Глубокое обучение

Тема 2. Генеративно-состязательные сети (продолжение)

Обычные (ванильные) сети GAN

Пусть $p_g(\mathbf{x})$ - это распределение наших данных для $\mathbf{x} \in X_{\text{train}}$ (например, встроенных изображений). Цель генеративного моделирования - изучить это распределение данных с помощью вероятностной модели. К сожалению, точное моделирование этой функции плотности вероятности вычислительно сложно для всех моделей, кроме самых тривиальных. Генеративно-состязательные сети (GAN) вместо этого моделируют распределение данных как преобразование фиксированного распределения случайных переменных $p_z(\mathbf{z})$ для $\mathbf{z} \in Z_{\text{train}}$. Преобразование, называемое **генератором**, выражается как детерминированная сеть прямого распределения $G : Z_{\text{train}} \to X_{\text{train}}$, где

$$p_G(\mathbf{x} \mid \mathbf{z}) = \delta(\mathbf{x} - G(\mathbf{z})), \; p_G(\mathbf{x}) = \mathbb{E}_{\mathbf{z} \sim p_z}[p_G(\mathbf{x} \mid \mathbf{z})].$$

Цель состоит в том, чтобы получить генератор G так, чтобы $p_G(\mathbf{x}) \approx p_g(\mathbf{x})$.

Фреймворк GAN обучает генератор G так, чтобы нижная дискриминатива модель (**дискриминатор**) $D : X_{\text{train}} \to [0, 1]$ не могла отличить образцы из распределения данных от образцов из генератора с распределением. Как генератор, так и дискриминатор обучаются с использованием состязательной (минимаксной) цели по отношению к функции потерь $V(D, G)$, где

$$V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_g}[\log D(\mathbf{x})] + \mathbb{E}_{\substack{\mathbf{z} \sim p_z \\ \mathbf{x} \sim p_G(\mathbf{x} \mid \mathbf{z})}}[\log(1 - D(\mathbf{x}))]$$

Гудфеллоу и др. (2014) показали, что для идеального распределения целевая функция $C(G) := \max_D V(D, G)$ эквивалентна дивергенции Дженсена-Шеннона между двумя распределениями p_g и p_G.

Дивергенция Дженсена-Шеннона (JS) определяется для любых двух абсолютно непрерывных распределений P и Q через дивергенцию Кульбака-Лейблера (KL) как:

$$D_{JS}(P \mid Q) = \frac{1}{2}(D_{KL}(P \mid M) + D_{KL}(Q \mid M)), \; M = \frac{1}{2}(P + Q),$$

$$D_{KL}(P \mid Q) = \int_{\mathbf{x}} p(\mathbf{x}) \log\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right) d\mathbf{x},$$

где p и q - плотности, связанные соответственно с распределениями P и Q. Можно показать, что $D_{JS}(P \mid Q) \geq 0$ с равенством нулю тогда и только тогда, когда $P = Q$.

Состязательная целевая функция $V(D, G)$ напрямую не подвергается эффективной оптимизации, поскольку каждый шаг генератора G требует обучения полного дискриминатора D до полного. Тем, идеальный дискриминатор больше не предоставляет генератору точной информации о градиенте, поскольку градиент любого глобального или локального максимума $V(D, G)$ равен нулю. Тем не менее, чтобы обеспечить более плавные градиенты, создатели GAN начинают с изменяют цель обновлений генератора и дискриминатора, сохраняя при этом те же характеристики дискриминатора. Они также предлагают оптимизировать $V(D, G)$ с помощью чередующегося переключения оптимизации между обновлениями генератора и дискриминатора. Хотя сходимость этой оптимизации не гарантируется, эмпирически она работает хорошо, если дискриминатор и генератор хорошо сбалансированы.

Обучение GAN

Хотя сети GAN рассматриваются как крутящий момент в области генеративного моделирования, их очень сложно обучать. В этом разделе мы рассмотрим некоторые из наиболее распространенных проблем и проблемы, возникающие при обучении GAN, а также потенциальные решения. В следующем разделе мы рассмотрим пару наиболее фундаментальных изменений в структуре GAN, которые мы можем ввести для решения многих из этих проблем.

Дискриминатор подавляет генератор

Если дискриминатор становится слишком сильным, то сигнал (градиент) функции потерь становится слишком слабым, чтобы обеспечить какие-либо значимые улучшения в генераторе. В худшем случае дискриминатор прекрасно учится отличать реальные изображения от подделок, и градиенты потоковых исчезают, что приводит к отсутствию какого-либо обучения.

Если функция потерь дискриминатора близка к нулю, то нужно найти способы остановить дискриминатор, например, за счет следующих мер:

- Увеличить параметр `rate` слоев Dropout в дискриминаторе, чтобы уменьшить объем информации, проходящей через сеть дискриминатора
- Уменьшить скорость обучения дискриминатора
- Уменьшить количество сверточных фильтров в дискриминаторе
- Добавить шум к меткам при обучении дискриминатора
- При обучении дискриминатора переворачивать метки некоторых изображений случайным образом

Генератор подавляет дискриминатор

Если дискриминатор недостаточно мощный, то генератор найдет способы легко обмануть дискриминатор с помощью небольшой выборки почти идеальных изображений. Это приводит как к потоковым мод, распределение (mode collapse): генератор коллапсирует, то есть выдает ограниченное разнообразие образцов.

Например, предположим, что нам нужно обучить генератор на нескольких пакетах без обновления дискриминатора между ними. Генератор будет склонен находить единственное наблюдение (также называемое модой), которое всегда обманывает дискриминатор и начинает отображать каждую точку скрытого входного пространства на это изображение. Более того, градиенты функции потерь склонятся почти до нуля, поэтому она не сможет выйти из этого состояния. Даже если бы мы затем попытались переобучить дискриминатор, чтобы его не обманывала эта одна точка, то генератор просто нашел бы другую моду, которая обманывает дискриминатор, поскольку он уже не восприимчивый скорый сигнал и, следовательно, не имеет стимула дифференцировать свой выходной сигнал.

Если генератор страдает от коллапса мод, то можно попробовать усилить дискриминатор, используя методы, перечисленные тем, которые были перечислены ранее. Кроме того, можно попробовать снизить скорость обучения обоих сетей и увеличить размер пакета.

Неинформативность показателя потерь

Поскольку модель глубокого обучения оптимизирована на минимизацию функции потерь, было бы естественным считать, что чем меньше функция потерь генератора, тем лучше качество полученных изображений. Однако, поскольку генератор оценивается только по способности обманывать дискриминатор, поэтому совершенно естественно, нельзя сравнивать функции потерь, оцениваемую на разных этапах процесса обучения. Может быть так, что функция потерь генератора фактически увеличивается со временем, при этом качество изображений явно улучшается. Отсутствие корреляции между потерями генератора и качеством изображения иногда затрудняет монитторинг обучения GAN.

Гиперпараметры GAN

Даже в случае простых GAN необходимо настраивать большое количество гиперпараметров. Помимо общей архитектуры дискриминатора и генератора, необходимо учитывать параметры, которые управляют нормализацией пакетов, гиперслои (dropout), скоростью обучения, слоями активации, сверточными фильтрами, размером ядра, шагом сверток (striding), размером пакета и размерностью скрытого пространства. Может быть так, что функция потерь генератора фактически увеличивается со временем, при этом качество изображений явно улучшается. Отсутствие корреляции между потерями генератора и качеством изображения иногда затрудняет монитторинг обучения GAN.

Ученик полезно важно понимать внутреннюю работу GAN и знать, как интерпретировать функции потерь, чтобы определить разумные корректировки гиперпараметров, которые могут улучшить стабильность модели.

В последнем году несколько конечных достижений реально улучшили общую стабильность моделей GAN и уменьшили вероятность возникновения некоторых проблем, перечисленных ранее, таких как коллапсные мод.

Нейронная сеть WGAN-GP (Wasserstein GAN with Gradient Penalty)

Сеть WGAN (Wasserstein GAN), представленный в статье Арджовского и др в 2017 году стала одним из первых больших шагов на пути к стабилизации обучения GAN. При помощи несколько изменений метрики сметки показала, как обучить GAN, обладающие следующими двумя свойствами:

- Значимая метрика потерь, которая коррелирует со следовательностью генератора и качеством выборки
- Повышенная стабильность процесса оптимизации.

Функция потерь Вассерштейна применяется как для дискриминатора, так и для генератора. Исполняемая этой функции потерь вместо бинарной кросс-энтропии приводит к более стабильной следовательности GAN. Определение функции потерь Вассерштейна, а затем постоянных, какие еще изменения нужно внести в архитектуру модели GAN и процедуру обучения, чтобы перейти на новую функцию потерь.

Функция потерь Вассерштейна

Вспомним определение функции потерь бинарной кросс-энтропии — функции, которую мы сейчас используем для обучения дискриминатора и генератора GAN.

Вообще говоря, кросс-энтропия распределения q относительно распределения p определяется по формуле

$$H(p, q) = -\mathbb{E}_p[\log q],$$

где $\mathbb{E}_p$ - оператор математического ожидания относительно распределения p. Для дискретных распределений p и q с поддержкой X имеем

$$H(p, q) = -\sum_{x \in X} p(x) \log q(x),$$

для абсолютно непрерывных распределений

$$H(p, q) = -\int_X p(x) \log q(x)\, dx.$$

Кросс-энтропия используется для построения функции потерь в машинном обучении. Например, в задаче бинарной классификации (с метками классов 0 и 1) для $p \in \{[1, -]], [-, 1]\}$ и $q \in \{[\hat{y}, 1 - \hat{y}]\}$ мера неполноценности p к q равна

$$H(p, q) = -\sum_{i=1}^{2} p_i \log q_i = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Для набора данных из n точек с истинными метками классов y_i и прогнозами $\hat{y}_i$ функция потерь на основе бинарной кросс-энтропии равна

$$-\frac{1}{n} \sum_{i=1}^{n} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

Чтобы обучить дискриминатор D, мы вычисляем потери (кросс-энтропию) при сравнении прогнозов для реальных изображений $p_i = D(x_i)$ с отклонами $y_i = 1$ и прогнозов для сгенерированных изображений $p_i = D(G(z_i))$ с отклонами $y_i = -1$. Поэтому для критика WGAN задачу минимизацию функции потерь можно записать следующим образом

$$\min_D -(\mathbb{E}_{x \sim p_r}[D(x)] - \mathbb{E}_{z \sim p_z}[D(G(z))])$$

Чтобы обучить генератор G, мы вычисляем потери (кросс-энтропию) при сравнении прогнозов для сгенерированных изображений $p_i = D(G(z_i))$ с отклонами $y_i = 1$. Следовательно, для генератора GAN задачу минимизации функции потерь можно записать, как показано ниже

$$\min_G -(\mathbb{E}_{z \sim p_z}[\log(D(G(z)))])$$

Сравним это с функцией потерь Вассерштейна. Во-первых, функция потерь Вассерштейна требует, чтобы мы использовали в качестве метки -1 для $-$, а не 1-й. Также удаляем логарифмическую активацию из последнего слоя дискриминатора, чтобы прогнозы были не бинарными, а больше не ограничивались диапазоном $[0, 1]$ из вместо этого могли быть любым числом в диапазоне $(-\infty, +\infty)$. По этой причине дискриминатор в WGAN обычно называют **критиком** (critic), который выдает **оценку** (score), а не вероятность.

Функция потерь Вассерштейна для набора данных с истинными метками классов y_i и прогнозами p_i определяется следующим образом:

$$-\frac{1}{n} \sum_{i=1}^{n} y_i p_i$$

Чтобы обучить критика WGAN пытается максимизировать разницу между своими прогнозами для реальных изображений и сгенерированных изображений.

Для обучения генератора WGAN мы вычисляем потери при сравнении прогнозов для сгенерированных изображений $p_i = D(G(z_i))$ с отклонами $y_i = 1$. Следовательно, для генератора WGAN задачу минимизации функции потерь можно записать следующим образом:

$$\min_G -(\mathbb{E}_{z \sim p_z}[D(G(z))])$$

Другими словами, генератор WGAN пытается создать изображения, которые получат как можно более высокую оценку критика (т. е. критика обманывает, заставив думать, что они реальны).

Условие (ограничение) Липшица

Может давать, что теперь мы позволяем критику выводить любое число в диапазоне $(-\infty, +\infty)$, вместо того, чтобы применять сигмоидную функцию для ограничения вывода обычным диапазоном $[0, 1]$. Поэтому потери Вассерштейна могут быть очень большими, что может травмировать - обычно создает шаги большого размера слишком сильно. Фактически, для обучения WGAN нам нужно, что для того, чтобы функция потерь Вассерштейна работала, нам также необходимо наложить дополнительное ограничение на веса. В частности, требуется, чтобы критик был 1-липшицевой непрерывной функцией. Давайте разберем это по отдельности, чтобы понять, что это значит более подробно.

Критик — это функция D, которая преобразует изображения в предсказания. Говорят, что эта функция является 1-липшицевой, если она удовлетворяет следующему неравенству для любых двух входных изображений x_1 и x_2:

$$\frac{|D(x_1) - D(x_2)|}{|x_1 - x_2|} \leq 1.$$

Здесь $|x_1 - x_2|$ - это сумма попиксельная абсолютная разница между двумя изображениями, а $|D(x_1) - D(x_2)|$ - это абсолютная разница между предсказаниями критика. По сути, чем требуется ограничение скорости, с которой предсказания критика могут меняться между двумя изображениями (т. е. абсолютное значение градиента должно быть не более 1 везде).

Соблюдение ограничения Липшица

В исходной статье WGAN авторы использовали, как можно обеспечить соблюдение ограничения Липшица, ограничивая веса критика так, чтобы они находились в небольшом диапазоне $[-0.01, 0.01]$ после каждого обучающего пакета. Одно из ограничений заключается по поводу этого подхода заключается в том, что способность критика к обучению значительно снижается, поскольку мы сужаем его вес. Фактически, даже в оригинальной статье WGAN авторы пишут: «Ограничение веса - это явно грязный способ обеспечить соблюдение ограничения Липшица». Сильный критик имеет огромное значение для успеха WGAN, поскольку без точных градиентов генератор не может научиться адаптировать свои веса для получения более качественных выборок. Поэтому другие исследователи искали альтернативные способы обеспечить соблюдение ограничения Липшица и улучшить способность WGAN изучать сложные функции. Одним из таких методов является WGAN с градиентным штрафом, в котором ограничение Липшица может быть реализовано напрямую путем включения штрафного члена за градиент в функцию потерь для критика, который наказывает модель, если норма градиента отклоняется от единицы. Это приводит к гораздо более стабильному тренировочному процессу. Далее мы увидим, как встроить этот дополнительный член в функцию потерь для нашего критика.

Потери в форме градиентного штрафа

На рисунке ниже представлена диаграмма процесса обучения критика WGAN-GP. Если мы сравним это с предыдущим процессом обучения дискриминатора, то мы увидим что изменения дополнениями является потери в форме градиентного штрафа, включенные как часть общей функции потерь, наряду с потерями Вассерштейна для реальных и сгенерированных изображений.

Метод для вычисления градиентного штрафа выглядит так:

```
def gradient_penalty(self, batch_size, real_images, fake_images):
    alpha = tf.random.normal([batch_size, 1, 1, 1], 0.0, 1.0)  # 1.
    diff = fake_images - real_images
    interpolated = real_images + alpha * diff                  # 2.

    with tf.GradientTape() as gp_tape:
        gp_tape.watch(interpolated)
        pred = self.critic(interpolated, training=True)        # 3.

    grads = gp_tape.gradient(pred, [interpolated])[0]          # 4.
    norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))  # 5.
    gp = tf.reduce_mean(norm - 1.0) ** 2                       # 6.
    return gp
```

Здесь:

1. Генерируем случайное число для каждого изображения в пакете.
2. Вычисляем линейную интерполяцию между реальными изображениями и нашими сгенерированными изображениями.
3. Вычисляем прогнозы критика для интерполированных изображений.
4. Вычисляем градиент прогнозов критика по отношению к интерполированным изображениям.
5. Вычисляем норму этого градиента.
6. Вычисляем штраф градиента.

Вычислить этот градиент вводе в процессе обучения сложно, поэтому вместо этого WGAN-GP оценивает градиент всего по нескольким точкам. Чтобы обеспечить сбалансированный набор точек, мы используем набор интерполированных изображений, которые лежат в случайно выбранных точках вдоль линий, попарно соединяющих пары реальных изображений с группой подделанных изображений.

1. каждое изображение в пакете получает случайный угол поворота, кратный 90°
2. далее вычисляется набор интерполированных изображений `interpolated`
3. затем оценивает каждое из интерполированных изображений (переменная `pred`)
4. вычисляется градиентно оценок `grad` по входам (интерполированным) изображениям
5. вычисляется норма L2 векторов градиента
6. метод возвращает среднее квадратическое расстояние между нормами L2 и значением 1

Обучение WGAN-GP

Ключевая принципиальная особенность потерь критика потерь Вассерштайна заключается в том, что нам больше не нужно беспокоиться о балансировании обучения критика и генератора — фактически, при использовании функции потерь Вассерштайна критик должен быть обучен до сходимости перед обновлением генератора, чтобы обеспечить, что градиенты для обновления генератора точны. В этом отличие от стандартного GAN, где важно не позволить дискриминатору стать слишком сильным.

Таким образом, с помощью GAN Вассерштайна мы можем просто обучать критика несколько раз между обновлениями генератора, чтобы обеспечить его близость к сходимости. Обычно используемое соотношение составляет от трех до пяти обновлений критика на одно обновление генератора.

Итак, ключевая концепция, лежащая в основе WGAN-GP - это потери Вассерштайна и штрафной член за градиент, который включен в функцию потерь критика. Этот штраф обучения модели WGAN, включающий в себя все эти идеи, реализован в функции `train_step`:

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
    for i in range(self.critic_steps):                                  # 1.
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )
        with tf.GradientTape() as tape:
            fake_images = self.generator(
                random_latent_vectors, training=True
            )
            fake_predictions = self.critic(fake_images,
                                           training=True)
            real_predictions = self.critic(real_images,
                                           training=True)
            c_wass_loss = tf.reduce_mean(
                fake_predictions) - tf.reduce_mean(real_predictions
            )                                                           # 2.
            c_gp = self.gradient_penalty(
                batch_size, real_images, fake_images
            )                                                           # 3.
            c_loss = c_wass_loss + c_gp * self.gp_weight                # 4.
        c_gradient = tape.gradient(c_loss,
                                   self.critic.trainable_variables)
        self.c_optimizer.apply_gradients(
            zip(c_gradient, self.critic.trainable_variables)
        )                                                               # 5.
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )
        with tf.GradientTape() as tape:
            fake_images = self.generator(random_latent_vectors,
                                         training=True)
            fake_predictions = self.critic(fake_images,
                                           training=True)
            g_loss = -tf.reduce_mean(fake_predictions)                  # 6.
        gen_gradient = tape.gradient(g_loss,
                                     self.generator.trainable_variables)
        self.g_optimizer.apply_gradients(
            zip(gen_gradient, self.generator.trainable_variables)
        )                                                               # 7.
        self.c_loss_metric.update_state(c_loss)
        self.c_wass_loss_metric.update_state(c_wass_loss)
        self.c_gp_metric.update_state(c_gp)
        self.g_loss_metric.update_state(g_loss)

        return {m.name: m.result() for m in self.metrics}
```

В методе `train_step`:

1. выполняется несколько обновлений критика
2. рассчитывается потеря Вассерштайна для критика — разница между средними прогнозом для фальшивых изображений и реальных изображений
3. вычисляется градиентный штраф
4. функция потерь критика представляет собой взвешенную сумму потерь Вассерштайна и градиентного штрафа
5. обновляются вес критика
6. рассчитывается потеря Вассерштайна для генератора
7. обновляются веса генератора

Пакетная нормализация в WGAN-GP

Последнее соображение, на которое мы должны обратить внимание перед обучением WGAN-GP, заключается в том, что в критике не следует использовать пакетную нормализацию. Это связано с тем, что пакетная нормализация создает корреляцию между изображениями в одном пакете, что делает потери выше градиентного штрафа менее эффективным. Экспериментально, что WGAN-GP по-прежнему могут давать отличные результаты даже без пакетной нормализации в критике.

Итак, ключевыми различиями между стандартным GAN и WGAN-GP состоят в следующем:

- WGAN-GP использует потери Вассерштайна.
- WGAN-GP обучается с использованием метки 1 для настоящего изображения и -1 для поддельных изображения.
- В последнем слое критика нет активации сигмоидой.
- В функции потерь для критика включен градиентный штраф.
- Критик обучается несколько раз для каждого обновления генератора.
- В критике нет слоев пакетной нормализации.

Итак, теперь мы можем начать кодировать наши модели, начиная с критика. Вот настоящий импорт:

```
import os
import numpy as np
import tensorflow as tf
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
%tensorflow_version 2.x
```

```
TensorFlow 2.x selected.
```

```
import os
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

```
2.18.0
```

WGAN для набора CelebA Faces

Попробуем сеть WGAN-GP для генерации лиц из набора данных CelebA.

```
from tensorflow.keras import (
    layers,
    models,
    callbacks,
    utils,
    metrics,
    optimizers,
)

from utils import display, sample_batch
```

Параметры

- `IMAGE_SIZE` - количество пикселей по ширине и высоте изображения
- `CHANNELS` - количество каналов изображения (слоев с информацией о цвете)
- `BATCH_SIZE` - размер пакета с изображениями при загрузке и последующем использовании
- `Z_DIM` - размерность латентного пространства
- `LEARNING_RATE` - скорость обучения нейронной сети
- `EPOCHS` - количество эпох
- `CRITIC_STEPS` - количество шагов обучения критика на один шаг обучения генератора
- `GP_WEIGHT` - весовой коэффициент градиентного штрафа
- `LOAD_MODEL` - параметр загрузки модели
- `ADAM_BETA_1`, `ADAM_BETA_2` - параметры оптимизатора Adam

```
IMAGE_SIZE = 64
CHANNELS = 3
BATCH_SIZE = 512
Z_DIM = 128
LEARNING_RATE = 0.0002
EPOCHS = 100
CRITIC_STEPS = 3
GP_WEIGHT = 10.0
LOAD_MODEL = False
ADAM_BETA_1 = 0.5
ADAM_BETA_2 = 0.9
```

Подготовка данных

```
# Load the data
train_data = utils.image_dataset_from_directory(
    "../2 vae/celeba-dataset/img_align_celeba/img_align_celeba",
    labels=None,
    color_mode="rgb",
    image_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE,
    shuffle=True,
    seed=42,
    interpolation="bilinear",
)
```

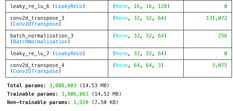
```
Found 202599 files.
```

```
WARNING: All messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1759510119.551650 6738123 gpu_device_factory.cc:305] Could not identif
ully NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been bui
lt with NUMA support.
I0000 00:00:1759510119.551650 6738123 gpu_device_factory.cc:271] Created TensorF
low device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical
PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
```

```
# Preprocess the data
def preprocess(img):
    """
    Normalize and reshape the image
    """
    img = (tf.cast(img, "float32") - 127.5) / 127.5
    return img

train = train_data.map(lambda x: preprocess(x))
```

```
# Show some faces from the training set
train_sample = sample_batch(train)
```

```
display(train_sample, cmap=None)
```



Построение WGAN-GP

```
critic_input = layers.Input(shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS))
x = layers.Conv2D(64, kernel_size=4, strides=2, padding="same")(critic_input)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2D(128, kernel_size=4, strides=2, padding="same")(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(256, kernel_size=4, strides=2, padding="same")(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(512, kernel_size=4, strides=2, padding="same")(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(1, kernel_size=4, strides=1, padding="valid")(x)
critic_output = layers.Flatten()(x)

critic = models.Model(critic_input, critic_output)
critic.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 3)	0
conv2d (Conv2D)	(None, 32, 32, 64)	3,136
leaky_re_lu (LeakyReLU)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 16, 16, 128)	131,200
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 128)	0
dropout (Dropout)	(None, 16, 16, 128)	0
conv2d_2 (Conv2D)	(None, 8, 8, 256)	524,544
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 256)	0
dropout_1 (Dropout)	(None, 8, 8, 256)	0
conv2d_3 (Conv2D)	(None, 4, 4, 512)	2,097,664
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 512)	0
dropout_2 (Dropout)	(None, 4, 4, 512)	0
conv2d_4 (Conv2D)	(None, 1, 1, 1)	8,193
flatten (Flatten)	(None, 1)	0

Total params: 2,764,737 (10.55 MB)
Trainable params: 2,764,737 (10.55 MB)
Non-trainable params: 0 (0.00 B)

```
generator_input = layers.Input(shape=(Z_DIM,))
x = layers.Reshape((1, 1, Z_DIM))(generator_input)
x = layers.Conv2DTranspose(
    512, kernel_size=4, strides=1, padding="valid", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    256, kernel_size=4, strides=2, padding="same", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    128, kernel_size=4, strides=2, padding="same", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    64, kernel_size=4, strides=2, padding="same", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    CHANNELS, kernel_size=4, strides=2, padding="same", activation="tanh"
)(x)
generator = models.Model(generator_input, generator_output)
generator.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 128)	0
reshape (Reshape)	(None, 1, 1, 128)	0
conv2d_transpose (Conv2DTranspose)	(None, 4, 4, 512)	1,048,576
batch_normalization (BatchNormalization)	(None, 4, 4, 512)	2,048
leaky_re_lu_4 (LeakyReLU)	(None, 4, 4, 512)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 8, 8, 256)	2,097,152
batch_normalization_1 (BatchNormalization)	(None, 8, 8, 256)	1,024
leaky_re_lu_5 (LeakyReLU)	(None, 8, 8, 256)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 16, 16, 128)	524,288
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 128)	512
leaky_re_lu_6 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 32, 32, 64)	131,072
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 64)	256
leaky_re_lu_7 (LeakyReLU)	(None, 32, 32, 64)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 64, 64, 3)	3,075

Total params: 3,808,003 (14.53 MB)
Trainable params: 3,806,403 (14.52 MB)
Non-trainable params: 1,920 (7.50 KB)

```
class WGANGPModel(models.Model):
    def __init__(self, critic, generator, latent_dim,
                 critic_steps=3, gp_weight=10.0):
        super(WGANGP, self).__init__()
        self.critic = critic
        self.generator = generator
        self.latent_dim = latent_dim
        self.critic_steps = critic_steps
        self.gp_weight = gp_weight

    def compile(self, c_optimizer, g_optimizer):
        super(WGANGP, self).compile()
        self.c_optimizer = c_optimizer
        self.g_optimizer = g_optimizer
        self.c_wass_loss_metric = metrics.Mean(name="c_wass_loss")
        self.c_gp_metric = metrics.Mean(name="c_gp")
        self.c_loss_metric = metrics.Mean(name="c_loss")
        self.g_loss_metric = metrics.Mean(name="g_loss")

    @property
    def metrics(self):
        return [
            self.c_loss_metric,
            self.c_wass_loss_metric,
            self.c_gp_metric,
            self.g_loss_metric,
        ]

    def gradient_penalty(self, batch_size, real_images, fake_images):
        # 1.5.random_uniform
        alpha = tf.random.normal([batch_size, 1, 1, 1], 0.0, 1.0)
        diff = fake_images - real_images
        interpolated = real_images + alpha * diff

        with tf.GradientTape() as gp_tape:
            gp_tape.watch(interpolated)
            pred = self.critic(interpolated, training=True)

        grads = gp_tape.gradient(pred, [interpolated])[0]
        norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
        gp = tf.reduce_mean((norm - 1.0) ** 2)
        return gp

    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]

        for i in range(self.critic_steps):  #2
            random_latent_vectors = tf.random.normal(
                shape=(batch_size, self.latent_dim)
            )

            with tf.GradientTape() as tape:
                fake_images = self.generator(
                    random_latent_vectors, training=True
                )
                fake_predictions = self.critic(fake_images,
                                               training=True)
                real_predictions = self.critic(real_images,
                                               training=True)
                c_wass_loss = tf.reduce_mean(
                    fake_predictions) - tf.reduce_mean(real_predictions
                )
                c_gp = self.gradient_penalty(
                    batch_size, real_images, fake_images
                )
                c_loss = c_wass_loss + c_gp * self.gp_weight
            c_gradient = tape.gradient(c_loss,
                                       self.critic.trainable_variables)
            self.c_optimizer.apply_gradients(
                zip(c_gradient, self.critic.trainable_variables)
            )
            random_latent_vectors = tf.random.normal(
                shape=(batch_size, self.latent_dim)
            )
            with tf.GradientTape() as tape:
                fake_images = self.generator(random_latent_vectors,
                                             training=True)
                fake_predictions = self.critic(fake_images,
                                               training=True)
                g_loss = -tf.reduce_mean(fake_predictions)
            gen_gradient = tape.gradient(g_loss,
                                         self.generator.trainable_variables)
            self.g_optimizer.apply_gradients(
                zip(gen_gradient, self.generator.trainable_variables)
            )
            self.c_loss_metric.update_state(c_loss)
            self.c_wass_loss_metric.update_state(c_wass_loss)
            self.c_gp_metric.update_state(c_gp)
            self.g_loss_metric.update_state(g_loss)

            return {m.name: m.result() for m in self.metrics}
```

В методе `train_step`:

1. выполняется несколько обновлений критика
2. рассчитывается потеря Вассерштайна для критика — разница между средними прогнозом для фальшивых изображений и реальных изображений
3. вычисляется градиентный штраф
4. функция потерь критика представляет собой взвешенную сумму потерь Вассерштайна и градиентного штрафа
5. обновляются веса критика
6. рассчитывается потеря Вассерштайна для генератора
7. обновляются веса генератора

Пакетная нормализация в WGAN-GP

Последнее соображение, на которое мы должны обратить внимание перед обучением WGAN-GP, заключается в том, что в критике не следует использовать пакетную нормализацию. Это связано с тем, что пакетная нормализация создает корреляцию между изображениями в одном пакете, что делает потери выше градиентного штрафа менее эффективным. Экспериментально, что WGAN-GP по-прежнему могут давать отличные результаты даже без пакетной нормализации в критике.

Итак, ключевыми различиями между стандартным GAN и WGAN-GP состоят в следующем:

- WGAN-GP использует потери Вассерштайна.
- WGAN-GP обучается с использованием метки 1 для настоящего изображения и -1 для поддельных изображения.
- В последнем слое критика нет активации сигмоидой.
- В функции потерь для критика включен градиентный штраф.
- Критик обучается несколько раз для каждого обновления генератора.
- В критике нет слоев пакетной нормализации.

Параметры

- `IMAGE_SIZE` - количество пикселей по ширине и высоте изображения
- `CHANNELS` - количество каналов изображения (слоев с информацией о цвете)
- `BATCH_SIZE` - размер пакета с изображениями при загрузке и последующем использовании
- `Z_DIM` - размерность латентного пространства
- `LEARNING_RATE` - скорость обучения нейронной сети
- `EPOCHS` - количество эпох
- `CRITIC_STEPS` - количество шагов обучения критика на один шаг обучения генератора
- `GP_WEIGHT` - весовой коэффициент градиентного штрафа
- `LOAD_MODEL` - параметр загрузки модели
- `ADAM_BETA_1`, `ADAM_BETA_2` - параметры оптимизатора Adam

```
IMAGE_SIZE = 64
CHANNELS = 3
BATCH_SIZE = 512
Z_DIM = 128
LEARNING_RATE = 0.0002
EPOCHS = 100
CRITIC_STEPS = 3
GP_WEIGHT = 10.0
LOAD_MODEL = False
ADAM_BETA_1 = 0.5
ADAM_BETA_2 = 0.9
```

Подготовка данных

```
# Load the data
train_data = utils.image_dataset_from_directory(
    "../2 vae/celeba-dataset/img_align_celeba/img_align_celeba",
    labels=None,
    color_mode="rgb",
    image_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE,
    shuffle=True,
    seed=42,
    interpolation="bilinear",
)
```

```
Found 202599 files.
```

```
WARNING: All messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1759510119.551650 6738123 gpu_device_factory.cc:305] Could not identif
ully NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been bui
lt with NUMA support.
I0000 00:00:1759510119.551650 6738123 gpu_device_factory.cc:271] Created TensorF
low device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical
PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
```

```
# Preprocess the data
def preprocess(img):
    """
    Normalize and reshape the image
    """
    img = (tf.cast(img, "float32") - 127.5) / 127.5
    return img

train = train_data.map(lambda x: preprocess(x))
```

```
# Show some faces from the training set
train_sample = sample_batch(train)
```

```
display(train_sample, cmap=None)
```


Построение WGAN-GP

```
critic_input = layers.Input(shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS))
x = layers.Conv2D(64, kernel_size=4, strides=2, padding="same")(critic_input)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2D(128, kernel_size=4, strides=2, padding="same")(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(256, kernel_size=4, strides=2, padding="same")(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(512, kernel_size=4, strides=2, padding="same")(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(1, kernel_size=4, strides=1, padding="valid")(x)
critic_output = layers.Flatten()(x)

critic = models.Model(critic_input, critic_output)
critic.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 64, 64, 3)	0
conv2d (Conv2D)	(None, 32, 32, 64)	3,136
leaky_re_lu (LeakyReLU)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 16, 16, 128)	131,200
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 128)	0
dropout (Dropout)	(None, 16, 16, 128)	0
conv2d_2 (Conv2D)	(None, 8, 8, 256)	524,544
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 256)	0
dropout_1 (Dropout)	(None, 8, 8, 256)	0
conv2d_3 (Conv2D)	(None, 4, 4, 512)	2,097,664
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 512)	0
dropout_2 (Dropout)	(None, 4, 4, 512)	0
conv2d_4 (Conv2D)	(None, 1, 1, 1)	8,193
flatten (Flatten)	(None, 1)	0

Total params: 2,764,737 (10.55 MB)
Trainable params: 2,764,737 (10.55 MB)
Non-trainable params: 0 (0.00 B)

```
generator_input = layers.Input(shape=(Z_DIM,))
x = layers.Reshape((1, 1, Z_DIM))(generator_input)
x = layers.Conv2DTranspose(
    512, kernel_size=4, strides=1, padding="valid", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    256, kernel_size=4, strides=2, padding="same", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    128, kernel_size=4, strides=2, padding="same", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    64, kernel_size=4, strides=2, padding="same", use_bias=False
)(x)
x = layers.BatchNormalization(momentum=0.9)(x)
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2DTranspose(
    CHANNELS, kernel_size=4, strides=2, padding="same", activation="tanh"
)(x)
generator = models.Model(generator_input, generator_output)
generator.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 128)	0
reshape (Reshape)	(None, 1, 1, 128)	0
conv2d_transpose (Conv2DTranspose)	(None, 4, 4, 512)	1,048,576
batch_normalization (BatchNormalization)	(None, 4, 4, 512)	2,048
leaky_re_lu_4 (LeakyReLU)	(None, 4, 4, 512)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 8, 8, 256)	2,097,152
batch_normalization_1 (BatchNormalization)	(None, 8, 8, 256)	1,024
leaky_re_lu_5 (LeakyReLU)	(None, 8, 8, 256)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 16, 16, 128)	524,288
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 128)	512
leaky_re_lu_6 (LeakyReLU)	(None, 16, 16, 128)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 32, 32, 64)	131,072
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 64)	256
leaky_re_lu_7 (LeakyReLU)	(None, 32, 32, 64)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 64, 64, 3)	3,075

Total params: 3,808,003 (14.53 MB)
Trainable params: 3,806,403 (14.52 MB)
Non-trainable params: 1,920 (7.50 KB)

```
class WGANGPModel(models.Model):
    def __init__(self, critic, generator, latent_dim,
                 critic_steps=3, gp_weight=10.0):
        super(WGANGP, self).__init__()
        self.critic = critic
        self.generator = generator
        self.latent_dim = latent_dim
        self.critic_steps = critic_steps
        self.gp_weight = gp_weight

    def compile(self, c_optimizer, g_optimizer):
        super(WGANGP, self).compile()
        self.c_optimizer = c_optimizer
        self.g_optimizer = g_optimizer
        self.c_wass_loss_metric = metrics.Mean(name="c_wass_loss")
        self.c_gp_metric = metrics.Mean(name="c_gp")
        self.c_loss_metric = metrics.Mean(name="c_loss")
        self.g_loss_metric = metrics.Mean(name="g_loss")

    @property
    def metrics(self):
        return [
            self.c_loss_metric,
            self.c_wass_loss_metric,
            self.c_gp_metric,
            self.g_loss_metric,
        ]

    def gradient_penalty(self, batch_size, real_images, fake_images):
        # 1.5.random_uniform
        alpha = tf.random.normal([batch_size, 1, 1, 1], 0.0, 1.0)
        diff = fake_images - real_images
        interpolated = real_images + alpha * diff

        with tf.GradientTape() as gp_tape:
            gp_tape.watch(interpolated)
            pred = self.critic(interpolated, training=True)

        grads = gp_tape.gradient(pred, [interpolated])[0]
        norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
        gp = tf.reduce_mean((norm - 1.0) ** 2)
        return gp

    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]

        for i in range(self.critic_steps):  #2
            random_latent_vectors = tf.random.normal(
                shape=(batch_size, self.latent_dim)
            )

            with tf.GradientTape() as tape:
                fake_images = self.generator(
                    random_latent_vectors, training=True
                )
                fake_predictions = self.critic(fake_images,
                                               training=True)
                real_predictions = self.critic(real_images,
                                               training=True)
                c_wass_loss = tf.reduce_mean(
                    fake_predictions) - tf.reduce_mean(real_predictions
                )
                c_gp = self.gradient_penalty(
                    batch_size, real_images, fake_images
                )
                c_loss = c_wass_loss + c_gp * self.gp_weight
            c_gradient = tape.gradient(c_loss,
                                       self.critic.trainable_variables)
            self.c_optimizer.apply_gradients(
                zip(c_gradient, self.critic.trainable_variables)
            )
            random_latent_vectors = tf.random.normal(
                shape=(batch_size, self.latent_dim)
            )
            with tf.GradientTape() as tape:
                fake_images = self.generator(random_latent_vectors,
                                             training=True)
                fake_predictions = self.critic(fake_images,
                                               training=True)
                g_loss = -tf.reduce_mean(fake_predictions)
            gen_gradient = tape.gradient(g_loss,
                                         self.generator.trainable_variables)
            self.g_optimizer.apply_gradients(
                zip(gen_gradient, self.generator.trainable_variables)
            )
            self.c_loss_metric.update_state(c_loss)
            self.c_wass_loss_metric.update_state(c_wass_loss)
            self.c_gp_metric.update_state(c_gp)
            self.g_loss_metric.update_state(g_loss)

            return {m.name: m.result() for m in self.metrics}
```

В методе `train_step`:

1. выполняется несколько обновлений критика
2. рассчитывается потеря Вассерштайна для критика — разница между средними прогнозом для фальшивых изображений и реальных изображений
3. вычисляется градиентный штраф
4. функция потерь критика представляет собой взвешенную сумму потерь Вассерштайна и градиентного штрафа
5. обновляются вес критика
6. рассчитывается потеря Вассерштайна для генератора
7. обновляются веса генератора

```
                c_wass_loss = tf.reduce_mean(
                    fake_predictions) - tf.reduce_mean(real_predictions
                )#2
                c_gp = self.gradient_penalty(
                    partial_size, real_images, fake_images
                )#2
                c_loss = c_wass_loss + c_gp * self.gp_weight #4

                c_gradient = tape.gradient(c_loss,
                                           self.critic.trainable_variables)
                self.c_optimizer.apply_gradients(
                    zip(c_gradient, self.critic.trainable_variables)
                )#5

            random_latent_vectors = tf.random.normal(
                shape=(batch_size, self.latent_dim)
            )
            with tf.GradientTape() as tape:
                fake_images = self.generator(random_latent_vectors,
                                             training=True)
                fake_predictions = self.critic(fake_images,
                                               training=True)
                g_loss = -tf.reduce_mean(fake_predictions) #6

            gen_gradient = tape.gradient(g_loss,
                                         self.generator.trainable_variables)
            self.g_optimizer.apply_gradients(
                zip(gen_gradient, self.generator.trainable_variables)
            )#7

            self.c_loss_metric.update_state(c_loss)
            self.c_wass_loss_metric.update_state(c_wass_loss)
            self.c_gp_metric.update_state(c_gp)
            self.g_loss_metric.update_state(g_loss)

            return {m.name: m.result() for m in self.metrics}
```

In [13]:
```
# Create a GAN
wganp = WGANGP(
    critic=critic,
    generator=generator,
    latent_dim=2_DIM,
    critic_steps=CRITIC_STEPS,
    gp_weight=GP_WEIGHT,
)
```

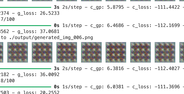
In [14]:
```
if LOAD_MODEL:
    wganp.load_weights("./checkpoint/checkpoint.ckpt")
```

In [15]:
```
# Compile the GAN
wganp.compile(
    c_optimizer=optimizers.Adam(
        learning_rate=LEARNING_RATE, beta_1=ADAM_BETA_1, beta_2=ADAM_BETA_2
    ),
    g_optimizer=optimizers.Adam(
        learning_rate=LEARNING_RATE, beta_1=ADAM_BETA_1, beta_2=ADAM_BETA_2
    ),
)
```

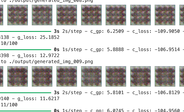
In [17]:
```
import os
# Create subdirectories
for path in ["./checkpoint","./logs","./models","./output"]:
    try:
        os.mkdir(path)
        print("Directory %s' created successfully" % path)
    except OSError as error:
        print("Directory %s' exists" % path)

Directory './checkpoint' created successfully
Directory './logs' created successfully
Directory './models' created successfully
Directory './output' created successfully
```

In [18]:
```
# Create a model save checkpoint
model_checkpoint_callback = callbacks.ModelCheckpoint(
    filepath="./checkpoint/checkpoint.weights.h5",
    save_weights_only=True,
    save_freq="epoch",
    verbose=0,
)

tensorboard_callback = callbacks.TensorBoard(log_dir="./logs")


class ImageGenerator(callbacks.Callback):
    def __init__(self, num_img, latent_dim):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.normal(
            shape=(self.num_img, self.latent_dim)
        )
        generated_images = self.model.generator(random_latent_vectors)
        generated_images = generated_images * 127.5 + 127.5
        generated_images = generated_images.numpy()
        display(
            generated_images,
            save_to="./output/generated_img_%03d.png" % (epoch),
            cmap=None,
        )

wganp.fit(
    train,
    epochs=EPOCHS,
    steps_per_epoch=2,
    callbacks=[
#ckpt_callback,
        model_checkpoint_callback,
        tensorboard_callback,
        ImageGenerator(num_img=10, latent_dim=2_DIM),
    ],
)
```

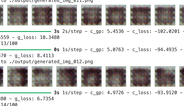
```
Epoch 1/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_loss: 0.7421 - c_loss: 5.9339 - c_wass_loss: -1.
4614 - g_loss: 0.5939
Saved to ./output/generated_img_000.png
```



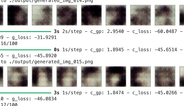
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 18s 2s/step - c_gp: 0.7283 - c_loss: 5.5261 - c_wass_loss: -
1.7572 - g_loss: 0.6635
Epoch 2/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3568 - c_loss: -24.7521 - c_wass_loss: -
28.3204 - g_loss: 5.1550
Saved to ./output/generated_img_001.png
```



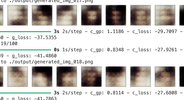
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - c_gp: 0.4236 - c_loss: -27.7684 - c_wass_loss: -
32.0408 - g_loss: 5.6280
Epoch 3/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 2.4945 - c_loss: -70.2611 - c_wass_loss: -
95.3905 - g_loss: 17.8862
Saved to ./output/generated_img_002.png
```



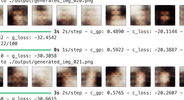
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - c_gp: 2.5606 - c_loss: -71.2084 - c_wass_loss: -
96.8144 - g_loss: 18.8347
Epoch 4/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 3.8421 - c_loss: -91.7473 - c_wass_loss: -
130.1687 - g_loss: 21.0103
Saved to ./output/generated_img_003.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 3.9052 - c_loss: -92.6450 - c_wass_loss: -
131.6973 - g_loss: 20.7760
Epoch 5/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 4.5245 - c_loss: -103.8052 - c_wass_loss:
-148.3305 - g_loss: 18.1408
Saved to ./output/generated_img_004.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 4.5944 - c_loss: -103.6818 - c_wass_loss:
-149.6263 - g_loss: 18.4303
Epoch 6/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 5.8572 - c_loss: -111.6585 - c_wass_loss:
-178.2386 - g_loss: 25.7255
Saved to ./output/generated_img_005.png
```

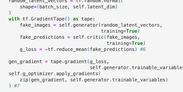


```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 5.8795 - c_loss: -111.4422 - c_wass_loss:
-178.2374 - g_loss: 26.5133
Epoch 7/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 6.4486 - c_loss: -112.1899 - c_wass_loss:
-176.8562 - g_loss: 37.8681
Saved to ./output/generated_img_006.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 6.3816 - c_loss: -112.4827 - c_wass_loss:
-176.2182 - c_loss: 36.0092
Epoch 8/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 6.8381 - c_loss: -111.3696 - c_wass_loss:
-171.7583 - g_loss: 20.2552
Saved to ./output/generated_img_007.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 6.8541 - c_loss: -111.1700 - c_wass_loss:
-171.6787 - g_loss: 20.7155
Epoch 9/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 6.1387 - c_loss: -109.9573 - c_wass_loss:
-157.1644 - g_loss: 24.7905
Saved to ./output/generated_img_008.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 6.2509 - c_loss: -109.9050 - c_wass_loss:
-172.4138 - g_loss: 25.1852
Epoch 10/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 5.8888 - c_loss: -106.9514 - c_wass_loss:
-165.8398 - g_loss: 12.9272
Saved to ./output/generated_img_009.png
```

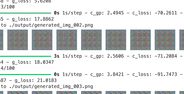


```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 5.8141 - c_loss: -106.8129 - c_wass_loss:
-164.9540 - g_loss: 11.6217
Epoch 11/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 6.0745 - c_loss: -104.9560 - c_wass_loss:
-165.7007 - g_loss: 18.5198
Saved to ./output/generated_img_010.png
```

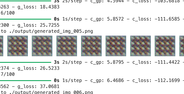
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 6.0616 - c_loss: -104.6406 - c_wass_loss:
-165.2568 - g_loss: 11.5423
Epoch 12/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 5.4485 - c_loss: -102.2222 - c_wass_loss:
-156.9075 - g_loss: 10.6294
Saved to ./output/generated_img_011.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 5.4536 - c_loss: -102.8291 - c_wass_loss:
-156.5559 - g_loss: 10.3408
Epoch 13/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 5.0763 - c_loss: -94.4935 - c_wass_loss: -
145.2578 - g_loss: 8.4161
Saved to ./output/generated_img_012.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 4.9726 - c_loss: -93.9120 - c_wass_loss: -
143.6388 - g_loss: 6.7354
Epoch 14/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 4.1529 - c_loss: -81.3243 - c_wass_loss: -
122.0529 - g_loss: -12.3202
Saved to ./output/generated_img_013.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 4.1119 - c_loss: -80.5521 - c_wass_loss: -
121.6718 - g_loss: -13.1346
Epoch 15/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 3.8141 - c_loss: -61.0056 - c_wass_loss: -
91.1469 - g_loss: -31.4713
Saved to ./output/generated_img_014.png
```



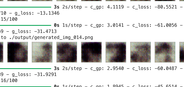
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 2.9540 - c_loss: -60.0487 - c_wass_loss: -
89.5889 - g_loss: -31.9291
Epoch 16/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 1.8945 - c_loss: -45.6514 - c_wass_loss: -
64.5965 - g_loss: -45.8928
Saved to ./output/generated_img_015.png
```



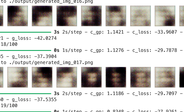
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 1.8474 - c_loss: -45.6206 - c_wass_loss: -
63.5818 - g_loss: -46.4816
Epoch 17/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 1.1641 - c_loss: -34.3005 - c_wass_loss: -
45.9413 - g_loss: -42.4594
Saved to ./output/generated_img_016.png
```



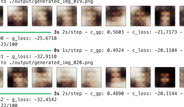
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 1.1421 - c_loss: -33.9687 - c_wass_loss: -
45.3021 - g_loss: -42.8274
Epoch 18/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 1.1276 - c_loss: -29.7878 - c_wass_loss: -
40.8035 - g_loss: -37.3984
Saved to ./output/generated_img_017.png
```



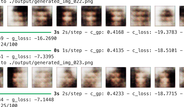
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 1.1106 - c_loss: -29.7097 - c_wass_loss: -
40.8068 - g_loss: -37.5755
Epoch 19/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.8348 - c_loss: -27.9261 - c_wass_loss: -
36.2739 - g_loss: -41.6060
Saved to ./output/generated_img_018.png
```



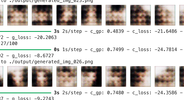
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.8514 - c_loss: -27.6080 - c_wass_loss: -
35.7958 - g_loss: -41.7083
Epoch 20/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.5668 - c_loss: -21.0568 - c_wass_loss: -
27.5249 - g_loss: -25.8364
Saved to ./output/generated_img_019.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.5603 - c_loss: -21.7173 - c_wass_loss: -
27.3208 - g_loss: -25.6710
Epoch 21/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.4924 - c_loss: -20.1184 - c_wass_loss: -
25.8341 - g_loss: -32.9110
Saved to ./output/generated_img_020.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.4890 - c_loss: -20.1144 - c_wass_loss: -
25.8042 - g_loss: -32.4542
Epoch 22/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.5922 - c_loss: -20.3087 - c_wass_loss: -
26.3118 - g_loss: -30.5058
Saved to ./output/generated_img_021.png
```

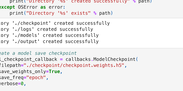


```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.5765 - c_loss: -20.2607 - c_wass_loss: -
26.4260 - g_loss: -30.6615
Epoch 23/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.4145 - c_loss: -19.5433 - c_wass_loss: -
23.6886 - g_loss: -16.8575
Saved to ./output/generated_img_022.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.4160 - c_loss: -19.3783 - c_wass_loss: -
23.5459 - g_loss: -16.2208
Epoch 24/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.4135 - c_loss: -18.5581 - c_wass_loss: -
22.6453 - g_loss: -7.3395
Saved to ./output/generated_img_023.png
```

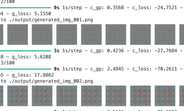
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.4233 - c_loss: -18.7715 - c_wass_loss: -
23.0044 - g_loss: -7.1448
Epoch 25/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.5870 - c_loss: -20.1834 - c_wass_loss: -
25.9733 - g_loss: -14.4559
Saved to ./output/generated_img_024.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - c_gp: 0.5043 - c_loss: -19.0824 - c_wass_loss: -
25.4456 - g_loss: -14.6442
Epoch 26/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.4729 - c_loss: -21.4358 - c_wass_loss: -
26.1649 - g_loss: -20.4[illegible]
Saved to ./output/generated_img_025.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.4839 - c_loss: -21.6406 - c_wass_loss: -
26.4872 - g_loss: -20.2063
Epoch 27/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.7499 - c_loss: -26.7614 - c_wass_loss: -
32.2002 - g_loss: -8.6727
Saved to ./output/generated_img_026.png
```

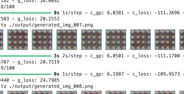
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.7460 - c_loss: -24.3506 - c_wass_loss: -
31.8387 - g_loss: -9.2743
Epoch 28/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.4811 - c_loss: -18.4045 - c_wass_loss: -
22.4157 - g_loss: -0.2845
Saved to ./output/generated_img_027.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - c_gp: 0.4481 - c_loss: -18.3975 - c_wass_loss: -
22.1983 - g_loss: -8.0656
Epoch 29/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.5371 - c_loss: -20.8468 - c_wass_loss: -
26.2188 - g_loss: -8.5423
Saved to ./output/generated_img_028.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.5366 - c_loss: -20.9591 - c_wass_loss: -
26.3246 - g_loss: -7.8787
Epoch 30/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.6339 - c_loss: -22.1338 - c_wass_loss: -
28.4526 - g_loss: 1.6337
Saved to ./output/generated_img_029.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.8374 - c_loss: -21.8262 - c_wass_loss: -
27.9998 - g_loss: 0.7064
Epoch 31/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3749 - c_loss: -17.0973 - c_wass_loss: -
20.8462 - g_loss: -3.1557
Saved to ./output/generated_img_030.png
```

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3815 - c_loss: -17.0991 - c_wass_loss: -
19.9141 - g_loss: -4.7891
Epoch 31/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3730 - c_loss: -19.7697 - c_wass_loss: -
23.4993 - g_loss: -7.2872
Saved to ./output/generated_img_031.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3894 - c_loss: -19.8921 - c_wass_loss: -
23.7856 - g_loss: -6.7791
Epoch 33/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.5154 - c_loss: -20.3547 - c_wass_loss: -
25.5085 - g_loss: 3.5661
Saved to ./output/generated_img_032.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.5372 - c_loss: -20.2520 - c_wass_loss: -
25.4245 - g_loss: 2.0747
Epoch 34/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.4462 - c_loss: -19.8699 - c_wass_loss: -
24.3318 - g_loss: -11.2161
Saved to ./output/generated_img_033.png

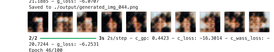
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4447 - c_loss: -19.8144 - c_wass_loss: -
24.2619 - g_loss: -11.0741
Epoch 35/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3634 - c_loss: -16.0867 - c_wass_loss: -
19.5211 - g_loss: -4.7420
Saved to ./output/generated_img_034.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3626 - c_loss: -17.0379 - c_wass_loss: -
20.6435 - g_loss: -4.0050
Epoch 36/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4604 - c_loss: -19.8961 - c_wass_loss: -
23.6998 - g_loss: -15.9704
Saved to ./output/generated_img_035.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4594 - c_loss: -19.2538 - c_wass_loss: -
23.8482 - g_loss: -16.0378
Epoch 37/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.5825 - c_loss: -19.0352 - c_wass_loss: -
24.0598 - g_loss: 4.3859
Saved to ./output/generated_img_036.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - g_cgp: 0.5006 - c_loss: -19.4985 - c_wass_loss: -
24.5842 - g_loss: 3.7361
Epoch 38/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3624 - c_loss: -17.0777 - c_wass_loss: -
20.7016 - g_loss: -11.7588
Saved to ./output/generated_img_037.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - g_cgp: 0.3005 - c_loss: -17.0739 - c_wass_loss: -
20.6786 - g_loss: -12.1215
Epoch 39/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4475 - c_loss: -20.6275 - c_wass_loss: -
25.1029 - g_loss: -3.9594
Saved to ./output/generated_img_038.png



2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4602 - c_loss: -20.8159 - c_wass_loss: -
25.4175 - g_loss: -2.9412
Epoch 40/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.5600 - c_loss: -19.6665 - c_wass_loss: -
25.2663 - g_loss: -18.2735
Saved to ./output/generated_img_039.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - g_cgp: 0.5473 - c_loss: -19.4808 - c_wass_loss: -
24.9716 - g_loss: -11.1146
Epoch 41/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2911 - c_loss: -15.3689 - c_wass_loss: -
18.2719 - g_loss: -10.4966
Saved to ./output/generated_img_040.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.2936 - c_loss: -15.3579 - c_wass_loss: -
18.2942 - g_loss: -8.4103
Epoch 42/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4170 - c_loss: -17.0185 - c_wass_loss: -
21.1887 - g_loss: -9.5867
Saved to ./output/generated_img_041.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4120 - c_loss: -16.9560 - c_wass_loss: -
21.0757 - g_loss: -8.3444
Epoch 43/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3675 - c_loss: -17.5711 - c_wass_loss: -
21.6403 - g_loss: -1.3076
Saved to ./output/generated_img_042.png

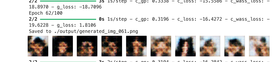
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3808 - c_loss: -17.7951 - c_wass_loss: -
21.4755 - g_loss: -0.5932
Epoch 44/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4836 - c_loss: -20.3551 - c_wass_loss: -
25.1915 - g_loss: -1.4559
Saved to ./output/generated_img_043.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4001 - c_loss: -20.4525 - c_wass_loss: -
23.3138 - g_loss: 0.0578
Epoch 45/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4411 - c_loss: -16.5777 - c_wass_loss: -
21.1885 - g_loss: -6.0787
Saved to ./output/generated_img_044.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4423 - c_loss: -16.3614 - c_wass_loss: -
20.7244 - g_loss: -6.2531
Epoch 46/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3200 - c_loss: -15.0011 - c_wass_loss: -
18.2887 - g_loss: 7.0958
Saved to ./output/generated_img_045.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3430 - c_loss: -15.2015 - c_wass_loss: -
18.6311 - g_loss: 3.5568
Epoch 47/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3607 - c_loss: -19.6358 - c_wass_loss: -
23.3028 - g_loss: -2.4585
Saved to ./output/generated_img_046.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3907 - c_loss: -19.9831 - c_wass_loss: -
23.8896 - g_loss: -2.3773
Epoch 48/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.5216 - c_loss: -18.5575 - c_wass_loss: -
22.7736 - g_loss: -5.2793
Saved to ./output/generated_img_047.png

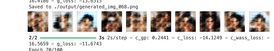


2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4992 - c_loss: -18.1928 - c_wass_loss: -
22.1044 - g_loss: -6.1712
Epoch 49/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2822 - c_loss: -15.3550 - c_wass_loss: -
18.1776 - g_loss: -15.8041
Saved to ./output/generated_img_048.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2842 - c_loss: -15.2622 - c_wass_loss: -
18.1044 - g_loss: -16.8487
Epoch 50/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2781 - c_loss: -16.8365 - c_wass_loss: -
19.6173 - g_loss: 14.8910
Saved to ./output/generated_img_049.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2871 - c_loss: -16.9550 - c_wass_loss: -
19.8257 - g_loss: 13.6647
Epoch 51/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.5491 - c_loss: -19.7954 - c_wass_loss: -
24.8860 - g_loss: 7.1487
Saved to ./output/generated_img_050.png

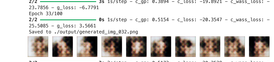
2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - g_cgp: 0.5102 - c_loss: -19.8204 - c_wass_loss: -
24.9222 - g_loss: 6.0188
Epoch 52/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4297 - c_loss: -17.1991 - c_wass_loss: -
21.4905 - g_loss: -0.2059
Saved to ./output/generated_img_051.png



2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4343 - c_loss: -17.8141 - c_wass_loss: -
21.3771 - g_loss: -4.6756
Epoch 53/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2312 - c_loss: -14.3482 - c_wass_loss: -
16.6606 - g_loss: -4.8252
Saved to ./output/generated_img_052.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - g_cgp: 0.2356 - c_loss: -14.5270 - c_wass_loss: -
16.8833 - g_loss: -9.0396
Epoch 54/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4593 - c_loss: -20.7422 - c_wass_loss: -
25.3352 - g_loss: -3.0999
Saved to ./output/generated_img_053.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4795 - c_loss: -20.7782 - c_wass_loss: -
25.5715 - g_loss: -2.1548
Epoch 55/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3760 - c_loss: -15.7865 - c_wass_loss: -
19.5462 - g_loss: -6.9615
Saved to ./output/generated_img_054.png



2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3648 - c_loss: -15.6623 - c_wass_loss: -
19.3183 - g_loss: -7.9012
Epoch 56/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2807 - c_loss: -15.4132 - c_wass_loss: -
18.2199 - g_loss: 3.2851
Saved to ./output/generated_img_055.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2845 - c_loss: -15.6530 - c_wass_loss: -
18.4038 - g_loss: 1.6890
Epoch 57/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.4772 - c_loss: -21.8251 - c_wass_loss: -
25.7067 - g_loss: 9.9062
Saved to ./output/generated_img_056.png

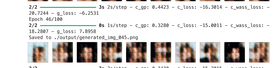
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.5043 - c_loss: -21.1943 - c_wass_loss: -
26.2378 - g_loss: 10.3569
Epoch 58/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4318 - c_loss: -16.6483 - c_wass_loss: -
20.9582 - g_loss: -22.2330
Saved to ./output/generated_img_057.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.4160 - c_loss: -16.5108 - c_wass_loss: -
20.6709 - g_loss: -21.4477
Epoch 59/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3571 - c_loss: -15.8841 - c_wass_loss: -
19.4554 - g_loss: -22.6788
Saved to ./output/generated_img_058.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3521 - c_loss: -15.8054 - c_wass_loss: -
19.3866 - g_loss: -19.9751
Epoch 60/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3664 - c_loss: -17.0925 - c_wass_loss: -
20.7566 - g_loss: -29.3474
Saved to ./output/generated_img_059.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3605 - c_loss: -17.0745 - c_wass_loss: -
20.6794 - g_loss: -27.1717
Epoch 61/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3366 - c_loss: -15.5819 - c_wass_loss: -
18.9476 - g_loss: -17.9478
Saved to ./output/generated_img_060.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3338 - c_loss: -15.5586 - c_wass_loss: -
18.8970 - g_loss: -18.7096
Epoch 62/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3196 - c_loss: -16.6272 - c_wass_loss: -
19.6228 - g_loss: 1.8386
Saved to ./output/generated_img_061.png



2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - g_cgp: 0.3194 - c_loss: -16.2842 - c_wass_loss: -
19.4778 - g_loss: -0.5895
Epoch 63/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3450 - c_loss: -15.1659 - c_wass_loss: -
18.6163 - g_loss: -15.4562
Saved to ./output/generated_img_062.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3420 - c_loss: -14.9463 - c_wass_loss: -
18.3659 - g_loss: -13.6022
Epoch 64/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2786 - c_loss: -15.8844 - c_wass_loss: -
17.7002 - g_loss: -14.1027
Saved to ./output/generated_img_063.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2715 - c_loss: -15.2777 - c_wass_loss: -
17.9925 - g_loss: -13.5454
Epoch 65/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3002 - c_loss: -18.8095 - c_wass_loss: -
22.8916 - g_loss: -9.8476
Saved to ./output/generated_img_064.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3030 - c_loss: -17.9209 - c_wass_loss: -
21.7509 - g_loss: -7.8118
Epoch 66/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2683 - c_loss: -13.5723 - c_wass_loss: -
16.2558 - g_loss: -2.6757
Saved to ./output/generated_img_065.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2635 - c_loss: -13.5574 - c_wass_loss: -
16.1527 - g_loss: -5.9605
Epoch 67/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2432 - c_loss: -14.3181 - c_wass_loss: -
16.7499 - g_loss: -22.0001
Saved to ./output/generated_img_066.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2408 - c_loss: -14.4920 - c_wass_loss: -
16.9096 - g_loss: -24.1429
Epoch 68/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2694 - c_loss: -15.4198 - c_wass_loss: -
18.1138 - g_loss: -0.5667
Saved to ./output/generated_img_067.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2688 - c_loss: -15.3861 - c_wass_loss: -
17.9948 - g_loss: -3.8262
Epoch 69/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2435 - c_loss: -13.9837 - c_wass_loss: -
16.4186 - g_loss: -13.6315
Saved to ./output/generated_img_068.png

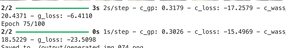
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2041 - c_loss: -14.1249 - c_wass_loss: -
16.5659 - g_loss: -11.6743
Epoch 70/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3291 - c_loss: -15.2596 - c_wass_loss: -
18.5585 - g_loss: -3.1937
Saved to ./output/generated_img_069.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.3277 - c_loss: -15.2297 - c_wass_loss: -
16.5803 - g_loss: -5.4813
Epoch 71/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2858 - c_loss: -15.5330 - c_wass_loss: -
18.3712 - g_loss: -18.7596
Saved to ./output/generated_img_070.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2093 - c_loss: -15.5808 - c_wass_loss: -
18.4986 - g_loss: -12.5878
Epoch 72/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2951 - c_loss: -14.8841 - c_wass_loss: -
17.8349 - g_loss: -31.1045
Saved to ./output/generated_img_071.png

2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.2926 - c_loss: -14.7725 - c_wass_loss: -
17.6988 - g_loss: -31.3580
Epoch 73/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - g_cgp: 0.1855 - c_loss: -12.2080 - c_wass_loss: -
14.8035 - g_loss: -34.6411
Saved to ./output/generated_img_072.png

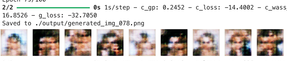
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.1842 - c_loss: -12.3329 - c_wass_loss: -
14.1752 - g_loss: -33.2482
Epoch 74/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3074 - c_loss: -17.2129 - c_wass_loss: -
20.2871 - g_loss: -6.9090
Saved to ./output/generated_img_073.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.3179 - c_loss: -17.2579 - c_wass_loss: -
20.4371 - g_loss: -6.4110
Epoch 75/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3026 - c_loss: -15.4969 - c_wass_loss: -
18.5229 - g_loss: -23.5098
Saved to ./output/generated_img_074.png
```

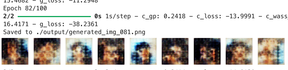
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2999 - c_loss: -15.3045 - c_wass_loss: -
18.3031 - g_loss: -23.3883
Epoch 76/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2315 - c_loss: -13.5635 - c_wass_loss: -
15.8788 - g_loss: -19.5438
Saved to ./output/generated_img_075.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2290 - c_loss: -13.6511 - c_wass_loss: -
15.9486 - g_loss: -18.2430
Epoch 77/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3108 - c_loss: -16.9851 - c_wass_loss: -
20.0928 - g_loss: -12.6437
Saved to ./output/generated_img_076.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.3191 - c_loss: -16.8244 - c_wass_loss: -
20.0152 - g_loss: -14.7762
Epoch 78/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2022 - c_loss: -11.4268 - c_wass_loss: -
13.4485 - g_loss: -25.7465
Saved to ./output/generated_img_077.png
```

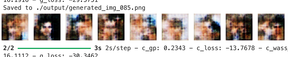
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.1964 - c_loss: -11.5399 - c_wass_loss: -
13.5040 - g_loss: -24.9115
Epoch 79/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2452 - c_loss: -14.4002 - c_wass_loss: -
16.8526 - g_loss: -32.7050
Saved to ./output/generated_img_078.png
```



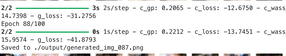
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2443 - c_loss: -14.4825 - c_wass_loss: -
16.9254 - g_loss: -31.4484
Epoch 80/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2973 - c_loss: -15.6417 - c_wass_loss: -
18.6150 - g_loss: -24.2902
Saved to ./output/generated_img_079.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.3008 - c_loss: -15.5477 - c_wass_loss: -
18.5559 - g_loss: -25.1018
Epoch 81/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2708 - c_loss: -12.7833 - c_wass_loss: -
15.4909 - g_loss: -18.2330
Saved to ./output/generated_img_080.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - c_gp: 0.2680 - c_loss: -12.7881 - c_wass_loss: -
15.4682 - g_loss: -11.2948
Epoch 82/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2418 - c_loss: -13.9991 - c_wass_loss: -
16.4171 - g_loss: -38.2361
Saved to ./output/generated_img_081.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2414 - c_loss: -14.0130 - c_wass_loss: -
16.4269 - g_loss: -36.4050
Epoch 83/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2195 - c_loss: -13.3026 - c_wass_loss: -
15.4974 - g_loss: -39.1645
Saved to ./output/generated_img_082.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2197 - c_loss: -13.3141 - c_wass_loss: -
15.5113 - g_loss: -38.8862
Epoch 84/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2246 - c_loss: -13.8557 - c_wass_loss: -
16.1022 - g_loss: -28.0713
Saved to ./output/generated_img_083.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2247 - c_loss: -13.9168 - c_wass_loss: -
16.1641 - g_loss: -28.4769
Epoch 85/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2785 - c_loss: -15.4338 - c_wass_loss: -
18.2189 - g_loss: -36.7165
Saved to ./output/generated_img_084.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2770 - c_loss: -15.3604 - c_wass_loss: -
18.1308 - g_loss: -35.2570
Epoch 86/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2357 - c_loss: -13.8351 - c_wass_loss: -
16.1916 - g_loss: -29.5751
Saved to ./output/generated_img_085.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2343 - c_loss: -13.7678 - c_wass_loss: -
16.1112 - g_loss: -30.3462
Epoch 87/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2039 - c_loss: -12.6393 - c_wass_loss: -
14.6783 - g_loss: -31.3793
Saved to ./output/generated_img_086.png
```

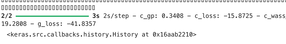
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2065 - c_loss: -12.6750 - c_wass_loss: -
14.7398 - g_loss: -31.2756
Epoch 88/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2212 - c_loss: -13.7451 - c_wass_loss: -
15.9574 - g_loss: -41.8793
Saved to ./output/generated_img_087.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2213 - c_loss: -13.9677 - c_wass_loss: -
16.1887 - g_loss: -40.8826
Epoch 89/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2764 - c_loss: -15.8300 - c_wass_loss: -
18.5941 - g_loss: -34.3520
Saved to ./output/generated_img_088.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2768 - c_loss: -15.7808 - c_wass_loss: -
18.5485 - g_loss: -33.5116
Epoch 90/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3135 - c_loss: -14.7763 - c_wass_loss: -
17.9111 - g_loss: -29.2003
Saved to ./output/generated_img_089.png
```

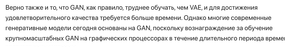
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.3120 - c_loss: -14.7647 - c_wass_loss: -
17.8843 - g_loss: -29.3679
Epoch 91/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2535 - c_loss: -13.9581 - c_wass_loss: -
16.4852 - g_loss: -32.8658
Saved to ./output/generated_img_090.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2489 - c_loss: -13.9482 - c_wass_loss: -
16.4297 - g_loss: -31.9009
Epoch 92/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2119 - c_loss: -13.7599 - c_wass_loss: -
15.8788 - g_loss: -30.8566
Saved to ./output/generated_img_091.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - c_gp: 0.2161 - c_loss: -14.0696 - c_wass_loss: -
16.2383 - g_loss: -30.8742
Epoch 93/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2655 - c_loss: -13.9468 - c_wass_loss: -
16.6021 - g_loss: -36.8230
Saved to ./output/generated_img_092.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2628 - c_loss: -13.9767 - c_wass_loss: -
16.6050 - g_loss: -37.7686
Epoch 94/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3012 - c_loss: -15.1168 - c_wass_loss: -
18.1286 - g_loss: -40.9869
Saved to ./output/generated_img_093.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - c_gp: 0.3045 - c_loss: -15.2382 - c_wass_loss: -
18.2835 - g_loss: -41.2970
Epoch 95/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2826 - c_loss: -13.9133 - c_wass_loss: -
16.7390 - g_loss: -38.9179
Saved to ./output/generated_img_094.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2790 - c_loss: -14.0359 - c_wass_loss: -
16.8255 - g_loss: -38.6885
Epoch 96/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2969 - c_loss: -14.9911 - c_wass_loss: -
17.9684 - g_loss: -47.0600
Saved to ./output/generated_img_095.png
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2941 - c_loss: -14.0648 - c_wass_loss: -
17.0058 - g_loss: -48.6049
Epoch 97/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2039 - c_loss: -14.6211 - c_wass_loss: -
16.6598 - g_loss: -42.8639
Saved to ./output/generated_img_096.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2093 - c_loss: -14.8646 - c_wass_loss: -
16.9576 - g_loss: -42.2368
Epoch 98/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3828 - c_loss: -16.6232 - c_wass_loss: -
20.4517 - g_loss: -42.5634
Saved to ./output/generated_img_097.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.3760 - c_loss: -16.3951 - c_wass_loss: -
20.1553 - g_loss: -43.8813
Epoch 99/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.2527 - c_loss: -14.5641 - c_wass_loss: -
17.0912 - g_loss: -54.4800
Saved to ./output/generated_img_098.png
```



```
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 2s/step - c_gp: 0.2518 - c_loss: -14.5487 - c_wass_loss: -
17.0663 - g_loss: -52.6531
Epoch 100/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - c_gp: 0.3410 - c_loss: -15.7678 - c_wass_loss: -
19.1781 - g_loss: -41.9830
Saved to ./output/generated_img_099.png
```



```
CPU times: user 1min 12s, sys: 1min 3s, total: 2min 15s
Wall time: 1min 12s
```

Out[20]:

```
<keras.src.callbacks.history.History at 0x16aab2210>
```

In [22]:

```
# Save the final models
generator.save("./models/generator.keras")
critic.save("./models/critic.keras")
```

Генерация изображений

In [23]:

```
z_sample = np.random.normal(size=(10, Z_DIM))
imgs = wgangp.generator.predict(z_sample)
display(imgs, cmap=None)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
```

Если сравнить выходные данные WGAN-GP с выходными данными VAE, то изображения GAN в целом более четкие, особенно четкость между волосами и фоном. В целом это верно – VAE, как правило, создают более мягкие изображения, которые размывают цветовые границы, тогда как GAN, как известно, создают более резкие и четко определенные изображения.

Верно также и то, что GAN, как правило, труднее обучать, чем VAE, и для достижения удовлетворительного качества требуется больше времени. Однако многие современные генеративные модели сегодня основаны на GAN, поскольку вознаграждение за обучение крупномасштабных GAN на графических процессорах в течение длительного периода времени является значительным.

Задание

Построить и обучить сеть WGAN-GP для набора данных Fashion MNIST (`tf.keras.datasets.fashion_mnist`), подобрав размерность латентного пространства. Использовать для обучения GAN обучающую часть набора данных Fashion MNIST.

Для случайного набора точек латентного пространства построить и визуализировать образы.

In [ ]: