Painting Style Transfer

Adam Gregor, xgrego18
Zdeněk Jelínek, xjelin47

9. května 2021

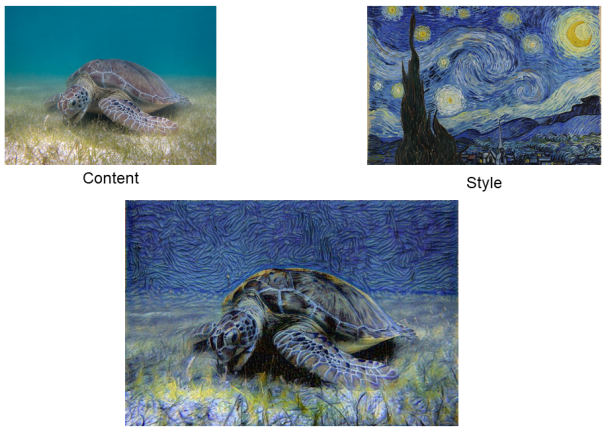
Úvod

Se zvyšováním výpočetního výkonu roste množina úloh, které byly dříve považovány za řešitelné výhradně člověkem, avšak u kterých se později ukázalo, že přeci jen existují metody (zpravidla založené na strojovém učení), pomocí nichž může stroj produkovat výsledky, které jsou při porovnání s těmi lidskými srovnatelné, či dokonce lepší. Příkladem takové úlohy může být hra Go, nebo malba obrazů.

Právě druhé uvedené disciplíně, v oboru strojového učení nazývané Painting Style Transfer, se bude tento text věnovat. Nejdříve uvedeme definici úlohy, poté rozebereme její implementaci v podání autorů textu a na závěr uvedeme experimenty a jejich vyhodnocení spojené s touto úlohou. **TOTO MOZNA SE TO ZMENI**.

Definice úlohy

Úloha Painting Style Transfer spočívá v kombinaci dvou vstupních obrázků do jednoho obrázku výstupního. Vstupy jsou obrázek *content*, na němž je obsah obrázku, který má být na výstupu a obrázek *style*, mající výtvarný styl, který má být do výstupu předán. Příklad vstupů a jejich kombinace je uveden na obrázku 1.



Obrázek 1: Příklad Paint Style Transfer. Vstupním obrázkem nesoucí obsah byla želva, stylem byla malba Starry Night. Výsledkem je obrázek želvy, jako by byla namalovaná ve stylu Starry Night.

Jedním z prvních článků, věnující se této úloze za použití neuronových sítí, a zároveň článkem, z nějž vycházeli autoři tohoto textu je A Neural Algorithm of Artistic Style od Leona A. Gatyse et al. [1] Gatys ve svém článku používal předtrénovanou síť VGG–19, v níž pozměnil max-pooling vrstvy za avg-pooling. Vstupem takovéto sítě pak byl obrázek, zpočátku bílý šum, který se během jednotlivých iterací aktualizoval na základě hodnot Loss funkce. Při tvorbě výstupního obrázku tedy nejsou aktualizovány parametry neuronové sítě, ale jednotlivé pixely toho obrázku a to tak, aby byla minimalizována hodnota Loss funkce. Označíme-li průběžně generovaný výstupní obrázek $\overrightarrow{x}$, *content* $\overrightarrow{p}$ a *style* $\overrightarrow{a}$, má Loss funkce L_{total} tvar

$$L_{total}(\overrightarrow{p}, \overrightarrow{a}, \overrightarrow{x}) = \alpha L_{content}(\overrightarrow{p}, \overrightarrow{x}) + \beta L_{style}(\overrightarrow{a}, \overrightarrow{x})$$

kde $L_{content}$ a L_{style} jsou funkce počítající, jak se generovaný obrázek $\overrightarrow{x}$ liší obsahem od obsahového obrázku *content* resp. stylem od stylového obrázku *style* a α a β jsou váhy těchto hodnot. V článku se používaly poměry $\alpha/\beta = 10^{-3}$ nebo 10^{-4}.

Dílčí Loss funkce $L_{content}$, měřící rozdíl obsahu mezi $\overrightarrow{x}$ a $\overrightarrow{p}$ porovnává aktivační mapy na l–té konvoluční vrstvě. Má-li tato vrstva N_l filtrů a aktivační mapy mají M_l pixelů, budou porovnávané matice $Q \in R^{N_l \times M_l}$. Jejich řádky budou představovat jednotlivé konvoluční filtry vrstvy a sloupce aktivační mapy těchto filtrů, převedené na vektor. Nyní, mámeli matice F a P obsahující tyto matice obrázků $\overrightarrow{x}$ a $\overrightarrow{p}$, je $L_{content}$ spočtena jako

$$L_{content} = \frac{1}{2} \sum_{i,j} (F_{ij} - P_{ij})^2$$

kde i je index filtru a j je index pixelu. V původním článku byla pro výpočet použita vrstva Conv4_2.

L_{style}, měřící rozdíl stylu mezi $\overrightarrow{x}$ a $\overrightarrow{a}$ se oproti $L_{content}$ nepočítá pouze nad jednou vrstvou ale nad několika. Je to z toho důvodu, že jinak hluboké vrstvy detekují jinak komplexní prvky stylu (např. mělčí vrstvy budou detekovat puze barvy, zatímco hlubší vrstvy mohou detekovat komplexnější entity jako tahy štětce) a pro kvalitní výsledek je podle Gatyse potřeba kombinace několika různě hlubokých vrstev. L_{style} je počítána pomocí gram matice. Ta počítá korelaci mezi filtry a míru jejich aktivace na jedné vrstvě. To si lze představit na příkladu, kdy máme dva filtry, jeden detekující modrou barvu a druhý detekující spirály. Pokud by tyto filtry měly vysokou korelaci, znamenalo by to, že bude každá spirála v obrazu modrá. Samotná gram matice je vypočtena opět pomocí matice Q. Pro

[1] https://arxiv.org/pdf/1508.06576.pdf

l–tou vrstvu tedy $Q \in R^{N_l \times M_l}$. Q je poté vynásobena sebou samou v transponované formě a tedy výsledná gram matice $G \in R^{N_l \times N_l}$.

$$G = Q \times Q^T$$

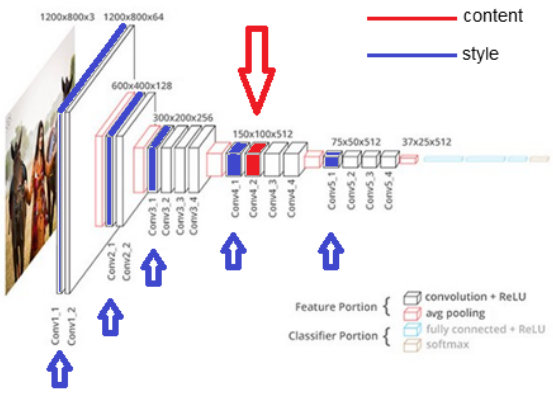
V l–té vrstvě je pak odchylka stylů $\overrightarrow{x}$ s gram maticí G^l a *style* s gram maticí A^l spočtena jako

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

Napříč všemi vrstvami je poté L_{style} spočtena jako

$$L_{syle} = \sum_{l=0}^{L} w_l E_l$$

kde E_l jsou dílčí chyby z různých vrstev a w_l jsou váhy těchto chyb. Gatys používal rovnoměrně rozdělené váhy, které v součtu dají hodnotu 1. V článku se používají chyby z vrstev Conv1_1, Conv2_1, Conv3_1, Conv4_1 a Conv5_1.
Na obrázku 2 je vyobrazena používaná síť VGG–19 spolu s místy, z nichž se počítá chyba.

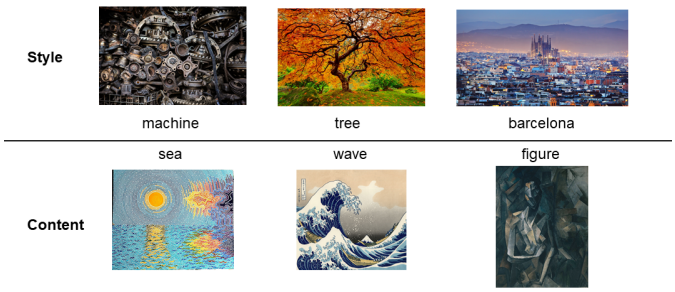


Obrázek 2: Znázornění použité architektury. Jsou zde také vyznačena místa, z nichž jsou počítány chyby. Původní obrázek převzat z https://towardsdatascience.com/art-with-ai-turning-photographs-into-artwork-with-neural-style-transfer-8144ece44bed

Implementace

Experimenty

V rámci experimentů s naší implementací jsme se rozhodli porovnat rychlost konvergence různých optimalizátorů, vyzkoušet vliv výběru odlišných vrstev pro výpočet L_{style} a **vyzkoušet vliv změny počátečního obrázku z bílého šumu na content obrázek**. Hodnoty α a β byly nastaveny na $\alpha = 10^{-1}$ $\beta = 10^6$ a tedy poměr $\alpha/\beta = 10^{-7}$. Tento poměr byl vybrán z toho důvodu, že produkoval L_{total} hodnoty blízko 0 a dle autorů na něm šly dobře pozorovat rozdíly v chování během jednotlivých experimentů. **Lež jak věž, Z. na to zapomnpěl, přepočítat, jestli bude čas**. V rámci experimentů používáme 3 *style* a 3 *content* obrázky, které jsou vyobrazeny na obrázku 3. Tyto obrázky pužíváme v *content-style* párech *machinery-sea*, *tree-wave* a *barcelona-figure*.



Obrázek 3: Obrázky použité v rámci experimentů. Jejich zdroje jsou uvedeny na gitu projektu.

Test optimalizátorů

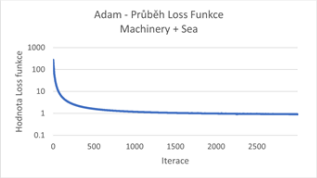
V rámci prvotní implementace, která byla vytvořena pomocí frameworku TensorFlow jsme zkoušeli téměř všechny její základní optimalizátory [2] z hlediska rychlosti konvergence a shledali jsme, že nejrychleji konverguje optimalizátor Adam. Když jsme však chtěli použít optimalizátor LBFGS, avšak v TensorFlow se nám jej nedařilo implementovat. To byl hlavní důvod migrace naší implementace do frameworku PyTorch.

V rámci experimentu jsme porovnávali optimalizátory Adam a LBFGS. Learning rate Adama byl nastaven na $0,01$, pro LBFGS byl nastaven na $0,8$. Výsledky experimentů zobrazují grafy na obrázku 4.

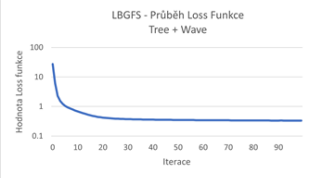
Z grafů lze vyčíst, že rychlost konvergence při použití optimalizátoru LBFGS je řádově rychlejší, než při použití Adama. Už po 20. iteraci dokázal LBFGS vygenerovat přijatelný výsledek, kdežto Adamovi trvalo vygenerování vizuálně zajímavějšího výsledku okolo 1000 iterací, viz **??**. Dalším zajímavým jevem je, že v případě optimalizátoru Adam občas docházelo k nenadálému nárustu Loss hodnoty. Pro tento úkaz nemají autoři vysvětlení.

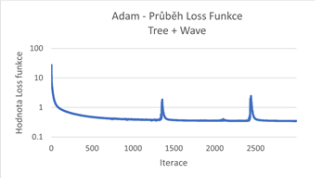
[2] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Machinery + Sea

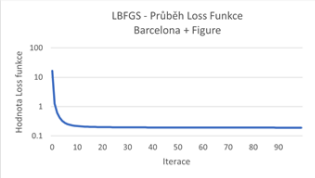


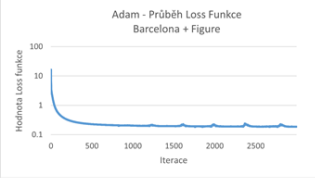
Tree + Wave





Barcelona + Figure





Obrázek 4: Grafy