main

February 4, 2025

[1]:
```
import os
import glob
from tqdm import tqdm

os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
import silence_tensorflow.auto
import tensorflow as tf

gpus = tf.config.experimental.list_physical_devices("GPU")
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)

config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = 1
config.gpu_options.per_process_gpu_memory_fraction = 1
session = tf.compat.v1.InteractiveSession(config=config)

# tf.debugging.enable_check_numerics()

import keras
import random
import numpy as np
import matplotlib.pyplot as plt
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
E0000 00:00:1738532209.480160  593088 cuda_dnn.cc:8310] Unable to register cuDNN
factory: Attempting to register factory for plugin cuDNN when one has already
been registered
E0000 00:00:1738532209.486506  593088 cuda_blas.cc:1418] Unable to register
cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has
already been registered
I0000 00:00:1738532213.195721  593088 gpu_device.cc:2022] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 3794 MB memory:  -> device: 0,
name: NVIDIA GeForce RTX 3050 Laptop GPU, pci bus id: 0000:01:00.0, compute
capability: 8.6
```

[2]:
```python
H = 256
W = 256
C = 3

# P = 16
P = 8
assert(H == W)
assert(H % P == 0)


h = 4

# D_model = 1024
D_model = 512
# D_head = 128
D_head = 64
# D_fcn = 2048
D_fcn = 1024
# num_layers = 4
num_layers = 2

N = (H * W) // (P * P)
BATCH_SIZE = 16
FLOAT = tf.float32
```

[3]:
```python
def viz_img(img):
    img = tf.cast(img, tf.float32)
    plt.imshow(tf.squeeze(img).numpy(), cmap="gray")
    plt.colorbar()
    plt.show()


def viz_mask(mask):
    plt.imshow(tf.squeeze(mask).numpy(), cmap="gray", vmin=0, vmax=1)
    plt.colorbar()
    plt.show()
```

[4]:
```python
def random_visibility_mask():
    x1 = tf.random.uniform(shape=(), minval=0, maxval=W - 100, dtype=tf.int32)
    y1 = tf.random.uniform(shape=(), minval=0, maxval=H - 100, dtype=tf.int32)
    x2 = tf.random.uniform(shape=(), minval=x1 + 50, maxval=W + 1, dtype=tf.
 ↪int32)
    y2 = tf.random.uniform(shape=(), minval=y1 + 50, maxval=H + 1, dtype=tf.
 ↪int32)
    # tf.print(x1,x2,y1,y2)

    mask = tf.ones((H, W), dtype=tf.bool)
```

```python
    mask = tf.tensor_scatter_nd_update(
        mask,
        indices=tf.stack(
            [
                tf.repeat(tf.range(y1, y2), x2 - x1),
                tf.tile(tf.range(x1, x2), [y2 - y1]),
            ],
            axis=-1,
        ),
        updates=tf.zeros([(y2 - y1) * (x2 - x1)], dtype=tf.bool),
    )
    return tf.expand_dims(mask, -1)  # expand channel wise


ds_shape_advertised = (512, 512, 3)

def load_and_validate(file_path):
    img = tf.io.read_file(file_path)
    img = tf.image.decode_image(img, channels=C, expand_animations=False)
    img = tf.divide(tf.cast(img, dtype=FLOAT), 255.0)
    is_valid = tf.reduce_all(tf.equal(tf.shape(img), tf.
  ↪constant(ds_shape_advertised)))

    return img, is_valid


dataset_path = "/mnt/Data/ML/datasets/portraits"
num_samples = 1000


all_files = [
    os.path.join(dataset_path, f)
    for f in os.listdir(dataset_path)
    if f.endswith((".jpg", ".png"))
]
random.shuffle(all_files)
selected_files = all_files[:num_samples]
dataset = tf.data.Dataset.from_tensor_slices(selected_files)
dataset = dataset.map(load_and_validate)
dataset = dataset.filter(lambda img, is_valid: is_valid)
dataset = dataset.map(lambda img, is_valid: img)
dataset = dataset.map(lambda img: tf.image.resize(img, (H, W)))

with tf.device("/cpu:0"):
    valid_count = dataset.reduce(
        tf.constant(0, dtype=tf.int32), lambda x, _: x + 1
    ).numpy() # type: ignore
```

```
print(f"Valid images count: {valid_count}")
assert valid_count, "Everything's gone"

masks = [random_visibility_mask() for _ in range(valid_count)]
mask_dataset = tf.data.Dataset.from_tensor_slices(masks)

ds_masks = tf.data.Dataset.zip((dataset,mask_dataset))

train_count = int(valid_count * 0.8)
test_count = int(valid_count * 0.1)
val_count = valid_count - train_count - test_count

train_ds = ds_masks.take(train_count).batch(BATCH_SIZE)
test_ds = ds_masks.skip(train_count).take(test_count).batch(BATCH_SIZE)
val_ds = ds_masks.skip(train_count + test_count).take(val_count).
 ↪batch(BATCH_SIZE)

train_batches = -(train_count // -BATCH_SIZE)
test_batches =  -(test_count // -BATCH_SIZE)
val_batches =  -(val_count // -BATCH_SIZE)

print("T,T,V:",train_count, test_count, val_count)
```

```
I0000 00:00:1738532213.390807  593088 gpu_device.cc:2022] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 3794 MB memory:  -> device: 0,
name: NVIDIA GeForce RTX 3050 Laptop GPU, pci bus id: 0000:01:00.0, compute
capability: 8.6
```

```
Valid images count: 1000
T,T,V: 800 100 100
```

[5]:

```
def extract_patches(image: tf.Tensor) -> tf.Tensor:
    "R^{BS x H x W x C} -> R^{BS x N x P^2 x C}"

    patches: tf.Tensor = tf.image.extract_patches(
        images=image,
        sizes=[1, P, P, 1],
        strides=[1, P, P, 1],
        rates=[1, 1, 1, 1],
        padding="VALID",
    )
    BS, H_prime, W_prime, _ = tf.unstack(tf.shape(patches))

    # Reshape patches to [BS, H' * W', P*P, C]
    patches = tf.reshape(patches, [BS, H_prime * W_prime, P * P, -1])

    return patches
```

```python
def patches_to_imgs(patches: tf.Tensor) -> tf.Tensor:
    "R^{BS x N x P^2 x C} -> R^{BS x H x W x C}"
    BS = tf.shape(patches)[0]
    grid_size = H // P  # same as W // P
    patches = tf.reshape(patches, [BS, grid_size, grid_size, P, P, C])
    patches = tf.transpose(patches, perm=[0, 1, 3, 2, 4, 5])
    image = tf.reshape(patches, [BS, grid_size * P, grid_size * P, C])

    return image


sample = tf.expand_dims(next(iter(dataset.take(1))), 0)
tf.assert_equal(patches_to_imgs(extract_patches(sample)) , sample)

def create_attention_mask(obvmask: tf.Tensor):
    "R^{BS x H x W} -> R^{BS x N x N}"
    # TF does not support native min pooling.
    # The mask shown is OBSERVATION MASK meaning 0 means missing.

    BS = tf.shape(obvmask)[0]
    mask_pooled = tf.nn.max_pool2d(
        tf.cast(
            tf.logical_not(obvmask), dtype=tf.int8
        ),
        ksize=[P, P],
        strides=[P, P],
        padding="VALID",
    )
    mask_pooled = tf.logical_not(tf.cast(mask_pooled, tf.bool))
    # viz_mask(mask_pooled)
    mask_pooled = tf.reshape(mask_pooled, [BS, N])
    mask_expanded = tf.expand_dims(mask_pooled, axis=1)  # (BS, 1, N)
    mask_expanded = tf.tile(mask_expanded, [1, N, 1])  # (BS, N, N)
    A = tf.where(
        mask_expanded,
        tf.constant(0.0, dtype=FLOAT),  # zero penanly
        tf.constant(-float("inf"), dtype=FLOAT),  # inf penalty
    )
    return A

# create_attention_mask(tf.expand_dims(random_visibility_mask(),0))
```

[6]:
```python
commonDense = {"dtype": FLOAT, "kernel_initializer": "glorot_uniform"}

class PatchEmbedding(keras.layers.Layer):
```

```
    def __init__(self):
        super().__init__(dtype=FLOAT)
        self.proj = keras.layers.Dense(D_model, **commonDense)  # (P² * C) ->␣
↪D_model

    def build(self, input_shape):
        self.positional_embedding = self.add_weight(
            shape=(N, D_model),
            initializer='glorot_uniform',
            name='pos_embed'
        )

    def call(self, patches_flat: tf.Tensor):
        # R^{BS x N x (P^2 . C)} -> R^{BS x N x D_model}
        X = self.proj(patches_flat)
        X += self.positional_embedding
        return X


class MultiHeadAttention(keras.layers.Layer):
    def __init__(self):
        super().__init__(dtype=FLOAT)
        # Project to h * D_head dimensions
        self.W_Q = keras.layers.Dense(h * D_head, **commonDense)
        self.W_K = keras.layers.Dense(h * D_head, **commonDense)
        self.W_V = keras.layers.Dense(h * D_head, **commonDense )
        # Project back to D_model
        self.W_O = keras.layers.Dense(D_model, **commonDense)

    def call(self, X, A):
        # X: R^{BS x N x D_model}
        # A: R^{BS x N x N}
        # returns: R^{BS x N x D_model}

        # In the standard implementation, each head has its own separate␣
↪projection matrices. However, a common optimization is to project the input␣
↪into h * D_head dimensions (which is D_model) with a single large␣
↪projection, then split into h heads. So, if D_model = h * D_head, then using␣
↪a Dense(D_model) for Q, K, V and then splitting into h heads each of D_head␣
↪is equivalent to having h separate projections. This is a standard approach␣
↪because it's more efficient to compute all heads in parallel with a single␣
↪matrix multiplication rather than h separate ones.
        # So the optimal way is to use combined projections.
        Q = self.W_Q(X)  # (BS, N, h * D_head)
        K = self.W_K(X)  # (BS, N, h * D_head)
        V = self.W_V(X)  # (BS, N, h * D_head)
```

```
        Q = tf.reshape(Q, (-1, N, h, D_head))  # (BS, N, h, D_head)
        K = tf.reshape(K, (-1, N, h, D_head))
        V = tf.reshape(V, (-1, N, h, D_head))

        # Transpose for attention computation
        Q = tf.transpose(Q, [0, 2, 1, 3])  # (BS, h, N, D_head)
        K = tf.transpose(K, [0, 2, 1, 3])
        V = tf.transpose(V, [0, 2, 1, 3])
        # scaled dot-product attention
        attn_scores = tf.matmul(Q, K, transpose_b=True)  # (BS, h, N, N)
        attn_scores /= tf.math.sqrt(
            tf.cast(D_head, attn_scores.dtype)
        )  # scale by sqrt(D_head)

        A = tf.expand_dims(A, 1)  # (BS, 1, N, N)
        attn_scores += A  # Broadcast to all heads

        attn_weights = tf.nn.softmax(attn_scores, axis=-1)  # (BS, h, N, N)

        output = tf.matmul(attn_weights, V)  # (BS, h, N, D_head)
        output = tf.transpose(output, [0, 2, 1, 3])  # (BS, N, h, D_head)
        output = tf.reshape(output, (-1, N, h * D_head))  # (BS, N, h * D_head)
        output = self.W_O(output)  # (BS, N, D_model)
        return output


class TransformerBlock(keras.layers.Layer):
    def __init__(self):
        super().__init__(dtype=FLOAT)
        self.attn = MultiHeadAttention()
        self.norm1 = keras.layers.LayerNormalization(dtype=FLOAT)
        self.norm2 = keras.layers.LayerNormalization(dtype=FLOAT)
        self.ffn = keras.Sequential([
            keras.layers.Dense(D_fcn, activation='relu', **commonDense),  #␣
↪Switched to ReLU
            keras.layers.Dense(D_model, **commonDense),
        ])

    def call(self, X, A):
        "R^{N x D_model} -> R^{N x D_model}"

        # NEW : pre norm blocks
        X_norm = self.norm1(X)
        X_attn = self.attn(X_norm, A)
        X = X + X_attn
        X_norm2 = self.norm2(X)
        X_ffn = self.ffn(X_norm2)
```

```
        X = X + X_ffn
        return X


class Decoder(keras.layers.Layer):
    def __init__(self):
        super().__init__(dtype=FLOAT)
        self.proj1 = keras.layers.Dense(D_model, activation='gelu',␣
↪**commonDense)
        self.proj2 = keras.layers.Dense(P*P*C, activation='sigmoid',␣
↪**commonDense)

    def call(self, X):
        BS = tf.shape(X)[0]
        X = self.proj1(X)
        X = self.proj2(X)
        return tf.reshape(X, (BS, N, P, P, C))


class ImageInpaintingTransformer(keras.Model):
    def __init__(self):
        super().__init__(dtype=FLOAT)
        self.embed = PatchEmbedding()
        self.transformer_blocks = [TransformerBlock() for _ in␣
↪range(num_layers)]
        self.decoder = Decoder()

    def build(self, input_shape):
        BS = input_shape[0]
        # dummy_images = tf.zeros((BS, H, W, C), dtype=FLOAT)  # THIS WASTED 40␣
↪MINUTES
        self.call(*next(iter(val_ds.take(1))))
        self.built = True

    def call(self, image, obvmask):
        image = tf.multiply(image, tf.cast(obvmask, FLOAT))
        # viz_img(image[0])
        patches = extract_patches(image)
        AttnMask = create_attention_mask(obvmask)
        # viz_img(AttnMask[0])

        BS = tf.shape(patches)[0]
        patches_flat = tf.reshape(patches, [BS, N, P**2 * C])
        # tf.print(tf.shape(patches_flat))
        X = self.embed(patches_flat)
        for block in self.transformer_blocks:
            X = block(X, AttnMask)
```

```
        reconstructed_patches = self.decoder(X)  # R^{BS x N x P x P x C}
        return patches_to_imgs(reconstructed_patches)


model = ImageInpaintingTransformer()
model.build((BATCH_SIZE, H, W, C))
model.summary()
```

```
Model: "image_inpainting_transformer"
```

Layer (type)	Output Shape	Param #
patch_embedding (PatchEmbedding)	?	623,104
transformer_block (TransformerBlock)	?	1,577,728
transformer_block_1 (TransformerBlock)	?	1,577,728
decoder (Decoder)	?	361,152

Total params: 4,139,712 (15.79 MB)

Trainable params: 4,139,712 (15.79 MB)

Non-trainable params: 0 (0.00 B)

[7]:
```
# model.load_weights("best_run.keras")
```

[8]:
```
session_epochs = 0
```

[20]:
```
@tf.function
def costfunc(y_true: tf.Tensor, y_pred: tf.Tensor, obsvmask: tf.Tensor):
    errors = tf.square(tf.subtract(y_true, y_pred))
    inpaintmask = tf.cast(tf.logical_not(obsvmask), FLOAT)
    masked_errors = tf.multiply(errors, inpaintmask)
    sum_masked_errors = tf.reduce_sum(masked_errors)
    area = tf.reduce_sum(inpaintmask)
    masked_loss = sum_masked_errors / (area + keras.backend.epsilon())

    global_loss = tf.reduce_mean(errors)
```

```python
    return masked_loss + global_loss
    # return masked_loss


# optimizer = keras.optimizers.AdamW(
#     learning_rate=3e-4,
#     weight_decay=0.05,
#     clipvalue=1.0
# )
optimizer = keras.optimizers.Adam(learning_rate=1e-5)

@tf.function
def train_step(image: tf.Tensor, mask: tf.Tensor):
    with tf.GradientTape() as tape:
        reconstructed_img = model(image, mask)  # N x P x P x C
        loss = costfunc(image, reconstructed_img, mask)
        tf.debugging.check_numerics(loss, "Loss contains NaN or Inf.")
        # if (tf.math.is_nan(loss)):
        # raise Exception("Divergence")
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

@tf.function
def val_step(image: tf.Tensor, mask: tf.Tensor):
    reconstructed_img = model(image, mask, training=False)
    loss = costfunc(image, reconstructed_img, mask)
    return loss


epochs = 50

print("Starting training")

best_val_loss = float("inf")
best_epoch = -1
for _ in range(epochs):
    epoch_loss = 0.0
    steps = 0
    pbar = tqdm(
        train_ds,
        desc=f"Epoch {session_epochs+1}",
        unit="batch",
        total=train_batches,
    )
    for image_batch, mask_batch in pbar:
        loss = train_step(image_batch, mask_batch)
```

```
        epoch_loss += loss
        steps += 1
        # Dynamically update the tqdm bar without spamming stdout
        pbar.set_postfix(loss=f"{loss:.4f}")
    train_loss = epoch_loss / steps

    val_loss_total = 0.0
    val_steps = 0
    pbar_val = tqdm(
        val_ds,
        desc=f"Epoch {session_epochs+1} Validation",
        unit="batch",
        total=val_batches,
    )
    for val_image_batch, val_mask_batch in pbar_val:
        loss = val_step(val_image_batch, val_mask_batch)
        val_loss_total += loss
        val_steps += 1
        pbar_val.set_postfix(loss=f"{loss:.4f}")

    avg_val_loss = val_loss_total / val_steps
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        best_epoch = session_epochs + 1
        model.save("best_run.keras")
    print(
        f"Epoch {session_epochs+1} Summary: Train Loss = {train_loss:.4f} |␣
↪Validation Loss = {avg_val_loss:.4f}"
    )
    session_epochs += 1
```

```
Starting training

Epoch 111: 100%|      | 50/50 [00:17<00:00,  2.84batch/s, loss=0.0732]
Epoch 111 Validation: 100%|      | 7/7 [00:02<00:00,  2.81batch/s,
loss=0.1958]

Epoch 111 Summary: Train Loss = 0.0791 | Validation Loss = 0.2417

Epoch 112: 100%|      | 50/50 [00:12<00:00,  4.13batch/s, loss=0.0716]
Epoch 112 Validation: 100%|      | 7/7 [00:01<00:00,  4.12batch/s,
loss=0.1953]

Epoch 112 Summary: Train Loss = 0.0748 | Validation Loss = 0.2410

Epoch 113: 100%|      | 50/50 [00:12<00:00,  4.15batch/s, loss=0.0706]
Epoch 113 Validation: 100%|      | 7/7 [00:01<00:00,  3.70batch/s,
loss=0.1956]

Epoch 113 Summary: Train Loss = 0.0729 | Validation Loss = 0.2408
```

```
Epoch 114: 100%|     | 50/50 [00:12<00:00,  4.09batch/s, loss=0.0699]
Epoch 114 Validation: 100%|     | 7/7 [00:01<00:00,  4.16batch/s,
loss=0.1959]

Epoch 114 Summary: Train Loss = 0.0716 | Validation Loss = 0.2407

Epoch 115: 100%|     | 50/50 [00:12<00:00,  4.14batch/s, loss=0.0693]
Epoch 115 Validation: 100%|     | 7/7 [00:01<00:00,  4.22batch/s,
loss=0.1962]

Epoch 115 Summary: Train Loss = 0.0706 | Validation Loss = 0.2408

Epoch 116: 100%|     | 50/50 [00:12<00:00,  4.15batch/s, loss=0.0689]
Epoch 116 Validation: 100%|     | 7/7 [00:01<00:00,  4.22batch/s,
loss=0.1965]

Epoch 116 Summary: Train Loss = 0.0698 | Validation Loss = 0.2410

Epoch 117: 100%|     | 50/50 [00:12<00:00,  4.08batch/s, loss=0.0685]
Epoch 117 Validation: 100%|     | 7/7 [00:01<00:00,  3.82batch/s,
loss=0.1969]

Epoch 117 Summary: Train Loss = 0.0691 | Validation Loss = 0.2412

Epoch 118: 100%|     | 50/50 [00:12<00:00,  4.08batch/s, loss=0.0681]
Epoch 118 Validation: 100%|     | 7/7 [00:01<00:00,  4.21batch/s,
loss=0.1971]

Epoch 118 Summary: Train Loss = 0.0686 | Validation Loss = 0.2414

Epoch 119: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0678]
Epoch 119 Validation: 100%|     | 7/7 [00:01<00:00,  3.96batch/s,
loss=0.1973]

Epoch 119 Summary: Train Loss = 0.0681 | Validation Loss = 0.2416

Epoch 120: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0675]
Epoch 120 Validation: 100%|     | 7/7 [00:01<00:00,  3.92batch/s,
loss=0.1975]

Epoch 120 Summary: Train Loss = 0.0676 | Validation Loss = 0.2418

Epoch 121: 100%|     | 50/50 [00:12<00:00,  4.08batch/s, loss=0.0672]
Epoch 121 Validation: 100%|     | 7/7 [00:01<00:00,  4.19batch/s,
loss=0.1976]

Epoch 121 Summary: Train Loss = 0.0673 | Validation Loss = 0.2419

Epoch 122: 100%|     | 50/50 [00:12<00:00,  4.09batch/s, loss=0.0669]
Epoch 122 Validation: 100%|     | 7/7 [00:01<00:00,  4.05batch/s,
loss=0.1977]

Epoch 122 Summary: Train Loss = 0.0669 | Validation Loss = 0.2421

Epoch 123: 100%|     | 50/50 [00:12<00:00,  4.09batch/s, loss=0.0667]
Epoch 123 Validation: 100%|     | 7/7 [00:01<00:00,  3.99batch/s,
loss=0.1977]
```

```
Epoch 123 Summary: Train Loss = 0.0666 | Validation Loss = 0.2422

Epoch 124: 100%|     | 50/50 [00:12<00:00,  4.13batch/s, loss=0.0664]
Epoch 124 Validation: 100%|     | 7/7 [00:01<00:00,  3.90batch/s,
loss=0.1978]

Epoch 124 Summary: Train Loss = 0.0663 | Validation Loss = 0.2424

Epoch 125: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0662]
Epoch 125 Validation: 100%|     | 7/7 [00:01<00:00,  3.99batch/s,
loss=0.1978]

Epoch 125 Summary: Train Loss = 0.0660 | Validation Loss = 0.2425

Epoch 126: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0660]
Epoch 126 Validation: 100%|     | 7/7 [00:01<00:00,  4.04batch/s,
loss=0.1978]

Epoch 126 Summary: Train Loss = 0.0657 | Validation Loss = 0.2427

Epoch 127: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0658]
Epoch 127 Validation: 100%|     | 7/7 [00:01<00:00,  4.08batch/s,
loss=0.1978]

Epoch 127 Summary: Train Loss = 0.0655 | Validation Loss = 0.2428

Epoch 128: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0656]
Epoch 128 Validation: 100%|     | 7/7 [00:01<00:00,  4.07batch/s,
loss=0.1978]

Epoch 128 Summary: Train Loss = 0.0653 | Validation Loss = 0.2430

Epoch 129: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0654]
Epoch 129 Validation: 100%|     | 7/7 [00:01<00:00,  3.91batch/s,
loss=0.1978]

Epoch 129 Summary: Train Loss = 0.0650 | Validation Loss = 0.2431

Epoch 130: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0653]
Epoch 130 Validation: 100%|     | 7/7 [00:01<00:00,  4.05batch/s,
loss=0.1978]

Epoch 130 Summary: Train Loss = 0.0648 | Validation Loss = 0.2432

Epoch 131: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0651]
Epoch 131 Validation: 100%|     | 7/7 [00:01<00:00,  3.91batch/s,
loss=0.1978]

Epoch 131 Summary: Train Loss = 0.0646 | Validation Loss = 0.2434

Epoch 132: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0649]
Epoch 132 Validation: 100%|     | 7/7 [00:01<00:00,  4.09batch/s,
loss=0.1978]

Epoch 132 Summary: Train Loss = 0.0644 | Validation Loss = 0.2435
```

```
Epoch 133: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0648]
Epoch 133 Validation: 100%|     | 7/7 [00:01<00:00,  3.82batch/s,
loss=0.1978]

Epoch 133 Summary: Train Loss = 0.0643 | Validation Loss = 0.2436

Epoch 134: 100%|     | 50/50 [00:12<00:00,  4.09batch/s, loss=0.0646]
Epoch 134 Validation: 100%|     | 7/7 [00:01<00:00,  3.88batch/s,
loss=0.1978]

Epoch 134 Summary: Train Loss = 0.0641 | Validation Loss = 0.2438

Epoch 135: 100%|     | 50/50 [00:12<00:00,  4.10batch/s, loss=0.0645]
Epoch 135 Validation: 100%|     | 7/7 [00:01<00:00,  3.86batch/s,
loss=0.1978]

Epoch 135 Summary: Train Loss = 0.0639 | Validation Loss = 0.2439

Epoch 136: 100%|     | 50/50 [00:12<00:00,  4.09batch/s, loss=0.0643]
Epoch 136 Validation: 100%|     | 7/7 [00:01<00:00,  4.01batch/s,
loss=0.1978]

Epoch 136 Summary: Train Loss = 0.0637 | Validation Loss = 0.2440

Epoch 137: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0642]
Epoch 137 Validation: 100%|     | 7/7 [00:01<00:00,  4.07batch/s,
loss=0.1979]

Epoch 137 Summary: Train Loss = 0.0636 | Validation Loss = 0.2442

Epoch 138: 100%|     | 50/50 [00:12<00:00,  4.10batch/s, loss=0.0640]
Epoch 138 Validation: 100%|     | 7/7 [00:02<00:00,  3.27batch/s,
loss=0.1979]

Epoch 138 Summary: Train Loss = 0.0634 | Validation Loss = 0.2443

Epoch 139: 100%|     | 50/50 [00:12<00:00,  4.10batch/s, loss=0.0639]
Epoch 139 Validation: 100%|     | 7/7 [00:01<00:00,  3.83batch/s,
loss=0.1979]

Epoch 139 Summary: Train Loss = 0.0632 | Validation Loss = 0.2445

Epoch 140: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0637]
Epoch 140 Validation: 100%|     | 7/7 [00:01<00:00,  4.07batch/s,
loss=0.1979]

Epoch 140 Summary: Train Loss = 0.0631 | Validation Loss = 0.2446

Epoch 141: 100%|     | 50/50 [00:12<00:00,  4.10batch/s, loss=0.0636]
Epoch 141 Validation: 100%|     | 7/7 [00:01<00:00,  4.05batch/s,
loss=0.1979]

Epoch 141 Summary: Train Loss = 0.0629 | Validation Loss = 0.2447

Epoch 142: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0635]
Epoch 142 Validation: 100%|     | 7/7 [00:01<00:00,  3.71batch/s,
loss=0.1979]
```

```
Epoch 142 Summary: Train Loss = 0.0628 | Validation Loss = 0.2449

Epoch 143: 100%|     | 50/50 [00:12<00:00,  4.10batch/s, loss=0.0633]
Epoch 143 Validation: 100%|     | 7/7 [00:01<00:00,  4.06batch/s,
loss=0.1979]

Epoch 143 Summary: Train Loss = 0.0626 | Validation Loss = 0.2450

Epoch 144: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0632]
Epoch 144 Validation: 100%|     | 7/7 [00:01<00:00,  4.09batch/s,
loss=0.1980]

Epoch 144 Summary: Train Loss = 0.0625 | Validation Loss = 0.2452

Epoch 145: 100%|     | 50/50 [00:12<00:00,  4.12batch/s, loss=0.0631]
Epoch 145 Validation: 100%|     | 7/7 [00:01<00:00,  4.10batch/s,
loss=0.1980]

Epoch 145 Summary: Train Loss = 0.0624 | Validation Loss = 0.2453

Epoch 146: 100%|     | 50/50 [00:12<00:00,  4.10batch/s, loss=0.0630]
Epoch 146 Validation: 100%|     | 7/7 [00:01<00:00,  4.13batch/s,
loss=0.1980]

Epoch 146 Summary: Train Loss = 0.0622 | Validation Loss = 0.2454

Epoch 147: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0628]
Epoch 147 Validation: 100%|     | 7/7 [00:01<00:00,  4.00batch/s,
loss=0.1980]

Epoch 147 Summary: Train Loss = 0.0621 | Validation Loss = 0.2456

Epoch 148: 100%|     | 50/50 [00:12<00:00,  4.10batch/s, loss=0.0627]
Epoch 148 Validation: 100%|     | 7/7 [00:01<00:00,  3.83batch/s,
loss=0.1980]

Epoch 148 Summary: Train Loss = 0.0619 | Validation Loss = 0.2457

Epoch 149: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0626]
Epoch 149 Validation: 100%|     | 7/7 [00:01<00:00,  3.99batch/s,
loss=0.1980]

Epoch 149 Summary: Train Loss = 0.0618 | Validation Loss = 0.2459

Epoch 150: 100%|     | 50/50 [00:12<00:00,  4.11batch/s, loss=0.0625]
Epoch 150 Validation: 100%|     | 7/7 [00:01<00:00,  3.84batch/s,
loss=0.1980]

Epoch 150 Summary: Train Loss = 0.0617 | Validation Loss = 0.2460

Epoch 151: 100%|     | 50/50 [00:12<00:00,  4.07batch/s, loss=0.0624]
Epoch 151 Validation: 100%|     | 7/7 [00:01<00:00,  3.77batch/s,
loss=0.1981]

Epoch 151 Summary: Train Loss = 0.0615 | Validation Loss = 0.2462
```

Epoch 152: 100%| | 50/50 [00:12<00:00, 4.11batch/s, loss=0.0622]
Epoch 152 Validation: 100%| | 7/7 [00:01<00:00, 3.87batch/s,
loss=0.1980]

Epoch 152 Summary: Train Loss = 0.0614 | Validation Loss = 0.2463

Epoch 153: 100%| | 50/50 [00:12<00:00, 4.10batch/s, loss=0.0621]
Epoch 153 Validation: 100%| | 7/7 [00:01<00:00, 4.15batch/s,
loss=0.1981]

Epoch 153 Summary: Train Loss = 0.0613 | Validation Loss = 0.2464

Epoch 154: 100%| | 50/50 [00:12<00:00, 4.10batch/s, loss=0.0620]
Epoch 154 Validation: 100%| | 7/7 [00:01<00:00, 3.82batch/s,
loss=0.1981]

Epoch 154 Summary: Train Loss = 0.0611 | Validation Loss = 0.2466

Epoch 155: 100%| | 50/50 [00:12<00:00, 4.12batch/s, loss=0.0619]
Epoch 155 Validation: 100%| | 7/7 [00:01<00:00, 4.04batch/s,
loss=0.1982]

Epoch 155 Summary: Train Loss = 0.0610 | Validation Loss = 0.2467

Epoch 156: 100%| | 50/50 [00:12<00:00, 4.10batch/s, loss=0.0618]
Epoch 156 Validation: 100%| | 7/7 [00:01<00:00, 4.04batch/s,
loss=0.1982]

Epoch 156 Summary: Train Loss = 0.0609 | Validation Loss = 0.2469

Epoch 157: 100%| | 50/50 [00:12<00:00, 4.04batch/s, loss=0.0616]
Epoch 157 Validation: 100%| | 7/7 [00:02<00:00, 3.20batch/s,
loss=0.1983]

Epoch 157 Summary: Train Loss = 0.0608 | Validation Loss = 0.2471

Epoch 158: 100%| | 50/50 [00:12<00:00, 4.06batch/s, loss=0.0615]
Epoch 158 Validation: 100%| | 7/7 [00:02<00:00, 3.41batch/s,
loss=0.1983]

Epoch 158 Summary: Train Loss = 0.0606 | Validation Loss = 0.2472

Epoch 159: 100%| | 50/50 [00:12<00:00, 4.08batch/s, loss=0.0614]
Epoch 159 Validation: 100%| | 7/7 [00:01<00:00, 4.07batch/s,
loss=0.1984]

Epoch 159 Summary: Train Loss = 0.0605 | Validation Loss = 0.2473

Epoch 160: 100%| | 50/50 [00:12<00:00, 4.08batch/s, loss=0.0613]
Epoch 160 Validation: 100%| | 7/7 [00:01<00:00, 3.55batch/s,
loss=0.1984]

Epoch 160 Summary: Train Loss = 0.0604 | Validation Loss = 0.2475

[16]:
```python
model.save("brr1.keras")
# model.load_weights("best_run.keras")
```

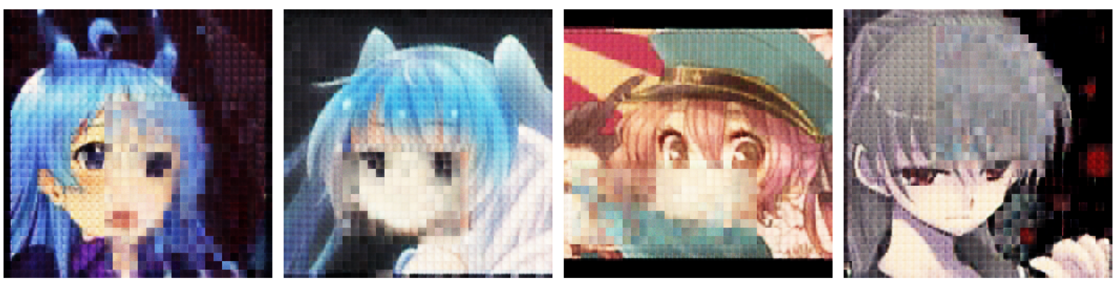
[11]:
```python
def apply_obsv_mask(image: tf.Tensor, obvmask: tf.Tensor) -> tf.Tensor:
    return tf.multiply(image, tf.cast(obvmask, FLOAT))


def viz_grid(batch: tf.Tensor, max: int = 4):
    batch_size: int = batch.shape[0]  # type: ignore
    num = min(batch_size, max)
    fig, axes = plt.subplots(nrows=1, ncols=num, figsize=(15, 15), dpi=300)
    if num == 1:
        axes = [axes]
    for i in range(num):
        # Original image
        axes[i].imshow(
            tf.clip_by_value(
                tf.cast(batch[i], dtype=tf.float32), 0, 1  # type: ignore
            ).numpy() # type: ignore
        )
        axes[i].axis("off")
    plt.tight_layout()
    plt.show()


def reconstruct(original: tf.Tensor, reconstruct: tf.Tensor, obvmask: tf.
↪Tensor):
    return tf.add(
        tf.multiply(tf.cast(obvmask, FLOAT), original),
        tf.multiply(tf.cast(tf.logical_not(obvmask), FLOAT), reconstruct),
    )
```

[19]:
```python
img, obvmask = next(iter(train_ds.take(1)))
viz_grid(img)
viz_grid(apply_obsv_mask(img, obvmask))
model_out = model(img, obvmask)
# reconstructed = reconstruct(img, model(img, obvmask), obvmask)
viz_grid(model_out)

# viz_img(model_out[0])
# viz_img(img[0])
```



[13]:

```
# Qualitative Eval
# visualize_unbatched_dataset(test_ds, 5)



# img = tf.image.decode_image(
#     tf.io.read_file("/home/navid/Dev/PaperTex/impl/naruto")
#     , dtype=tf.float32)
# img = tf.image.resize_with_crop_or_pad(img, H, W)
# img = tf.expand_dims(img, 0)
# tf.print(tf.shape(img))
# obvmask = tf.expand_dims(random_visibility_mask(),0)
# tf.print(tf.shape(obvmask))
```