mask*, which means we need to increase the input size to 5 channels.

If it's a masked image, you're probably thinking, "Wait, isn't that 4 channels? RGB + mask?"

Well, not quite. It *starts* as 4 channels, but when the masked image passes through the VAE, it gets encoded into 4 latent channels, and we'll add one more channel for the mask.

Let's visualize this with the simple diagram "**Inpainting Training input: RGB Image and Mask Concatenation in ControlNet",** In the top row of the diagram, we see an RGB color image concatenated with a corresponding mask, generated using Bria's RMBG 1.4 model (alpha mask). This combination transforms the input into a 4-channel representation. In the bottom row, we observe the process that the conditioning input (the masked image) undergoes in ControlNet, both during training and inference. The masked image is passed through the VAE, and its output—containing 4 channels—is then concatenated with the resized mask, resulting in an actual input of 5 channels.



Inpainting Training input: RGB Image and Mask Concatenation in ControlNet

Top row: An RGB image is concatenated with a mask, turning it into 4 channels. Bottom row: The conditioning image (masked image) goes through the VAE, producing 4 channels. After concatenation with the resized mask, the input expands to 5 channels.

To sum up: we want to use the masked image latents as control input, along with the concatenated mask, so the input size should be 5 channels instead of 3.

Note :ControlNet usually takes inputs like depth maps or Canny edges. Although these seem like single-channel inputs (e.g., grayscale depth or edge maps), they're repeated three times, which is why in the training code you're familiar with, the conditioning default input size is set to

3. What’s actually happening is that it’s receiving a repeated single channel generated by the depth map or Canny algorithm

2. Why Modify the Stride Size

Let’s first remind ourselves that the control input needs to "meet" the diffusion model at some point, meaning the sizes must match up. Also, diffusion models operate in the latent space, where the size is much smaller compared to the pixel space (typically reduced to 128x128 instead of 1024x1024 in pixel space).

In the default implementations of ControlNet, the input size is typically 1024x1024, which is quite large, so it needs to be downsampled. This is done with the stride. In the original code, the stride is set to 2, because we need to reduce the size of the tensor somehow.

Why We’re Changing the Stride:Preparing for VAE Integration In step 5)

But here’s what we’re doing differently: instead of feeding a 1024x1024 image directly into ControlNet, we’re passing it through the VAE first, and the output from the VAE (which is already downsampled) is what we’ll feed into ControlNet. Since the VAE output is already small, we don’t need to downsample it again, which is why we’re changing the stride to 1.I’ll go into more detail about the VAE process later in this document.

Brief Explanation of VAE

Before we dive into the technical aspects, it's crucial to understand how the Variational Autoencoder (VAE) works and why it's central to this process. The VAE is responsible for converting input images into a compressed latent space, allowing diffusion models to work efficiently. In the context of ControlNet, the VAE is used to encode both the original and masked images, which are then processed by the model. The VAE ensures that the inputs are properly scaled and aligned with the architecture of ControlNet. If you’d like to explore more about the VAE and its role in generative AI, check out this detailed blog.

3. Add a custom embedding layer

We'll add a custom embedding layer for the conditioning images in the `ControlNet` model, overriding the default embedding initialization.

In the original version of the code, when the `ControlNet` model is loaded (either from a `UNet` or from a pre-trained model), a default embedding layer, which I've referred to earlier as the "Transformer," is created for processing the conditioning images. This "Transformer" plays a crucial role in converting (transforming) conditioning images into a latent format that the model can efficiently use during the diffusion process.

In this modification, we'll create a new custom embedding layer to handle the conditioning images, replacing the existing "Transformer" embedding layer. We'll also specify the number of output channels for this layer and define the input size accordingly.

4. Create custom data

Preparing the Data: Masked Images and Masks

To effectively replace a background in an image using diffusion models, the first step is creating a mask that separates the foreground object from the background. Using Bria's Background removal model, we generate a masked image where the background is black (And the front object remains unchanged). This mask is then paired with the masked image, forming a concatenated input that will be fed into the ControlNet model. It's important to prepare the data carefully, as the quality of both the mask and the masked image significantly influences the model's output.

This data preparation step is essential to make sure that the mask is aligned with the masked image, allowing the VAE to encode them properly. It's important to note that this modification won't be made in the `controlnet.py` script, which handles the architecture, but rather within the training script itself, where we manage the data flow and transformations. Preparing the data accurately here is key to the success of the model.

This is the specific example I've chosen to guide you through in this blog, for a more detailed explanation of the data preparation process, you can check out the previous blog post [here]. If you're working with a different input type, make sure to make the necessary adjustments for your use case.

5. The Role of the VAE in Image Encoding

Now we're getting to what we discussed earlier—if we want to reduce the dimensions of our input using the VAE, it's time to put that into action. The VAE is a great fit here, both from a technical and theoretical standpoint. And if you're in the mood to dig deeper into the theory behind it, I've got you covered—check out this blog where I explain everything in detail.

6. Update the control image

To ensure that ControlNet can handle the concatenated masked image and mask, we need to adjust the input. First, we resize the mask to match the dimensions of the latents (the compressed representation from the VAE). Then, we concatenate the resized mask with the masked latents, creating a control image that includes both elements. This updated control image is now ready to be processed by ControlNet, ensuring both inputs are properly integrated during the image generation process.

Technical Part 3: Let's Change

In the previous section, we covered the logic and reasoning behind the required modifications to adapt ControlNet for more complex inputs. Now, we'll move on to the practical application of these changes. Here, you'll find the full code implementation to help you apply the changes in your own projects, tailored to your specific needs.

1. Implementing Input Size Changes in Code (in the model architecture script)

We'll be updating the `conditioning_channels` parameter from 3 to 5 in the `ControlNetConditioningEmbedding` class.

Inside this classControlNetConditioningEmbedding, `conditioning_channels` is used to define how many input channels will be passed into the first convolution layer (`conv_in`) and the subsequent layers within the embedding process.

Note: The `conditioning_channels` parameter appears twice in the code—once during the initialization of the `ControlNetConditioningEmbedding` class and once during the initialization of the `ControlNetModel` class. We need to change it specifically inside `ControlNetConditioningEmbedding`.

Example of how to change the code:

```python
class ControlNetConditioningEmbedding(nn.Module):

    def __init__(
        self,
        conditioning_embedding_channels: int,
        conditioning_channels: int = 5,
        block_out_channels: Tuple[int, ...] = (16, 32, 96, 256),
    ):
```

2. Modifying the Stride Size in the Architecture (in the model architecture script)

In the `ControlNetConditioningEmbedding, in the second conv layer:`

Python

```
class ControlNetConditioningEmbedding(nn.Module):

...


        for i in range(len(block_out_channels) - 1):
            channel_in = block_out_channels[i]
            channel_out = block_out_channels[i + 1]
            self.blocks.append(nn.Conv2d(channel_in, channel_in,
kernel_size=3, padding=1))
            self.blocks.append(nn.Conv2d(channel_in, channel_out,
kernel_size=3, padding=1, stride=1))
```

3. Adding a Custom Embedding Layer for Conditioning Images(in the training script)

Python

```
if args.controlnet_model_name_or_path:
    logger.info("Loading existing controlnet weights")
    controlnet =
ControlNetModel.from_pretrained(args.controlnet_model_name_or_path)
else:
    logger.info("Initializing controlnet weights from unet")
    controlnet = ControlNetModel.from_unet(unet)
```

```
## Add:
controlnet.controlnet_cond_embedding =
ControlNetConditioningEmbedding(
                conditioning_embedding_channels=320,
                conditioning_channels=5,
            )
```

Note :

When continuing with fine-tuning on a model that has already been trained with architectural modifications (such as changes to input and output channels in `ControlNetConditioningEmbedding`),**there is no need to redefine the layer** with these lines. This is because, when the trained model is saved, both the updated architecture and the newly trained weights are preserved, including any replaced or modified layers. Therefore, when loading the trained model for further training, all layers and weights are present exactly as they were in the previous training, so there is no need to reapply the architectural changes made earlier.

However, if starting with a new model or re-initializing it from weights that do not include the modification (such as a generic model without the adapted layer for 5 input channels and 320 output channels), you will need to redefine the layer to apply the architectural change again.

4. Create custom data (in the training script)

To prepare the custom data for inpainting, we'll create a masked image where the background (or the area that needs to be inpainted) is black, represented by a value of 0. This ensures that the inpainting model focuses on the correct areas. By using the `image_mask`, we can selectively mask out the background:

Python
```
masked_images[:,image_mask < 0.5] = 0  # mask background
```

This step prepares the data properly, allowing the model to handle the inpainting task with the desired focus on the masked areas.

Note that image_mask need to come from the dataloader while masked_images can be created on the fly

5. Encoding and Scaling the Latents Using the VAE

(in the training script)

To encode both the original and masked images using the VAE, we first pass the original image through the VAE to convert it into its latent representation. This latent representation, which remains unchanged, is later used with added noise for the diffusion process as part of the T2I Foundation model (which stays the same and is not modified). This latent feeds both the UNet and ControlNet models. Next, we do the same for the masked image, converting it into its own latent space. This step is specific to our use case, as the masked image serves as the conditional input for ControlNet. Both the original and masked latents are scaled by the VAE's scaling factor to ensure they fit properly within the model's architecture.

Python

```
# Encode the original image into the latent space
latents = vae.encode(
    pixel_values.to(vae.dtype)
).latent_dist.sample()  # Output dimensions: [1, 4, 64, 64], latent
representation of the original image

# Scale the latents to match the model's expected input size
latents = latents * vae.config.scaling_factor

# Encode the masked image into the latent space (specific to our
use case)
masked_latents = vae.encode(
```

```
    masked_images.to(vae.dtype)
).latent_dist.sample()  # Output dimensions: [1, 4, 64, 64], latent
representation of the masked image

# Scale the masked latents to ensure they align with the model's
architecture
masked_latents = masked_latents * vae.config.scaling_factor
```

6. **Updating the Control Image for ControlNet**

(in the training script)

To fully integrate the mask and the latents (i.e., the VAE output), we need to adjust how we process the control image. First, we resize the mask to match the dimensions of the latent space, and then we concatenate the resized mask with the masked latents. This step ensures that ControlNet receives the proper input and can effectively handle both the latents and the mask.

Here's the code to do this:

Python

```
## Add:

# Resize the mask to match the latent dimensions
masks = torch.nn.functional.interpolate(masks, size=(1024 // 8,
1024 // 8))
```

```
# Concatenate the latents and the resized mask
controlnet_image = torch.cat([masked_latents, masks],
dim=1).to(dtype=vae.dtype)
```

At this point, the mask has been resized to match the latent space (as we're working with a reduced resolution in the latent space), and the mask is now concatenated with the masked latents. The result is a control image (`controlnet_image`) that contains both the latents and the mask, ready to be passed into ControlNet.

Next, we use this updated control image in the ControlNet forward pass:

Python

```
# Resize the mask to match the latent size
masks = torch.nn.functional.interpolate(masks, size=(1024 // 8,
1024 // 8))

# Concatenate the latents and the mask
controlnet_image = torch.cat([masked_latents, masks],
dim=1).to(dtype=vae.dtype)  # bs, 5, 64, 64

# Forward pass in ControlNet with the updated control image
down_block_res_samples, mid_block_res_sample = controlnet(
    noisy_latents,
    timesteps,
    encoder_hidden_states=batch["prompt_ids"],
    added_cond_kwargs=batch["unet_added_conditions"],
    controlnet_cond=controlnet_image,
    return_dict=False,
)
```

In this part of the code, we pass the concatenated `controlnet_image` (which includes both the latents and the mask) as the `controlnet_cond` argument in ControlNet's forward pass. This ensures that ControlNet processes both inputs correctly, allowing the latents and the mask to work together to influence the image generation process effectively.

Tips for Avoiding Mistakes When Modifying ControlNet's Architecture

During our work on this project, we encountered a few pitfalls that could save you time if you learn from our experience. Here are some key insights:

Using Only the Mask or Only the Masked RGB Image

Initially, we experimented with creating a control network using the default architecture (similar to Canny-ControlNet) by inputting only the masked RGB image. While this approach didn't work well for our specific use case, it might work for others. In our scenario, we found that providing both the mask and the image was essential for giving the model enough information to effectively "guide" the generation process. The combination of these inputs offered significantly better control and results, but depending on your use case, you might see different outcomes.

Correctly Modifying the `conditioning_channels` Parameter

The `conditioning_channels` parameter appears twice in the code—once during the initialization of the `ControlNetConditioningEmbedding` class and once during the initialization of the `ControlNetModel` class. It's important to modify this parameter specifically in the `ControlNetConditioningEmbedding` class. Changing it elsewhere will not have the desired effect.

Modifying the `stride` in the Convolutional Layer

When working with the convolutional layers in `ControlNetConditioningEmbedding`, it's essential to pay attention to where you modify the `stride`. Since there are several convolutional blocks in the architecture, it can be confusing.

Make sure to update the `stride` in the second convolutional layer within each block. Here's the specific line where the change happens This adjustment ensures that the input size matches the latents from the VAE. Be careful to apply this stride modification consistently across all necessary layers to ensure the input flows correctly through the architecture.

Optimize Performance with Mixed Precision

When working with large models like ControlNet, using mixed precision can significantly speed up training and reduce memory consumption without sacrificing accuracy. Make sure to enable mixed precision by setting the model to run with `mixed_precision="bf16"` if you're using compatible hardware (like NVIDIA A100 GPUs). This will allow the model to perform computations using 16-bit floating point precision, optimizing memory usage while maintaining the necessary precision for key operations

Normalization

We found that “normalizing” the data works well for us. If you're planning to fine-tune and are using Bria's -ControlNet-Background-Generation as your base model, you must also normelise the data as we did:

```python
def get_control_image_tensor(vae, image, mask)->torch.Tensor:
        masked_image, image_mask = get_masked_background_image(image, mask)
        masked_image_tensor = torch.from_numpy(masked_image)
        masked_image_tensor = (masked_image_tensor - 0.5) / 0.5 # normalize for
vae
        masked_image_tensor = masked_image_tensor.unsqueeze(0).to(device="cuda:0")
        # encode the image to get the control latents
        control_latents = vae.encode(
                masked_image_tensor[:, :3, :, :].to(vae.dtype)
            ).latent_dist.sample()
        control_latents = control_latents * vae.config.scaling_factor

        mask_tensor = torch.tensor(image_mask, dtype=torch.float32)[None, None,
...].to(device="cuda:0")
        mask_tensor = torch.where(mask_tensor > 0.5, 1.0, 0) # binarize the mask
        mask_resized = torch.nn.functional.interpolate(mask_tensor,
size=(control_latents.shape[2], control_latents.shape[3]), mode='nearest')
        control_tensor = torch.cat([control_latents, mask_resized], dim=1)
        return control_tensor
```

The function `get_control_image_tensor` is designed to create a tensor based on a masked image, which can be used in both training and inference processes. Note that one of the critical steps in this function is the **binarization** of the mask, where any value above 0.5 is converted to 1, and all values below are set to 0. I found this step essential because it creates a sharp and clear distinction between object areas and background areas in the image, which improved the results. Intermediate values in the mask can confuse the model, causing it to process parts of the background or ignore object areas in an unintended way. Thus, binarization ensures that the model precisely understands which areas require processing and which should be defined as background, providing the clear separation needed to maintain consistent and accurate results.

If you're using a different base model or training (Bria) from scratch, then I encourage you to consider making this change, as we've found it improves results.

Wrapping It All Up

By now, you've gained insight into the inner workings of ControlNet and learned how to adapt it for more complex inputs and custom use cases. From adjusting the input size to integrating a custom VAE process, you've got a solid foundation to tackle more advanced modifications and optimizations. Remember, experimenting and fine-tuning is key, and don't be afraid to make mistakes along the way—they're part of the learning process. With these tips and techniques, you're well-equipped to create a more versatile and powerful ControlNet for your unique applications. Good luck, and happy coding!

And hey, before you dive back into your code, a friendly reminder: Bria's offerings are more than just our Foundation models. We've got the full package- it is a full open source: architecture setups, code, weights—everything you need. Once you sign up on our platform, everything is wide open. Plus, you're not going at it alone. You'll have guidance from our Solution team, R&D, and from me. So, don't hesitate to reach out—we're here to help!

If you're in academia, just sign up here, and it's all yours (**free**!). If you're in the industry—sign up here (and yeah, you'll need to pay), but once you do, everything's open for you too. We've got you covered either way, so jump in and let's build something amazing together!