Super resolucija MRI slik s pomočjo generativnih modelov

Osnove strojnega učenja in podatkovnega rudarjenja

2024/25

Jan Panjan

Contents

1. Uvod

Uporaba magnetne resonance za slikanje pacientov je izjemnega pomena za diagnozo in sledenju boleznim, ki motijo zdravje pacientov. Podatki o organih, mehkih tkivih in kosteh - ki jih pridobijo z MRI slikanjem - omogočajo zdravnikom, da bolj učinkovito ocenijo stopnjo bolezni in posledično primerno prilagodijo način zdravljenja. Pacienti so tako deležni boljšega zdravljenja, kar je še predvsem pomembno pri raznih kompleksnih boleznih. Z MRI slikanjem - kljub temu da je izjemno orodje - ni enostavno pridobiti kvalitetnih podatkov. K temu pripomore več različnih faktorjev, tako človeških kot mehanskih.

1.1. Problemi MRI slik

Čas priprave na slikanje je dolg, saj mora biti naprava kalibrirana na pacienta. Čas slikanja je dolg, saj potrebuje naprava dovolj časa, da zajame toliko informacij, da jih lahko zdravniki natančno ocenijo. Naprava mora narediti mnogo slik (t.i. *slices*) iz različnih smeri. Vse te slike se na koncu združijo v smiselno celoto (t.i. *volume*). Med slikanjem mora biti pacient na miru. Kakršnokoli premikanje vmesti v slike nezaželen šum in razne artefakte. Čas pridobivanja slike je sorazmeren s končno kvaliteto slik (manjši čas pridobivanja, manjša ločljivost). Med drugim so MRI slikanja tudi zelo draga za zdravstvene klinike kot posledica oskrbe naprave. Zaradi tega so dolžni tudi pacienti plačati več. Tu pripomorejo t.i. *low-field MRI scanners*, ki so cenejši. To omogoča, da je MRI slikanje dostopno vsem, vendar so slike pridobljene s temi napravami, relativno nižje resolucije.

Pri tem problemu lahko pomagajo SR (super resolution) metode, ki so zmožne rekonstruirati slike nizkih ločljivosti (LR, “low resolution”) v slike visokih ločljivosti (HR, “high resolution”). Tu so aktualni t.i. GAN (“generative adversarial networks”) modeli, ki so z globokim učenjem zmožni rekonstruirati slike z visoko natančnostjo. Razvitih je bilo več GAN modelov katerih namen je super resolucija slik, npr. SRGAN, ESRGAN, Real-ESRGAN,... ter tudi SRResCycGAN oziroma *Super Resolution Residual Cycle-consistent GAN*. [1]

1.2. O projektu

Za ta projekt sem si zadal implementirati SRResCycGAN model v jeziku Python z uporabo knjižnice TensorFlow. Programska koda je dostopna na github in na Google Colab. Model je bil treniran na FastMRI podatkih, specifično na t.i. *singlecoil* (2D) slikah kolen.

2. Predstavitev podatkov

Zaradi velike količine podatkov (~100GB) sem se omejil na podmnožico `singlecoil_train` in `singlecoil_val` podatkov. Imena uporabljenih datotek so vidna tu.

Podatki so shranjeni v `h5` datotekah. Vsaka datoteka predstavlja eno MRI slikanje. Na voljo so rekonstruirane slike pod ključem `reconstruction_rss` in surovi podatki k-prostora pod ključem `kspace`.

K-prostor predstavlja MRI sliko v obliki prostorskih frekvenc. Te frekvence so pridobljene preko MR signalov. Signali, ki so v osnovi kompleksna števila, so pretvorjeni v t.i. *image space* realnih števil s pomočjo inverzne Fourierjeve preslikave.

Oblike (uporabljenih) k-prostorskih podatkov se gibljejo od (28, 640, 320), kjer je 28 število slik, 640x320 pa velikost polja števil, do (50, 640, 640). Oblike rekonstruiranih slik se razlikujejo samo v številu slik, od 28 do 50 z velikostmi 320x320.

Spodaj je primer podatkov 10. slike shranjene v datoteki `file1000001.h5`.

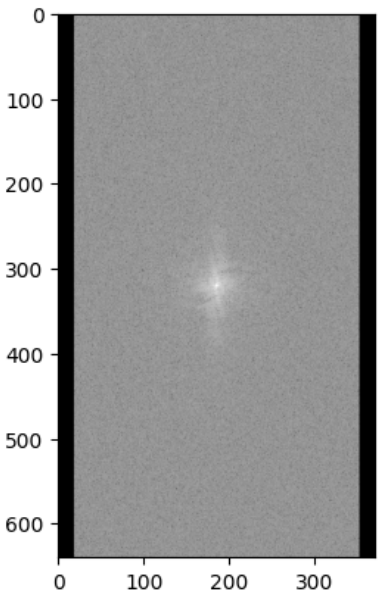


Figure 1: vizualizacija k-prostorskih podatkov

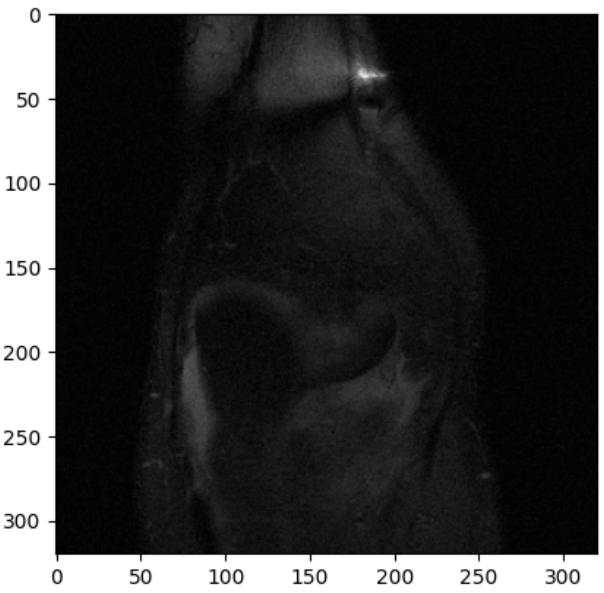


Figure 2: rekonstruirana slika pridobljena iz k-prosorskih podatkov

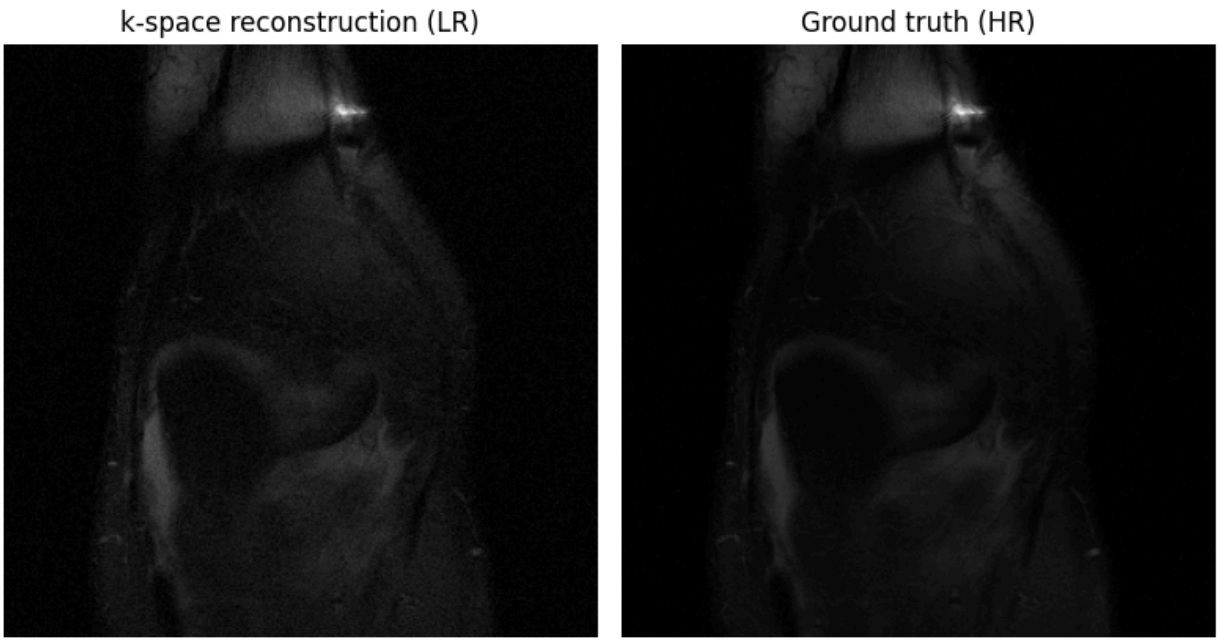


Figure 3: primerjava rekonstruirane (kspace) in očiščene slike (reconstruction_rss)

Za učenje modela je bila uporabljena množica rekonstruiranih slik (`reconstruction_rss`) in sicer v obliki **parov (LR, HR)**, torej slike nizke in visoke ločljivosti. LR slike so pridobljene iz HR slik **bikubično interpolacijo** in so velikosti 80x80, HR slike pa 320x320. Oba nabora slik sta shranjena s 3 kanali.

Metoda `create_paired_dataset` postopoma na vsaki datoteki pokliče metodo `h5_generator`, ki naloži, transformira in vrne slike kot par. Podatkovna zbirka `tf.data.Dataset` je ustvarjena z `BATCH_SIZE=8`. Programska koda je v modulu data_loader.py.

3. Predstavitev GAN strutkure

GAN modeli so v osnovi sestavljeni iz dveh nevronskih mrež - generatorja in diskriminatorja - ki med sabo tekmujeta. Cilj **generatorja** je, da se nauči porazdelitev podatkov tako, da bo sposoben generirati resnične slike. Ker sam po sebi ne more prepoznati kdaj so njegove slike resnične je tu potreben **diskriminator**. Njegova naloga je, da se nauči razlikovati med generiranimi in resničnimi slikami.

Z drugimi besedami, cilj generatorja je, da za nek vzorec LR slik ustvari neresnične HR slike, ki bodo prepričale (oz. preslepile) diskriminatorja v to, da jih oceni kot resnične. Ocene diskriminatorja o resničnosti slik so potrebne za učenje obeh modelov skozi *backpropagation*, kjer bo diskriminator kaznovan ob napačnih ocenah slike (npr. neresnično oceni kot resnično), generator pa ko bodo njegove slike (pravilno) ocenjene kot neresnične.

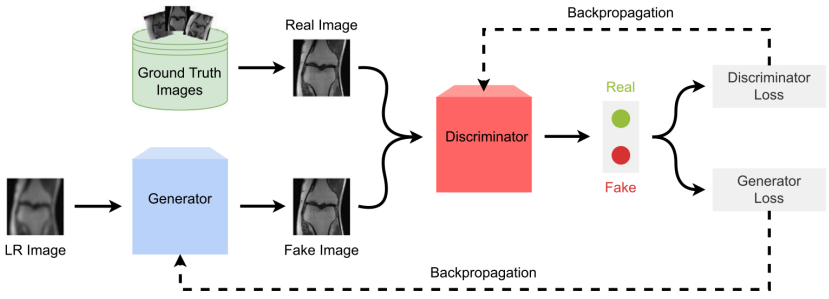


Fig. 2. Main concept behind GANs.

[1]

3.1. Adversarialna funkcija izgube

Standardna funkcija izgube, ki jo uporabljajo GAN modeli je t.i. **adversarialna izguba**. Deluje na *min-max* principu, saj poiskuša generator vrednosti funkcije čimbolj zmanjšati, diskriminator pa zvečati. V sklopu SRResCycGAN modela lahko funkcijo opišemo z enačbo

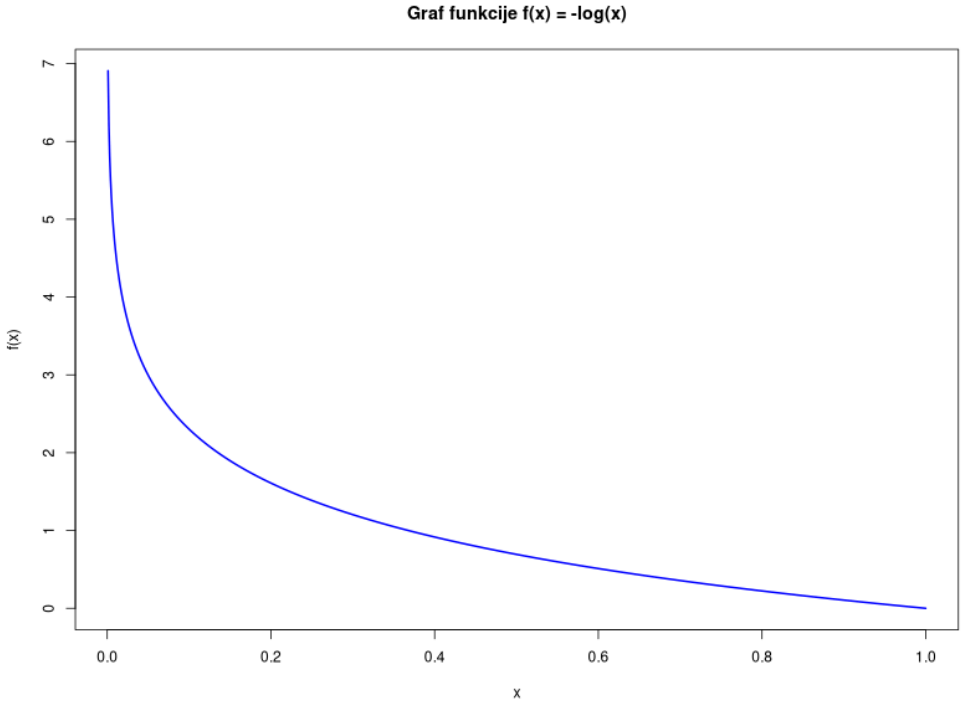
$$\min_{\theta_G} \max_{\theta_G} \mathbb{E}_{x_r}[\log\big(D_{\theta_G}(x_r)\big)] + \mathbb{E}_y[\log\big(1 - D_{\theta_G}\big(G_{\theta_G}(y)\big)\big)] \tag{1}$$

- x_r zaznamuje resnično LR sliko in $G_{\theta_G}(y)$ generirano neresnično HR sliko z vhodno LR sliko y. $\mathbb{E}_{x_r}$ je pričakovana vrednost od vseh resničnih slik in $D_{\theta_G}(x_r)$ diskriminatorjeva ocena verjetnosti, da je vhod x_r resničen.
- $\mathbb{E}_y$ je pričakovana vrednost vseh vhodnih LR slik y in posledično pričakovana vrednost od vseh generiranih slik $G_{\theta_G}(y)$.
- $D_{\theta_G}\big(G_{\theta_G}(y)\big)$ je diskriminatorjeva ocena verjetnosti, da je generirana slika resnična.
- θ_G in θ_G predstavljata uteži in biase generatorja G in diskriminatorja D.

Taka kot je deluje kot funkcija izgube za diskriminatorja, medtem ko generatorja zanima samo ocena diskriminatorja na njegovih neresničnih podatkih, zato je levi člen med učenjem generatorja izpuščen. Njegova funkcija izgube je torej

$$L_G = \frac{1}{N}\sum_{i=1}^{N} -\log(D_{\theta_G}\big(G_{\theta_G}(y_i)\big) \tag{2}$$

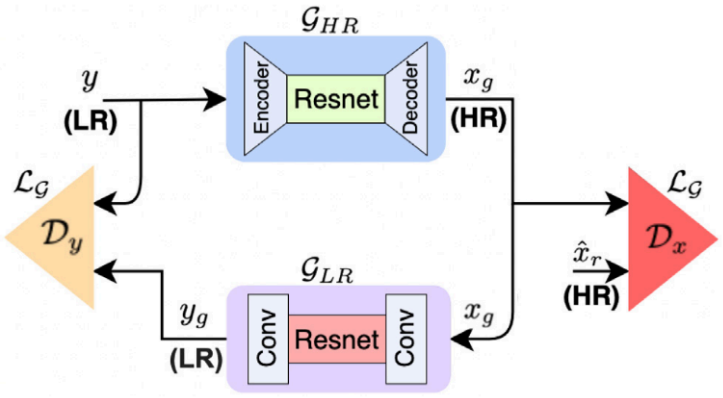
kjer je N število LR vzorcev za učenje (batch) in y_i vhodna LR slika. Bolj kot je ocena diskriminatorja blizu 1, bolj je vrednost logaritma (ocena verjetnosti) blizu 0.



4. SRResCycGAN

Večino SR GAN metod uporablja parne podatke LR in HR slik, kjer so LR slike pridobljene z bikubično interpolacijo HR slike. To prisili modele v to, da se naučijo izničiti rezultat tega procesa, kar pa ne odraža realnega sveta – **v realnem svetu je degradacija odvisna od veliko faktorjev**, ki na različne načine degradirajo sliko (šum, artefakti zaradi kompresije, zamegljenost zaradi premikanja, napake v lečah, itd.). Modeli zato postanejo zelo dobri v obračanju bikubične interpolacije, vendar se ne obnesejo dobro na pravih *umazanih* slikah, kjer je funkcija degradacije bolj kompleksna.

SRResCycGAN poiskuša rešiti ta problem tako, da se ne uči samo super resolucije (LR → HR), ampak tudi realistično degradacijo (HR → LR). Zaradi tega je njegova struktura bolj kompleksna – namesto dveh nevronskih mrež ima štiri, dva generatorja in dva diskriminatorja.



(a) Global structure, adapted from Umer and Micheloni (2020).

Figure 4: Struktura SRResCycGAN

[2]

Par G_{HR}, D_{HR} (generator in diskriminator za HR slike) skrbi za učenje super resolucije, medtem ko par G_{LR}, D_{LR} (analogno za LR slike) skrbi za učenje degradacije. [2]

4.1. CycleGAN

Struktura je osnovana, med drugimi, na CycleGAN modelu, katerega cilje je, da se nauči aproksimirati preslikavo med vhodno in izhodno sliko v primerih, ko pravi pari slik za treniranje niso na voljo (npr. pri *style transfer* metodah). Preslikavo G med domeno degradiranih LR slik X in domeno čistih slik visoke resolucije Y zapišemo kot $G : X \to Y$. Aproksimacija mora biti taka, da postane porazdelitev generiranih slik $G(X)$ nerazločljiva od porazdelitve Y. Ker je preslikava sama po sebi (z uporabo adversarialne izgube) slabo omejena, obstaja neskončno možnih preslikav. V ta namen je združena z inverzno preslikavo $F : Y \to X$.

4.1.1. Ciklična izguba

Uvedena je tudi nova funkcija izgube, ki se imenuje **cycle (cyclic) consistency loss** oziroma ciklična izguba, ki zagotavlja, da je $F(G(X)) \approx X$ in obratno. Intuitivno si lahko to predstavljamo kot "prevod stavka iz slovenščine v angleščino mora biti identičen prevodu stavka nazaj v slovenščino". Za $\forall x \in X$ in $\forall y \in Y$ so avtorji cikel preslikav $x \to G(x) \to F(G(x)) \approx x$ poimenovali **forward cycle consistency** ter $y \to F(y) \to G(F(y)) \approx y$ **backward cycle consistency**. Cikla skupaj tvorita omenjeno funkcijo izgube:

$$L_{\text{cyc}}(G, F) = \mathbb{E}_x[\|F(G(x)) - x\|_1] + \mathbb{E}_y[\|G(F(y)) - y\|_1] \tag{3}$$

V sklopu strukture SRResCycGAN (glej Figure 4) vzame model G_{HR} za vhod LR sliko in vrne generirano HR sliko, medtem ko G_{LR} vzame HR sliko in vrne generirano LR sliko. G_{HR} nadzoruje D_{HR}, ki ocenjuje kako resnična je vhodna HR slika, G_{LR} pa D_{LR}, ki ocenjuje kako resnična je vhodna LR slika. [3]

4.2. HR Generator

Struktura generatorja HR slik je prevzeta iz modela SRResCGAN [4]. Njegova struktura je prikazana spodaj. Deluje na principu "povečaj → izboljšaj", kar pomeni, da v začetnem delu poveča resolucijo vhodne LR slike na željeno ločljivost, skozi vmesni **residual network** (ali ResNet) izpostavi napake v sliki (šum, artefakti, predstavljene kot feature maps), v zadnjem delu pa se te napake od povečane slike odstranijo. Avtorji so poimenovali njegove dele "Encoder → ResNet → Decoder".

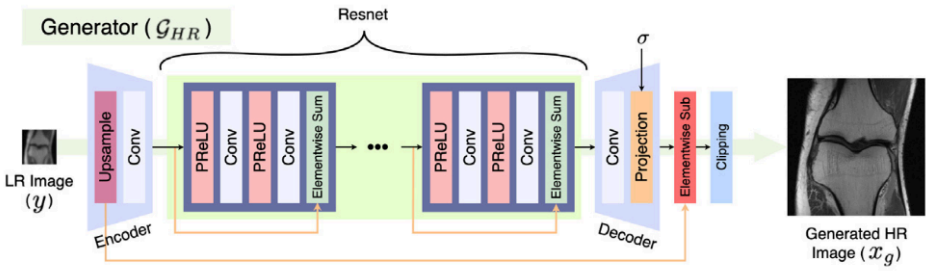


Figure 5: HR generator model

[1]

4.2.1. Encoder

Struktura se začne z **Encoder** delom, kjer prvi `Conv2DTranspose` poskrbi, da se vhodna slika velikosti `LR_SHAPE` (80, 80, 3) poveča na (320, 320, 3). Zadnja številka predstavlja število kanalov slike (RGB slike ⇒ 3 kanali). Drugi `Conv2D` sloj poskrbi, da se slika preslika v iz *image* v *feature space*.

Privzeta vrednost za filtre/kanale (`FILTERS`) slojev Encoder-ja in Decoder-ja je 64 ter 5 za velikost konvolucijskega jedra (`kernel_size`).

```python
def Generator_HR(input_shape=LR_SHAPE):
  lr_input = layers.Input(shape=input_shape)
  encoder_out = layers.Conv2DTranspose(
      filters=FILTERS,  # število kanalov izhoda; rgb slika ima 3, tu jih vrne 64)
      kernel_size=5,    # dimenzije jedra, ki bo procesiral sliko (5x5)
      strides=4,        # premik jedra (4 => 4x upsampling => 320)
      padding="same"    # doda ravno prav ničel na robove, da je končna dimenzija odvisna od strides
  )(lr_input)

  # služi kot vhod in izhod ResNet-a.
  x = layers.Conv2D(filters=FILTERS, kernel_size=5, padding="same")(encoder_out)
...
```

4.2.2. Residual Network

ResNet je arhitektura, ki poiskuša rešiti težave učenja globokih nevronskih mrež, npr. problem izginjajočih gradientov (vanishing gradients) in manjšanje natančnosti pri večjem številu slojev. To doseže preko t.i. **globokega residualnega učenja** - namesto, da bi vsak sloj aproksimiral preslikavo izhoda prejšnjega sloja v naslednjega, ResNet uči te sloje, da aproksimirajo t.i. **residualno preslikavo**.

Če je $H(x)$ želena preslikava, potem to pomeni, da ResNet uči svoje sloje, da aproksimirajo preslikavo $F(x) := H(x) - x$, $H(x) = F(x) + x$. Avtorji so postavili hipotezo, da je **lažje optimizirati residualno preslikavo** kot originalno. Če bi bila v skrajnem primeru optimalna rešitev identična preslikava (t.j. $H(x) = x$) naj bi bilo lažje potisniti $F(x)$ proti 0, kot pa se naučiti identične preslikave preko zaporednih slojev. V praksi je formulacija $H(x) - x$ realizirana preko t.i. **preskočnih povezav (skip connections)**. [5]

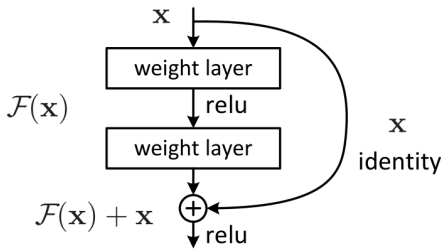


Figure 6: Residualni blok

To preprosto izgleda tako, da shranimo izhod prejšnjega sloja v spremenljivko (x na sliki) za poznejšo uporabo.

```python
...
global_skip = encoder_out
...
```

ResNet je sestavljen iz 5 residualnih blokov. Implementirani so z dvema t.i. **pre-activation** konvolucijskema slojema, kar pomeni, da aktivacijski sloj (v tem modelu PReLU) nastopi pred uteženimi sloji. Izkazalo se je, da to izboljša proces optimizacije. [6]

Za vsakim konvolucijskim slojem v residualnih blokih nastopi še normalizacijski sloj, ki prav tako rešuje problem izginjajočih gradientov in prekomerno prileganje na podatke (overfitting) tako da normalizira vrednosti. Namesto standardnega `BatchNormalization` (BN) je uporabljen `InstanceNormalization` (IN), saj BN izračuna povprečje in varianco celotnega batch-a, kar pomeni, da je normalizacija vsake slike odvisna od vseh. Izkazalo se je, da je to krivo za razne artefakte v generiranih slikah, zato priporočajo uporabo IN, ki normalizira vsak kanal neodvisno in tako ohranja značilnosti vsake slike. [1], [7]

```python
...
for _ in range(5):
    block_skip = x
    x = layers.PReLU(shared_axes=[1, 2])(block_skip)
    x = layers.Conv2D(filters=FILTERS, kernel_size=3, strides=1, padding="same", use_bias=False)(x)
    x = layers.GroupNormalization(groups=-1, axis=-1)(x)  # -1 za InstanceNormalization

    x = layers.PReLU(shared_axes=[1, 2])(x)
    x = layers.Conv2D(filters=FILTERS, kernel_size=3, strides=1, padding="same", use_bias=False)(x)
    x = layers.GroupNormalization(groups=-1, axis=-1)(x)
    x = layers.Add()([block_skip, x])
...
```

4.2.3. Decoder

Decoder vzame izhod residualnih blokov in ga pošlje še skozi dva konvolucijska sloja. Drugi sloj naj bi bil t.i. *projection layer* parametriziran s σ. Zaradi kompleksnosti sem implementacijo tega poljubnega sloja izpustil in uporabil preprost konvolucijski sloj.

```python
...
decoder_out = layers.Conv2D(filters=FILTERS, kernel_size=5, strides=1, padding="same")(x)
decoder_out = layers.Conv2D(filters=FILTERS, kernel_size=5, strides=1, padding="same")(decoder_out)
...
```

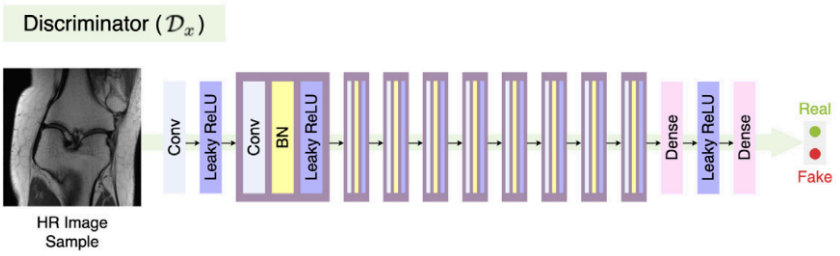
Končna *residualna slika* je po decoder-ju odšteta od prejšnje LR slike, shranjena kot `global_skip`. Zadnji konvolucijski sloj poskrbi, da ima izhodna slika 3 kanale, ReLU pa da so vrednosti omejene na interval (0, 255).

```python
...
subtracted = layers.Subtract()([global_skip, decoder_out])
model_out = layers.Conv2D(filters=3, kernel_size=3, strides=1, padding="same", use_bias=False)
(subtracted)
model_out = layers.ReLU(max_value=255)(model_out)  # vrednosti spravi na interval [0,255]

return Model(inputs=lr_input, outputs=model_out, name="G_HR")
```

4.3. HR Diskriminator

HR Generator je pod nadzorom HR Diskriminatorja, ki ocenjuje kako resnične so vhodne HR slike.



(b) Architecture

Figure 7: HR diskriminator

[1]

Struktura je posvojena od SRGAN modela. Ima 10 konvolucijskih slojev z izmenjajočimi velikostmi konvolucijskega jedra 3 in 4 ter naraščujočim številom filtrov (64 → 512). Za vsakim konvolucijskim slojem nastopita LeakyReLU in normalizacijski sloj. Ponovno je BN zamenjan z IN.

```python
def Discriminator_HR(input_shape=HR_SHAPE):
  hr_input = layers.Input(shape=input_shape)
  x = layers.Conv2D(filters=64, kernel_size=3, strides=1, padding="valid")(hr_input)
  x = layers.LeakyReLU()(x)

  KERNEL_SIZE = (4, 3)
  STRIDES = (2, 1)
  FILTERS = (
      64,
      128, 128,
      256, 256,
      512, 512, 512, 512
  )

  for i in range(9):
      kernel_size = KERNEL_SIZE[0] if i % 2 == 0 else KERNEL_SIZE[1]
      strides = STRIDES[0] if i % 2 == 0 else STRIDES[1]

      x = layers.Conv2D(filters=FILTERS[i], kernel_size=kernel_size, strides=strides, padding="valid")
(x)
      x = layers.GroupNormalization(groups=-1, axis=-1)(x)
      x = layers.LeakyReLU()(x)

  x = layers.Flatten()(x)
  x = layers.Dense(units=100)(x)
  x = layers.LeakyReLU()(x)
  model_out = layers.Dense(units=1)(x)

  return Model(inputs=hr_input, outputs=model_out, name="D_HR")
```

Izhod modela je v tem primeru logit vrednost, ki predstavlja preslikane vrednosti verjetnosti iz intervala $[0, 1]$ na $(-\infty, \infty)$.

4.4. LR Generator

LR Generator ima bolj preprosto, "Conv → ResNet → Conv", strukturo. Vsi konvolucijski sloji, razen zadnji, uporabljajo 64 filtrov. **Glava** s tremi konvolucijskimi sloji poskrbi za downsampling vhodne HR slike.

```python
def Generator_LR(input_shape=HR_SHAPE):
  hr_input = layers.Input(shape=input_shape)

  x = layers.Conv2D(filters=FILTERS, kernel_size=7, padding="same")(hr_input)
  x = layers.GroupNormalization(groups=-1, axis=-1)(x)
  x = layers.LeakyReLU(negative_slope=0.2)(x)

  for _ in range(2):  # downsample
      x = layers.Conv2D(filters=FILTERS, kernel_size=3, strides=2, padding="same")(x)
      x = layers.GroupNormalization(groups=-1, axis=-1)(x)
      x = layers.LeakyReLU(negative_slope=0.2)(x)
...
```

Vmesni **ResNet** je implementiran z residualnimi bloki "Conv → Norm → LReLU" s preskočnimi povezavami.

```python
...
for _ in range(6): # ResNet
    block_skip = x
    x = layers.Conv2D(filters=FILTERS, kernel_size=3, padding="same")(x)
    x = layers.GroupNormalization(groups=-1, axis=-1)(x)
    x = layers.LeakyReLU(negative_slope=0.2)(x)

    x = layers.Conv2D(filters=FILTERS, kernel_size=3, padding="same")(x)
    x = layers.GroupNormalization(groups=-1, axis=-1)(x)
    x = layers.Add()([block_skip, x])
    x = layers.LeakyReLU(negative_slope=0.2)(x)
...
```

Zadnji konvolucijski sloj v **repu** poskrbi, da ima izhodna slika 3 kanale in zadnji ReLU sloj, da so vrednosti na intervalu [0,255].

```python
...
for _ in range(2):
    x = layers.Conv2D(filters=FILTERS, kernel_size=3, padding="same")(x)
    x = layers.GroupNormalization(groups=-1, axis=-1)(x)
    x = layers.LeakyReLU(negative_slope=0.2)(x)

x = layers.Conv2D(filters=3, kernel_size=7, padding="same")(x)
model_out = layers.ReLU(max_value=255)(x)  # vrednosti spravi na interval [0,255]

return Model(inputs=hr_input, outputs=model_out, name="G_LR")
```

4.5. LR Diskriminator

Deluje podobno kot diskriminator v PatchGAN modelu, kjer model ne poiskuša oceniti celotne slike v kosu kot resnično, ampak oceni posamezne kose (patches). To doseže tako, da na koncu ne agregira vseh vrednosti v en `Flatten -> Dense` sloj, ampak vrne matriko logitov velikosti 20x20. Vsi konvolucijski sloji uporabljajo velikost jedra 5. Število filtrov se skozi sloje poveča iz 64 na 256, na koncu pa se združijo v enega. [8]

Prvi `Conv2D` sloj zmanjša vhodno LR sliko iz 80x80 na 40x40 zaradi `strides=2`. Drugi sloj ponovno prepolovi dimenziji na pol, zato je matrika logitov oblike 20x20.

Za vsakim konvolucijskim slojem nastopita normalizacijski in aktivacijski sloj, podobno kot pri ostali modelih.

```python
KERNEL_SIZE = 5
FILTERS = [64, 128, 256]

lr_input = layers.Input(shape=input_shape)

x = layers.Conv2D(filters=FILTERS[0], kernel_size=KERNEL_SIZE, strides=2, padding="same")(lr_input)
x = layers.GroupNormalization(groups=-1, axis=-1)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(filters=FILTERS[1], kernel_size=KERNEL_SIZE, strides=2, padding="same")(x)
x = layers.GroupNormalization(groups=-1, axis=-1)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(filters=FILTERS[2], kernel_size=KERNEL_SIZE, padding="same")(x)
x = layers.GroupNormalization(groups=-1, axis=-1)(x)
x = layers.LeakyReLU()(x)

model_out = layers.Conv2D(filters=1, kernel_size=KERNEL_SIZE, padding="same")(x)

return Model(inputs=lr_input, outputs=model_out, name="D_LR")
```

4.6. Funkcije izgube

Da naučimo model super resolucije slik, je potrebna uporaba več različnih funkcij izgube. Enačba, ki se bo skozi učenje optimizirala je

$$L_{G_{HR}} = L_P + L_G + L_{TV} + \lambda_{\text{content}} \cdot L_1 + \lambda_{\text{cyclic}} \cdot L_{CYC} \tag{4}$$

kjer je L_P izguba zaznave (perceptual loss), L_G adversarialna izguba (adversarial loss), L_{TV} izguba popolne variacije (total variation loss), L_1 izguba vsebine (content loss) in L_{CYC} ciklična izguba (cyclic loss). Naloga skalarjev λ_{content} in λ_{cyclic} je, da doda vsebinski in ciklični izgubi večjo težo, kar sili učenje generatorja, da postavi več pozornosti na ti dve izgubi.

4.6.1. Izguba zaznave

Fokusira se na "zaznavno podobnost" dveh slik. To doseže tako, da računa razdalje med posameznimi piksli v prostoru značilnosti (feature space) namesto v prostoru slik (image space). Definirana je kot L2 (evklidska) norma

$$L_P = \frac{1}{N} \sum_{i=1}^{N} \left\| \Phi\left(\hat{x}_{r_i}\right) - \Phi\left(x_{g_i}\right) \right\|_2^2 \tag{5}$$

med zemljevidoma značilnosti (feature maps) generirane HR slike x_{g_i} in njene resnične HR vrednosti $\hat{x}_{r_i}$. Zemljevida značilnosti, označena z $\Phi(I)$, kjer je I vhodna slika, sta pridobljena iz nekega vmesnega konvolucijskega sloja VGG19 klasifikatorja.

4.6.2. Izgube popolne variacije

MRI slike so ponavadi podvržene nizkemu razmerju med šumom in signalom (signal-to-noise ratio). To vpliva na učenje modela, saj obravnava pomanjkljivosti v slikah kot njihove značilnosti, kar vodi do popačenih HR slik. Izguba popolne variacije tako poiskuša zagotoviti, da je šum med učenjem pravilno obravnavan. Definirana je kot

$$L_{TV} = \frac{1}{N} \sum_{i=1}^{N} (\left\| \nabla_h G(y_i) - \nabla_h\left(\hat{x}_{r_i}\right) \right\|_1 + \left\| \nabla_v G(y_i) - \nabla_v\left(\hat{x}_{r_i}\right) \right\|_1) \tag{6}$$

kjer sta ∇_h in ∇_v vodoravna in navpična gradienta slike. Kot je razvidno na spodnji sliki, lahko z gradienti zaznamo prisotnost šuma.

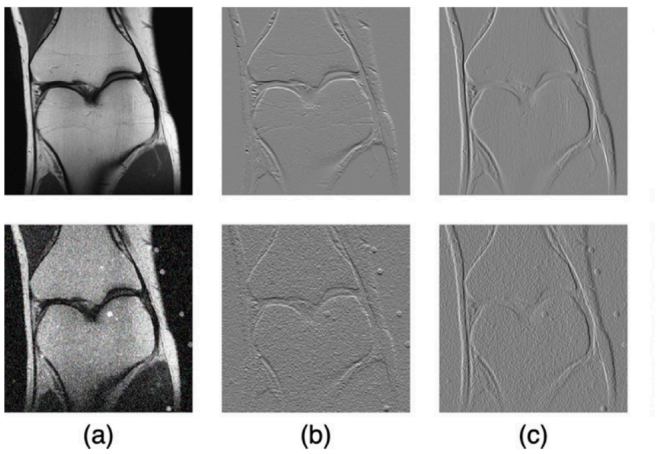


Figure 8: (a) originalna slika, (b) navpični gradienti, (c) vodoravni gradienti

[1]

4.6.3. Izguba vsebine

Preprosto izračuna povprečje razlik med piksli generirane in resnične HR slike. S tem na široko oceni rekonstrukcijo slike. Definirana je kot L1 norma

$$L_1 = \frac{1}{N}\sum_{i=1}^{n}\left\|G(y_i) - \hat{x}_{r_i}\right\|_1 \tag{7}$$

med generirano HR sliko $G(y_i)$ pridobljena iz vhodne LR slike y_i in resnično HR sliko $\hat{x}_{r_i}$.

Adversarialna in ciklična izguba sta bili opisani prej. Vse funkcije izgube so definirane znotraj slovarja `losses_dict` v modulu losses.py

4.7. Učenje

Celotni model je definiran v modulu SRResCycGAN.py. Za učenje sta za λ_{content} in λ_{cyclic} uporabljeni privzeti vrednosti 5 in 10. Za vse modele je uporabljen poljuben Adam optimizator [1], definiran na sledeč način:

```python
def adam_opt(lr=1e-4, b1=0.9, b2=0.999, decay_steps=1e4, decay_rate=0.5):
    """ Custom Adam optimizer s padajočim learning rate. """
    lr_schedule = ExponentialDecay(
        initial_learning_rate=lr,
        decay_steps=decay_steps,
        decay_rate=decay_rate,
        staircase=True  # da se lr spremeni na vsakih 10k korakov
    )
    return Adam(learning_rate=lr_schedule, beta_1=b1, beta_2=b2,
weight_decay=False)
```

Metoda `ExponentialDecay` poskrbi, da se stopnja učenja (na začetku 1×10^{-4}) vsakih 10'000 korakov prepolovi.

Vsak korak učenja, model pokliče metodo `train_step`, ki skrbi za izračun vrednosti funkcij izgub in gradientov ter posodabljanja uteži ob dani vhodni resnični LR in HR sliki.

5. Rezultati

Kljub temu, da je vse potrebno sprogramirano, žal nisem uspel pridobiti rezultatov zaradi konstantnih napak v Kaggle in Google Colab okolju, ko sem poiskušal pognati model z GPU ali TPU pospeševalniki. Model je možno pognati na CPU, vendar je, kljub mojem zoožanem izboru podatkov, za 1 epoch predviden čas izvajanja približno 52 ur (slika spodaj). 😭

```
[ ] epoch_steps_sub, val_steps_sub = calculate_steps(TRAIN_PATH, VAL_PATH, BATCH_SIZE)

Calculating slices for  /root/.cache/kagglehub/datasets/janpanjan/fastmri-knee/versions/2/train/train-sample ...
Total slices: 21423
Calculating slices for  /root/.cache/kagglehub/datasets/janpanjan/fastmri-knee/versions/2/val/val-sample ...
Total slices: 4620
epoch steps: 2677, val steps: 577

[ ] EPOCHS_SUB = 1

history_small = model_small.fit(x=train_sub, validation_data=val_sub, epochs=EPOCHS_SUB)
--NORMAL--
...     393/Unknown 22840s 58s/step - Adversarial loss: 12.8761 - Content loss: 0.1360 - Cyclic loss: 0.0849 - Percept
```

Figure 9: Google Colab, CPU runtime, predviden čas izvajanja za vseh 3254 korakov je približno 52 ur

Bibliography

[1] J. Guerreiro, P. Tomás, N. Garcia, and H. Aidos, "Super-resolution of magnetic resonance images using Generative Adversarial Networks," March 15, 2023. doi: https://doi.org/10.1016/j.compmedimag.2023.102280.

[2] R. Muhhamad Umer and C. Micheloni, "Deep Cyclic Generative Adversarial Residual Convolutional Networks for Real Image Super-Resolution," September 7, 2020. doi: https://doi.org/10.48550/arXiv.2009.03693.

[3] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks," August 24, 2020. doi: https://doi.org/10.48550/arXiv.1703.10593.

[4] R. Muhammad Umer, C. Micheloni, and G. Luca Foresti, "Deep Generative Adversarial Residual Convolutional Networks for Real-World Super-Resolution," May 3, 2020. doi: https://doi.org/10.48550/arXiv.2005.00953.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," December 10, 2015. doi: https://doi.org/10.48550/arXiv.1512.03385.

[6] M. Mortal, "Pre-activation in Neural Networks." [Online]. Available at: https://learningstracker.wordpress.com/2017/01/04/pre-activation-in-neural-networks/

[7] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Instance Normalization: The Missing Ingredient for Fast Stylization," November 6, 2017.

[8] U. Demir and G. Unal, "Patch-Based Image Inpainting with Generative Adversarial Networks." doi: https://doi.org/10.48550/arXiv.1803.07422.