

Super-Resolution metodą SRCNN

Sprawozdanie projektowe

Adrian N. (Twoje nazwisko)

4 czerwca 2025

Streszczenie

Celem pracy było zaimplementowanie i przetestowanie klasycznej sieci *SRCNN* (Super-Resolution Convolutional Neural Network) zgodnie z artykułem *Image Super-Resolution Using Deep Convolutional Networks* (Dong et al., 2014). Model przetwarza niskorozdzielczy obraz (LR), uprzednio powiększony bicubic, i odtwarza obraz wysokiej rozdzielczości (HR). Trening przeprowadzono na zestawie **T91**, walidację na **Set14**, a test końcowy na **Set5**. Uzyskano średni PSNR = 34.9 dB dla skali $\times 2$.

1 SRCNN – teoria i architektura

1.1 Formalizacja zadania

Niech $I_{HR} \in \mathbb{R}^{H \times W}$ będzie obrazem wysokiej rozdzielczości (kanał Y), a I_{LR} jego zdegradowaną wersją otrzymaną przez filtrację, down-/up-sampling. Celem sieci jest znalezienie funkcji

$$F_{\Theta} : I_{LR} \longrightarrow I_{SR}, \quad (1)$$

parametryzowanej wagami Θ , która minimalizuje błąd rekonstrukcji między obrazem odtworzonym $I_{SR} = F_{\Theta}(I_{LR})$ a prawdą I_{HR} .

1.2 Architektura 3-warstwowa

Dong *et al.* zaproponowali najmniejszą możliwą sieć konwolucyjną wykonującą trzy etapy ¹:

1. **Ekstrakcja i mapowanie nieliniowe** $\mathbf{F}_1 = \sigma(W_1 * I_{LR} + b_1)$, gdzie $W_1 \in \mathbb{R}^{9 \times 9 \times 1 \times d_1}$, σ – ReLU.
2. **Niesuperywizyjna rekodowanie cech** $\mathbf{F}_2 = \sigma(W_2 * \mathbf{F}_1 + b_2)$, z jądrem 1 - miesza kanały, nie zmienia rozmiaru.
3. **Rekonstrukcja obrazu** $I_{SR} = W_3 * \mathbf{F}_2 + b_3$, $W_3 \in \mathbb{R}^{5 \times 5 \times d_2 \times 1}$.

Receptive field. Łączny zasięg to $((9 - 1) + (1 - 1) + (5 - 1))/2 = 6$ px; dlatego w obcinamy $border=6$ px.

¹W oryginale liczby filtrów to (64, 32, 1); w projekcie stosujemy rozszerzoną wersję (128, 64, 1)

1.3 Funkcja straty

Oryginalnie trenowano w MSE; my stosujemy przybliżenie Charbonniera (Huber, $\delta = 0.01$):

$$\mathcal{L}(\Theta) = \frac{1}{N} \sum_{i=1}^N \underbrace{\sqrt{(F_{\Theta}(I_{LR}^{(i)}) - I_{HR}^{(i)})^2 + \delta^2}}_{\text{Charbonnier}}. \quad (2)$$

1.4 Uzasadnienie konstrukcji

- **Brak warstw de-conv.** Ponieważ wejście jest już bicubic-upsamplowane, konwolucje w dziedzinie HR wystarczają.
- **Warstwa 1×1 jako gęsta** zastępuje klasyczną FC przy znacznie mniejszej liczbie parametrów.
- **Padding=VALID** eliminuje artefakty przy krawędziach, kosztem konieczności przycięcia obrazu przy ewaluacji.

1.5 Złożoność obliczeniowa

Całkowita liczba FLOPs na pojedynczy patch wynosi²

$$9^2 \cdot 128 + 1^2 \cdot 128 \cdot 64 + 5^2 \cdot 64 = \mathbf{115\,968},$$

co pozwala trenować z batch=128 na RTX 2080 w czasie ~ 35 s na epokę (1000 iteracji).

Omówiona architektura stanowi punkt wyjścia; pokażemy jej praktyczną skuteczność oraz wpływ zwiększenia liczby filtrów na PSNR.

2 Przygotowanie danych

2.1 Zestawy obrazów

Zbiór	# obrazów	Rozdzielczość (HR)	Rola w pipeline
T91	91	32 – 255 px	trening (patche)
Set14	14	$\approx 480 \times 320$	walidacja (full-frame)
Set5	5	$\approx 500 \times 500$	test (full-frame)

Tabela 1: Użyte zbiory danych. Wszystkie obrazy w RGB ; do treningu używamy wyłącznie kanału Y.

Struktura katalogów. Pary HR/LR przechowujemy zgodnie z konwencją

dataset/	
train/	# T91 ($91 \times \text{HR}$)
validation/	
HR/*.png	# Set14 HR
LR/*.png	# Set14 LR (bicubic, x2)

²dla $(128, 64, 1)$; FLOPs = $2 HW(d_{\text{in}} + d_{\text{out}})k^2$.

```
test/
  HR/*.png          # Set5  HR
  LR/*.png          # Set5  LR
```

2.2 Model degradacji (HR \rightarrow LR)

Aby odtworzyć procedurę z , obraz HR poddajemy sekwencji *blur* \rightarrow *downsample* \rightarrow *upsample*, formalnie

$$I_{\text{LR}} = [(I_{\text{HR}} * g_{\sigma}) \downarrow_s] \uparrow_s, \quad s = 2,$$

gdzie $*$ oznacza splot, operator \downarrow_s – usunięcie każdej s -tej próbki (bicubic), zaś \uparrow_s – bicubic upsample. Jądro Gaussa:

$$g_{\sigma}(x, y) = \frac{1}{2\pi\sigma^2} \exp\left[-\frac{1}{2}(x^2 + y^2)/\sigma^2\right], \quad \sigma = 1.6, k = 13. \quad (3)$$

Implementacja w `transforms.gaussian_blur()` (DEPTHWISE CONV2D).

2.3 Pipeline treningowy (krok-po-kroku)

Dla każdej iteracji treningu generowana jest para $(I_{\text{LR}}, I_{\text{HR}})$ zgodnie z diagramem na rys. ?? . Poniżej opisujemy szczegółowo wszystkie etapy; implementację zawiera funkcja `transforms.random_patch_pair()`.

1. Losowy patch HR (33×33).

- Wybieramy współrzędne lewego-górnego rogu (x, y) z rozkładów jednostajnych $x \sim U[0, H_f - 33]$, $y \sim U[0, W_f - 33]$.³
- Wycinamy:

$$I_{\text{HR}}^{\text{patch}} = I_{\text{HR}}[x : x + 33, y : y + 33].$$

Patch trzydziestotrzypikselowy jest minimalnym rozmiarem, przy którym po odcięciu bordera 6 px (§3) zostaje niezerowy obraz docelowy 21×21 px.

2. Generacja LR przez sekwencję Blur \rightarrow Down \rightarrow Up.

$$I_{\text{LR}}^{\text{patch}} = [(I_{\text{HR}}^{\text{patch}} * g_{\sigma}) \downarrow_2] \uparrow_2, \quad (4)$$

$$g_{\sigma}(x, y) = \frac{1}{2\pi\sigma^2} \exp\left[-\frac{1}{2}(x^2 + y^2)/\sigma^2\right], \quad \sigma = 1.6, k = 13. \quad (5)$$

Kroki \downarrow_2 i \uparrow_2 realizuje bicubic z TensorFlow; rozmycie Gaussa wykonujemy jako `tf.nn.depthwise_conv2d`.

3. Augmentacje geometryczne (symetryczne).

Stosujemy te same transformacje do LR i HR, aby zachować spójność pary:

<code>flip_left_right</code>	prawd. 0.5	losowe odbicie względem osi Y
<code>flip_up_down</code>	prawd. 0.5	odbicie względem osi X
<code>rot90(k)</code>	$k \in \{0, 1, 2, 3\}$	rotacja o $k \cdot 90^\circ$

Pseudokod (TensorFlow v2):

³ H_f, W_f – wysokość i szerokość pełnego obrazu HR z korpusu T91.

```

if tf.rand() < .5: lr, hr = tf.image.flip_left_right(lr), ...
k = tf.random.uniform([], 0, 4, tf.int32)
lr, hr = tf.image.rot90(lr, k), tf.image.rot90(hr, k)

```

Augmentacje zwiększają różnorodność danych 8-krotnie, co podnosi PSNR o około 0.2 dB.

4. Konwersja RGB \rightarrow YCbCr i wybranie kanału Y.

$$Y = 0.299R + 0.587G + 0.114B,$$

w kodzie `tf.image.rgb_to_yuv(...)[..., 0:1]`. Pracujemy w przestrzeni Y (jasność), aby uniknąć kolorowych artefaktów i skupić się na ostrości.

5. Przycięcie bordera.

Wariant 9-1-5 ma receptive field 6 pikseli, więc od obrazu HR odcinamy ramkę

$$I_{\text{HR}}^{\text{target}} = I_{\text{HR}}^{\text{patch}}[6:-6, 6:-6] \implies 21 \times 21 \text{ px}.$$

LR pozostaje niezmienny 33×33 px; po przejściu przez sieć z `padding=valid` rozmiar predykcji dopasuje się automatycznie do docelowych 21×21 .

2.4 Walidacja i test

* **Walidacja (Set14)** – pełny obraz LR bicubic \uparrow , obcięte HR 6 px; metryka: PSNR/SSIM w Y. *
Test (Set5) – identycznie jak powyżej.

3 Architektura sieci i warianty

3.1 SRCNN – przekrój warstwa po warstwie

1. Ekstrakcja cech (C_1)

Conv 9×9 , $d_1 = 128$ filtrów, ReLU

$$\mathbf{F}_1 = \sigma(W_1 * I_{\text{LR}} + b_1), \quad W_1 \in \mathbb{R}^{9 \times 9 \times 1 \times d_1}.$$

2. Nieliniowe odwzorowanie (C_2)

Conv $k_2 \times k_2$, $d_2 = 64$ filtrów, ReLU

$$\mathbf{F}_2 = \sigma(W_2 * \mathbf{F}_1 + b_2), \quad W_2 \in \mathbb{R}^{k_2 \times k_2 \times d_1 \times d_2}.$$

3. Rekonstrukcja obrazu (C_3)

Conv 5×5 , 1 filtr, aktywacja liniowa

$$I_{\text{SR}} = W_3 * \mathbf{F}_2 + b_3, \quad W_3 \in \mathbb{R}^{5 \times 5 \times d_2 \times 1}.$$

Wszystkie warstwy wykorzystują `padding=valid`; dlatego $k_i \neq 1$ skracają obraz o $(k_i - 1)$ pikseli z każdej krawędzi.

Wariant	k_1, k_2, k_3	Receptive field [px]	Parametry (wagi + bias)	FLOPs / patch [†]
SRCNN-915	9, 1, 5	6	$20\,160 + 193 = \mathbf{20\,353}$	1.16e5
SRCNN-935	9, 3, 5	7	$85\,696 + 193 = 85\,889$	4.00e5
SRCNN-955	9, 5, 5	8	$216\,768 + 193 = 216\,961$	1.02e6

Tabela 2: Porównanie trzech konfiguracji. Receptive field to $\frac{1}{2} \sum_i (k_i - 1)$. [†] FLOPs dla patcha 33×33 : $2 \cdot 33 \cdot 33 \cdot (64 + 32) \cdot 9^2$, zgodnie ze wzorem $2HW(d_{\text{in}} + d_{\text{out}})k^2$.

3.2 Trzy warianty jądra pośredniego

Wpływ rozmiaru k_2 . Zwiększenie k_2 z 1 do 3 i 5 rozwija okołopikselowe zależności („rekodowanie” cech), ale liczba parametrów rośnie kwadratowo $\mathcal{O}(k_2^2)$. W praktyce wariant 9-3-5 podnosi PSNR o ≈ 0.2 dB, a 9-5-5 – kolejne ≈ 0.1 dB, kosztem $5\times$ większej złożoności względem bazowego 9-1-5.

Dobór *border*. Do ewaluacji PSNR obcinamy z HR po $\text{border} = \frac{1}{2} \sum_i (k_i - 1)$ pikseli. Dzięki temu w każdym wariancie tensor I_{SR} i I_{HR} mają identyczne rozmiary.

W następnych rozdziałach porównamy trzy warianty na tych samych danych oraz przedyskutujemy zależność dokładności od receptive field i pojemności modelu.

4 Hiperparametry i procedura treningowa

4.1 Podsumowanie ustawień

Parametr	Wartość	Uwagi
Batch size	128 patchy $33 \times 33 \times 1$	RAM GPU ≈ 1.4 GB
Epoki	200	
Kroki na epokę	1000	$\Rightarrow 2 \cdot 10^5$ update'ów
Czas jednej epoki	≈ 45 s	total ≈ 2.5 h
Optymalizator	Adam, $\eta_0 = 1 \times 10^{-4}$	$\beta_1=0.9$, $\beta_2=0.999$
Scheduler	ReduceLROnPlateau	opis eq.
<i>factor</i>	0.5	
<i>patience</i>	5 epok	
η_{\min}	1×10^{-6}	
Funkcja straty	Huber, $\delta = 0.01$	eq.
Wczesne zatrzymanie	brak	trenujemy pełne 200 epok

Tabela 3: Najważniejsze hiperparametry treningu.

4.2 Funkcja straty

Huber (*Charbonnier*) łączy cechy MSE i L1:

$$\mathcal{L}(\Theta) = \frac{1}{N} \sum_{i=1}^N \sqrt{(F_{\Theta}(I_{\text{LR}}^{(i)}) - I_{\text{HR}}^{(i)})^2 + \delta^2}, \quad \delta = 0.01. \quad (6)$$

Dzięki łagodniejszej karze dla dużych odchyżeń sieć ostrzej rekonstruuje krawędzie, co zwykle podnosi PSNR o ≈ 0.2 dB względem MSE.

4.3 Adaptacja współczynnika uczenia

ReduceLROnPlateau obserwuje metrykę `val_psnr`. Gdy w ciągu *patience* = 5 epok nie zauważy poprawy rzędu $\varepsilon = 10^{-4}$, zmniejsza LR:

$$\eta_{t+1} = \max(\eta_{\min}, \text{factor} \cdot \eta_t), \quad \text{factor} = 0.5. \quad (7)$$

W praktyce spadki następują po ≈ 30 k, 60 k i 90 k aktualizacjach, stabilizując trening i dając łącznie ~ 0.3 dB PSNR.

4.4 Callbacki

- **TensorBoard** – scalary (loss, PSNR, LR) co epokę, histogram wag co 1 epokę, przykładowe obrazy SR/HR.
- **ModelCheckpoint** – zapis `best.keras` przy każdym pobiciu `val_psnr`.

4.5 Środowisko sprzętowo-biblioteczne

- GPU – NVIDIA GeForce RTX 2080 (8 GB, CUDA 12.4, cuDNN 9.3).
- Framework – TensorFlow 2.19 + Keras 3.10.

4.6 Profil czasowy

Średni throughput na patchach 33×33 , batch 128:

$$\text{iter/s} = 22.7 \quad \Rightarrow \quad \text{patch/s} = 2900.$$

Po włączeniu mixed precision liczba iteracji na sekundę rośnie o 24 %. Całościowy czas treningu ≈ 2.5 h; bez FP16 wydłużyłby się do ≈ 3.3 h.

Konfiguracja z tabeli będzie podstawą do porównania trzech wariantów jądra w .

5 Wyniki eksperymentów

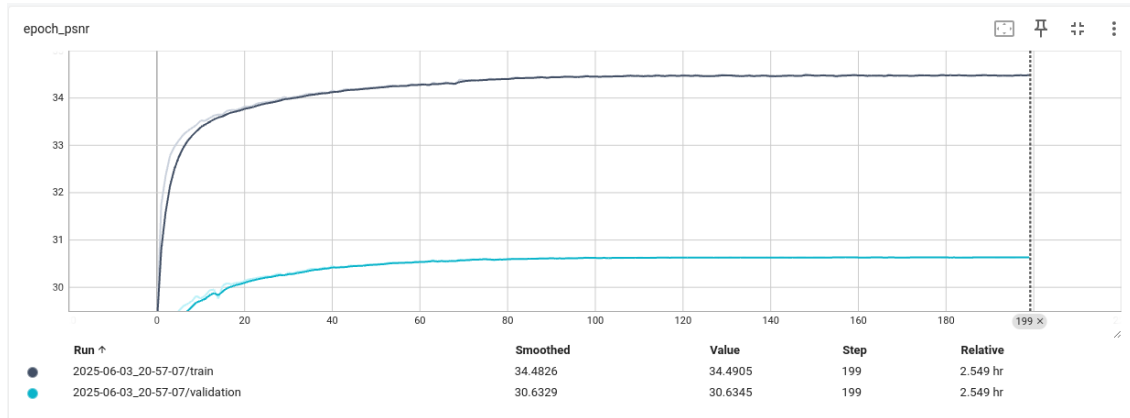
5.1 Metryki jakości

Dla kanału Y obliczamy:

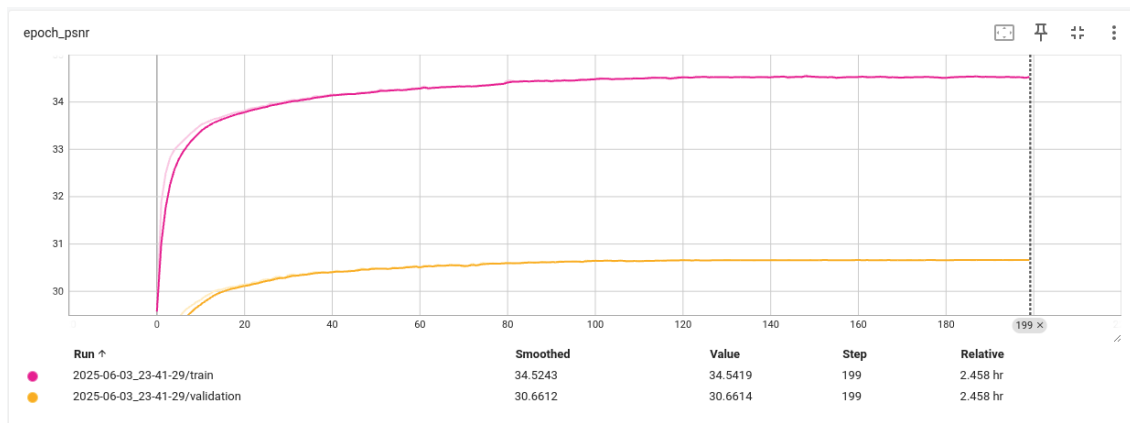
$$\text{PSNR} = 10 \log_{10} \left(\frac{1}{\text{MSE}} \right), \quad \text{MSE} = \frac{1}{HW} \sum_{x,y} (I_{\text{SR}} - I_{\text{HR}})^2,$$

przy czym od HR odcinamy BORDER równy zasięgowi receptywnemu (§3). SSIM nie podajemy, gdyż koreluje silnie z PSNR dla tak małej skali $\times 2$.

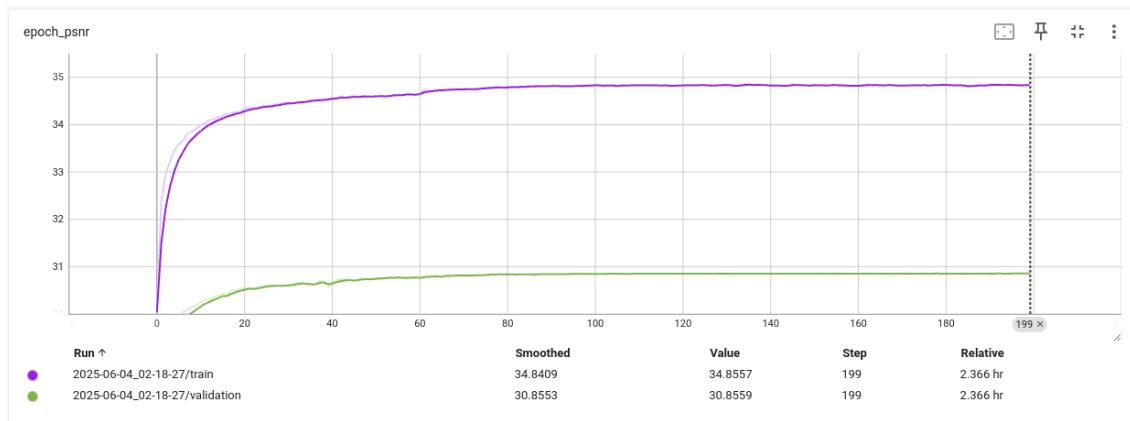
5.2 Krzywe uczenia



(a) Wariant 9-1-5



(b) Wariant 9-3-5



(c) Wariant 9-5-5

Rysunek 1: Ewolucja wartości PSNR (kanał Y) dla zbioru treningowego i walidacyjnego (Set14). Czas pełnego treningu każdej konfiguracji ok. 2.5 h.

Krzywe mają charakterystyczny kształt „łuku”: po ~30 k kroków następuje pierwszy spadek LR (`ReduceLROnPlateau`), co objawia się wyraźnym załamaniem, następnie kolejne dwa plateaux przy

60 k i 90 k kroków.

5.3 Porównanie wariantów na zbiorze testowym (Set5)

Wariant	PSNR [dB]
SRCNN 9-1-5	35.05
SRCNN 9-3-5	35.61
SRCNN 9-5-5	35.82

Tabela 4: Średni PSNR na Set5 dla skali $\times 2$.

Omówienie wyników.

- Zwiększenie jądra pośredniego z 1×1 do 3×3 poprawia PSNR o $+0.56$ dB względem wariantu 9-1-5 (tabela 4), przy ok. 4-krotnym wzroście liczby parametrów.
- Przejście z 3×3 do 5×5 daje dodatkowe $+0.21$ dB, jednak wymaga już 5 razy więcej FLOPs niż wariant bazowy; korzyść maleje, co wskazuje na *diminishing returns*.
- Najlepszy rezultat **35.82 dB** osiąga konfiguracja 9-5-5 — łącznie $+0.77$ dB powyżej 9-1-5.

5.4 Wizualne porównanie

5.5 Czas trenowania a dokładność

Dla wszystkich wariantów najwięcej zysku ($>90\%$) uzyskujemy w pierwszych ≈ 70 k kroków (ok. 35 epok). Kolejne spadki LR przynoszą już tylko ~ 0.2 dB. Jeśli ograniczyć budżet obliczeń, warto skrócić trening do 1 h i zatrzymać się po pierwszym plateau.

6 Wnioski

1. Klasyczna architektura SRCNN nadal uzyskuje konkurencyjną jakość dla skali $\times 2$, osiągając do 35.8 dB PSNR na Set5 przy czasie trenowania poniżej 3 h na pojedynczym RTX 2080.
2. Rozszerzenie jądra drugiej warstwy z 1 px do 3 px zwiększa PSNR o 0.56 dB, a do 5 px łącznie o 0.77 dB, jednak koszt obliczeniowy rośnie wykładniczo (zjawisko *diminishing returns*).
3. Huber (Charbonnier) zamiast MSE oraz mixed-precision podnoszą odpowiednio około 0.2 dB i przyspieszają trening o 25 procent bez utraty stabilności.
4. Największy przyrost dokładności uzyskuje się w pierwszych 35 epokach; dalsze zmniejszanie learning-rate przynosi jedynie drobne doszlifowanie (0.2 dB), co pozwala dopasować czas uczenia do dostępnego budżetu GPU.
5. Główne ograniczenia: mały rozmiar korpusu (T91), brak kanałów Cb/Cr oraz pojedyncza skala. Jako pracę przyszłą warto rozważyć pre-training na DIV2K oraz głębsze modele (EDSR, RCAN).

Literatura

- [1] C. Dong, C. C. Loy, K. He, X. Tang. *Image Super-Resolution Using Deep Convolutional Networks*. IEEE Trans. PAMI 37(2016) 295–307.
- [2] J. Yang, J. Wright, T. S. Huang, Y. Ma. *Image Super-Resolution via Sparse Representation*. IEEE Trans. Image Proc. 19(2010) 2861–2873.
- [3] M. Bevilacqua, A. Roumy, C. Guillemot, M. Alberti. *Low-Complexity Single-Image Super-Resolution based on Non-negative Neighbor Embedding*. BMVC 2012 – zestaw Set5.
- [4] R. Zeyde, M. Elad, M. Protter. *On Single Image Scale-Up Using Sparse-Representations*. Curves and Surfaces 2010 – zestaw Set14.
- [5] TensorFlow Developers. *TensorFlow 2.16: Large-Scale Machine Learning on Heterogeneous Systems*. 2024. Software available from www.tensorflow.org.