

Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Przemysław Chlipała

Nr albumu: 1148609

**Metody wizualizacji optymalizacji
aktywacji neuronów sieci konwolucyjnych**

Praca magisterska
na kierunku Informatyka Stosowana

Praca wykonana pod kierunkiem
dr hab. Tomasz Kawalec
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Kraków 2019

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....
Kraków, dnia

.....
Podpis autora pracy

Oświadczenie kierującego praca

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....
Kraków, dnia

.....
Podpis kierującego pracą

Spis treści

1. Wstęp.....	5
2. Gęste sieci neuronowe.....	7
2.1. Algorytmy uczące się.....	7
2.2. Gęsta sieć neuronowa.....	7
2.3. Neuron gęstej sieci neuronowej	8
2.4. Propagacja w przód.....	9
2.5. Funkcja kosztu	11
2.6. Propagacja wstecz	12
3. Neuronowa sieć konwolucyjna.....	15
3.1. Wstęp	15
3.2. Opis elementów składających się na konwolucyjną sieć neuronową	15
3.2.1. Filtry i splot w konwolucyjnej sieci neuronowej.....	15
3.2.2. Warstwy łączące	17
3.3. Przykład sieci konwolucyjnej LeNet-5	18
3.3.1. Schemat sieci LeNet-5.....	18
3.3.2. Opis implementacji sieci LeNet-5	19
3.3.3. Implementacja modelu przy pomocy <i>Keras</i>	20
3.3.4. Wizualizacja neuronów warstw konwolucyjnych sieci LeNet-5.....	22
4. Wizualizacja przy pomocy warstw ukrytych sieci VGG-19.....	27
4.1. Model sieci VGG-19	27
4.1.1. Wizualizacja poprzez rekonstrukcję obrazu przy pomocy maksymalizacji aktywacji neuronu	28
4.1.2. Wizualizacja maksymalnej aktywacji danej klasy	30
4.2. Neural Style Transfer przy pomocy sieci VGG	37
5. Konwolucyjna sieć incepcja	41

5.1.	Architektura sieci incepcja.....	41
5.1.1.	Wstęp.....	41
5.1.2.	Blok sieci incepcja.....	41
5.1.3.	Schemat sieci.....	42
5.2.	<i>DeepDream</i> i wizualizacje warstw ukrytych GoogleLeNet.....	44
5.2.1.	Opis i zasada działania	44
5.2.2.	Wizualizacja warstw ukrytych modelu GoogleLeNet.....	44
5.2.3.	Wizualizacje uzyskane przy pomocy <i>DeepDream</i>	45
6.	Podsumowanie	51

1. Wstęp

Rozwój nauki oraz stały wzrost mocy obliczeniowej komputerów pozwolił na rzeczy, które wydawały się niemożliwe jeszcze dekadę temu. Chociaż pierwsze ważne dla sieci neuronowych koncepty sięgają lat pięćdziesiątych, a kluczowe kwestie takie jak algorytm propagacji wstecznej lat siedemdziesiątych, to dopiero w ciągu ostatnich dziesięciu lat nastąpiła popularyzacja tej technologii na nie spotykaną dotąd skalę.

Przy pomocy sieci neuronowych zaczęto budować systemy przewidujące ceny nieruchomości, indeksy giełdowe czy to, jakie jest prawdopodobieństwo tego, że ktoś zapadnie na przewlekłą chorobę. Co więcej, wiele firm zbudowało na tych modelach ogromny kapitał. Przykładowo, sieci posłużyły im do predykcji zainteresowania danym produktem. W 2014 roku Internet obiegła informacja, że firma Amazon przewiduje to, co ich klient zamierza kupić zanim jeszcze złoży zamówienie. Między innymi, dzięki temu udało im się skrócić czas oczekiwania na zamówienie do 48 godzin.

Ale na przewidywaniach się nie skończyło. Rekurencyjne sieci neuronowe rozpoznają słowa kluczowe i sprawiają, że możemy wydawać polecenia naszym urządzeniom. Co więcej, z ich pomocą tłumaczenia prosto z internetowego translatora brzmią z dnia na dzień coraz bardziej naturalnie. Automatyczne transkrypcje są coraz szerzej dostępne pod filmami udostępnionymi w Internecie a ich poprawność rośnie z dnia na dzień.

Komputery zaczęły rozpoznawać przedmioty zapisane na wideo i zdjęciach. Ta futurystyczna technologia wspomaga rzeczy tak prozaiczne jak kontrola jakości czipów na liniach produkcyjnych. Automatyczne sortownie odpadów mieszanych przynoszą nieoceniony zysk dla środowiska i realny, liczony w dolarach zysk dla ich właścicieli.

W mojej pracy skupię się na sieciach z dziedziny przetwarzania obrazu. Po wprowadzeniu w podstawowe wiadomości z dziedziny głębokich sieci neuronowych, zaprezentuję budowę prostego modelu takiej sieci, na przykładzie neuronowej sieci konwolucyjnej (z angielskiego Convolutional Neural Networks – CNN) LeNet-5. Przy jej pomocy, wpierw sklasyfikuję kilka prostych symboli, a następnie, zwizualizuję, czego tak naprawdę nauczył się mój automat.

Przybliżę również architekturę sieci VGG oraz bardziej złożone wizualizacje na podstawie

jej warstw. Uzyskane zostaną one poprzez maksymalizację mediany wygenerowanych aktywacji dla obrazu wyjściowego.

Opiszę zastosowanie sieci VGG w generowaniu artystycznych obrazów przy pomocy neuro-nowego transferu stylu (ang. *Neural Style Transfer*). Zaprezentuję pokrótce, również najnowsze implementacje tej metody, nie oparte o sieci VGG, a dające niewspółmiernie lepsze rezultaty.

Pod koniec mojej pracy, wprowadzę koncepcję sieci typu incepcja (ang. *Inception network*) i na jej podstawie wytlumaczę na czym polega *DeepDream* zaprezentowany w 2015 roku przez inżynierów z Google.

Mam nadzieję, że każdy po zapoznaniu się z treścią tej pracy, będzie lepiej zaznajomiony z tematem głębokich sieci neuronowych służących do przetwarzania obrazu.

2. Gęste sieci neuronowe

2.1. Algorytmy uczące się

W przypadku każdego wariantu sieci neuronowej mówimy o modelu luźno inspirowanym biologicznym odpowiednikiem. Jego zadaniem jest rozwiązywanie pewnej zadanej klasy problemu T . Proces ten nazywamy nauką lub treningiem. Nauka została zdefiniowana przez prof. Toma Mitchella w następujący sposób: „*Program komputerowy uczy się na podstawie doświadczenia E, w związku z pewną klasą problemów T przy uwzględnieniu miary skuteczności P, jeżeli jego sprawność w wykonywaniu zadań T, wyrażona przy pomocy P rośnie razem z doświadczeniem E.*”

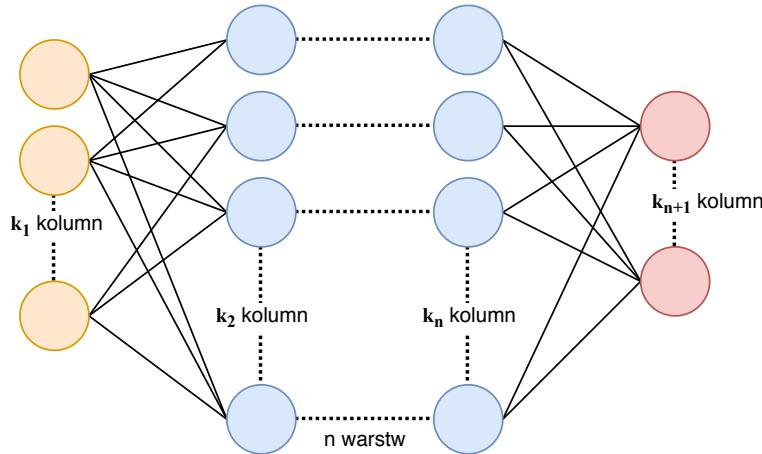
Proces treningu w przypadku modeli przytoczonych w tej pracy będzie polegał na ustaleniu wag przy poszczególnych węzłach sieci neuronowej, dalej nazwanych neuronami. Składać się on będzie z dwóch części: propagacji w przód oraz propagacji wstecznej.

Praca traktuje o zagadnieniach z dziedziny uczenia z nauczycielem (ang. *Supervised Learning*). W związku z tym, podczas treningu, propagacja w przód będzie polegała na obliczeniu wartości poprzez sieć dla danych uprzednio sklasyfikowanych i oznakowanych przy pomocy innego automatu lub człowieka. Uzyskany wynik jest początkowo losowy i zostaje użyty do obliczenia wartości funkcji kosztu, opisanych w rozdziale 2.5. Na podstawie funkcji kosztu, wagi sieci zostaną skorygowane w procesie propagacji wstecznej.

2.2. Gęsta sieć neuronowa

Podstawową implementacją sieci neuronowej jest tzw. sieć w pełni połączona (ang. *Fully Connected Network*). Składa się z warstw: wejściowej, jednej lub więcej warstw ukrytych i warstwy wyjściowej.

W przypadku, gdy model zawiera stosunkowo dużą liczbę warstw ukrytych, na schemacie oznaczonych jako n , mówimy o głębokich sieciach neuronowych. Ten wariant sieci neuronowej charakteryzuje się tym, że każdy węzeł jej warstwy jest połączony z każdym. Warstwa

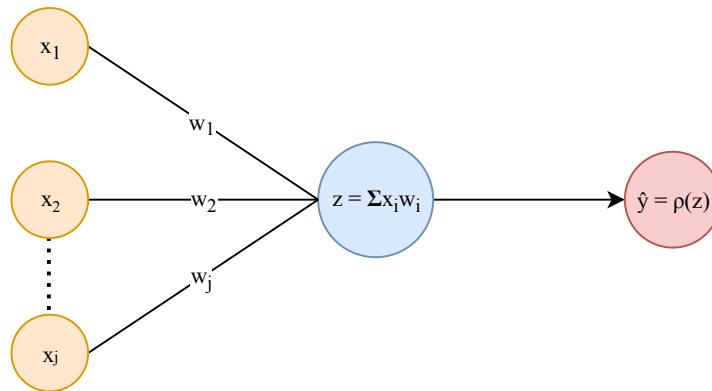


Rysunek 2.1: Schemat w pełni połączonej sieci neuronowej.

wejściowa, oznaczona kolorem żółtym, doprowadza dane wejściowe do warstw ukrytych oznaczonych kolorem niebieskim. Warstwa wyjściowa oznaczona jest kolorem czerwonym.

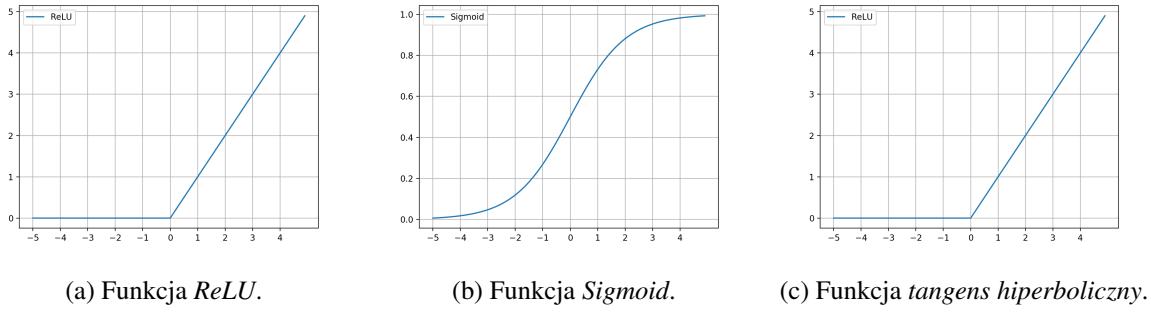
2.3. Neuron gęstej sieci neuronowej

Pojedynczy neuron można przedstawić następującym schematem 2.2.



Rysunek 2.2: Schemat w pełni połączonej sieci neuronowej.

Wartości a_1 do a_j reprezentują wartości z warstwy sieci neuronowej, a w szczególności wartości początkowe $x_1 \dots x_j$. Każda z nich jest przemnożona przez odpowiadającą jej wagę w i zsumowana. Następnie, tak uzyskaną wartość używamy do wyliczenia ρ nazwanego funkcją aktywacji neuronu. Zadaniem ρ jest wprowadzenie elementu nieliniowości. Mniej formalnie, ma na celu ustalenie czy neuron powinien zostać aktywowany czy też nie. Najpopularniejsze funkcje aktywacji przedstawia rysunek 2.3.



Rysunek 2.3: Wybrane funkcje aktywacji.

2.4. Propagacja w przód

By ustalić wartość neuronu, posługujemy się następującym wzorami:

$$z_n^{[l]} = \sum_{i=1}^j w_{ni}^{[l]} a_i^{[l-1]} + b_n^{[l]},$$

$$a_n^{[l]} = \rho(z_n^{[l]}),$$

Gdzie:

- ρ to funkcja aktywacji,
- (n) oznacza numer neuronu w warstwie,
- $^{[l]}$ to numer warstwy,
- b to tak zwany bias. Jego wartość jest ustalana razem z wagami w procesie propagacji wstecznej,
- $w_{ni}^{[l]}$ jest to i -ta waga neuronu numer n , warstwy l ,
- $a_i^{[l-1]}$ jest to wartość i -tego neuronu warstwy $[l - 1]$, w szczególnym przypadku może być to jedna z wartości wejściowych x .

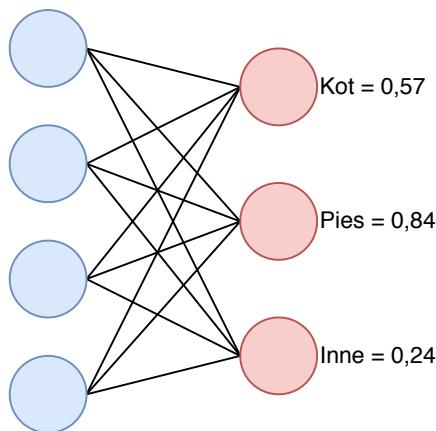
Używając tych wzorów, możemy obliczać wartości wag kolejnych warstw sieci aż do warstwy ostatniej. Tam, w zależności od postawionego problemu, możemy wyliczyć kolejne aktywacje przy pomocy ρ (tylko w odróżnieniu od warstw ukrytych nie stosuje się tam $ReLU$, tylko np. $sigmoid$ lub możemy użyć funkcji $softmax$):

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Jest to funkcja, która przyjmuje y_n elementowy wektor składający się z liczb rzeczywistych i przeprowadza go w rozkład prawdopodobieństwa składającego się z y_n elementów. Przydatną jej cechą jest to, że zachowuje proporcjonalny udział wartości w wynikowym prawdopodobieństwie – tj. elementy o wyższej wartości mają przypisane wyższe wartości prawdopodobieństwa.

Na przykładzie klasyfikacji, by uzyskać wynik, który można zinterpretować, należy posłużyć się następującym sposobem postępowania. Niech przedostatnia warstwa $a^{[L-1]}$ sieci posiada n węzłów, których liczba jest równa liczbie klas rzeczy, które chcemy rozpoznawać. Przymijmy również, że dana rzecz nie może należeć do dwóch klas równocześnie.

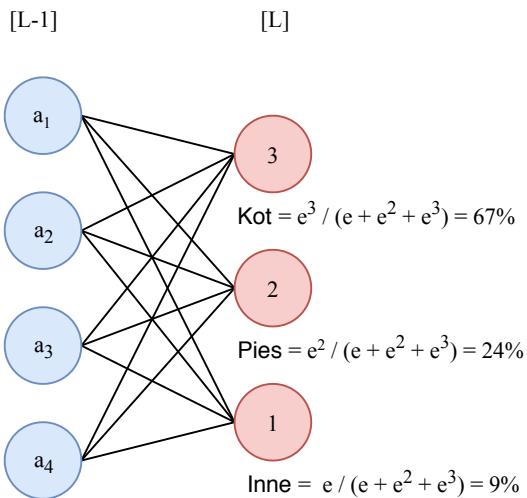
W przypadku aktywacji przy pomocy sigmoid uzyskujemy algorytm nazywany „jeden kontra każdy” (ang. *one vs all*). Każdy z węzłów $a_i^{[L-1]}$ jest wejściem dla ρ . Po wyliczeniu wartości, przyjmujemy próg z przedziału $[0, 1]$, by uzyskać wynik dla każdego neuronu z osobna.



Rysunek 2.4: Przykładowy wynik *1 vs all* dla trzech klas ze zbioru $\{\text{Kot}, \text{Pies}, \text{Inne}\}$

By zobrazować działanie algorytmu posłużę się przykładem. Założmy, że zbudowano klasyfikator, mający na celu przypisanie danej postaci klasy ze zbioru kot, pies, inne. Każdą aktywację z warstwy $L - 1$ przeprowadzamy z osobna, przy pomocy *sigmoid* w wartość z przedziału od 0 do 1. Następnie, przyjmujemy próg, powyżej którego mówimy, że sieć rozpoznała daną klasę. Przykładowo, w przypadku z rysunku 2.4 przyjmując próg równy 0.5 mamy dwa możliwe wyniki. By wyeliminować jeden z nich możemy zwiększyć próg do 0.6 lub wybrać neuron z wyższą wartością. W obu przypadkach będzie to neuron oznaczający to, że klasyfikator rozpoznał psa.

Istnieją inne metody aktywacji warstw wyjściowych. Każda z nich, która zostanie użyta w tej pracy będzie trzymała się podobnego schematu. Wynikiem sieci gęsto połączonej będzie wektor zer i jedynek, w którym 1 ustawione na odpowiednim miejscu w rzędzie będzie oznaczało, że dany przedmiot został rozpoznany. W przypadku rozrysowanym na rysunku 2.5 będzie to wektor [100].



Rysunek 2.5: Przykładowy wynik z użyciem *softmax*.

2.5. Funkcja kosztu

W przypadku problemów poruszanych w tej pracy, funkcję kosztu należy rozumieć jako przyporządkowanie wynikom uzyskanym przy pomocy sieci neuronowej wraz ze znanyimi poprawnymi odpowiedziami, liczbami reprezentującymi „koszt”. Wartość kosztu należy interpretować jako skuteczność w odgadywaniu poprawnych wyników w taki sposób, że im niższy koszt tym wyższa poprawność predykcji. Można powiedzieć, że sieć neuronowa próbuje znaleźć najwierszjsze przybliżenie oryginalnej, nieznanej funkcji nazywanej hipotezą, a celem funkcji kosztu jest modelowanie różnicę między przybliżeniem a funkcją, której próbujemy nauczyć sieć.

Istnieje wiele różnych funkcji kosztu. Opiszę tutaj tylko funkcję *cross entropy loss*, która jest często wybierana przy treningu sieci, które na ostatniej warstwie używają *softmax*.

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

gdzie:

- w – wagi sieci neuronowej,
- m – liczba klasyfikowanych przykładów,
- y_i – wartość oczekiwana dla przykładu numer i
- \hat{y}_i – wartość uzyskana przy pomocy sieci dla przykładu numer i .

Jest to suma błędów pojedynczych predykcji dla wszystkich przykładów dostępnych w danym zestawie danych treningowych. Zadaniem treningu, będzie znalezienie jej minimum dla

danego problemu. By udało się to sprawnie zrobić, będę korzystał z faktu, że ta i inne funkcje koszta, którymi będę się posługiwał są różniczkowalne i ciągłe.

2.6. Propagacja wstecz

Jest to modyfikacja wag sieci neuronowej w z uwzględnieniem funkcji koszta $J(w)$. Popularnym sposobem jest metoda gradientu prostego (częściej spotykana w wariacji z tzw. pędem, z angielskiego *Gradient descent with momentum* lub dalsze jej modyfikacje jak np. algorytm *Adaptive moment estimation*, znany jako *Adam*). W swej najprostszej wersji można sprowadzić go do następujących kroków:

1. Wybierz punkt startowy w_0 .
2. $w_{k+1} = w_k - \alpha \nabla f(w_k)$.
3. Jeżeli $\|w_{k+1} - w_k\| \geq \varepsilon$ idź do 2.
4. Koniec.

gdzie:

- α – współczynnik uczenia: zazwyczaj niewielka ($\alpha < 0.01$) dodatnia liczba rzeczywista. Podczas treningu sieci neuronowych często $\alpha = \text{const}$, choć sam algorytm gradientu prostego posiada wariację ze zmienną wartością α .
- $\nabla f(w_k)$ – jest to gradient funkcji reprezentującej funkcję przybliżającą hipotezę.
- w – wagi sieci,
- ε – niewielka wartość będąca kryterium stopu dla algorytmu. Wykorzystuje fakt, że różnice w wartości wag będą maleć wraz ze zbliżaniem się do minimum funkcji $J(w)$.

Krok pierwszy, czyli wybranie punktu startowego jest rozwiązywany przy pomocy losowej inicjalizacji wag w . Często stosuje się tu różne heurystyki dostosowane do użytej funkcji aktywacji (dla tanh jest to przykładowo inicjalizacja *Xaviera*).

Kłopotliwym zagadnieniem może wydawać się wyliczenie gradientu $\nabla f(w)$. Nie posiadając formalnej definicji hipotezy, tylko jej aproksymację w postaci sieci neuronowej musimy, posłużyć się funkcją koszta $J(w)$ i zależnością:

gdy:

$$J(z) = J(a(z))$$

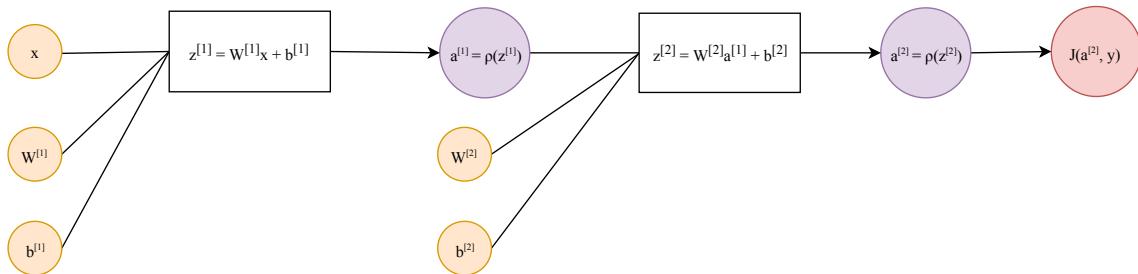
to:

$$\frac{dJ}{dz} = \frac{dJ}{da} \frac{da}{dz}$$

Posługując się nimi, możemy wyjść od wzoru funkcji koszta $J(w)$ i na tej podstawie aktualizować wartość wag ostatniej warstwy. Co więcej, możemy przy pomocy nowo wyliczonych wag, wyliczyć nowe wartości wag warstwy przedostatniej i tak niejako licząc warstwa po warstwie, aktualizować wszystkie wagi sieci neuronowej.

Niestety konkretne wzory służące do aktualizacji wag muszą być wyprowadzone dla każdej z funkcji koszta z osobna. Całe szesście biblioteki do uczenia maszynowego mają zaimplementowane większość z nich w taki sposób, że użytkownik musi tylko zaimplementować propagację w przód i wybrać funkcję koszta. Nie mniej jednak, by zobrazować ten proces wyprowadzę wzory potrzebne dla propagacji wstecznej dla dwuwarstwowej sieci neuronowej. Propagację na dalsze warstwy będą tylko ponownym wykorzystaniem tych samych wzorów w zastosowaniu do kolejnych wag.

Schemat sieci neuronowej wykorzystanej do obliczeń znajduje się na rysunku 2.6.



Rysunek 2.6: Przykładowy schemat sieci neuronowej, przy liczeniu propagacji wstecznej.

Dla pojedynczego przykładu ze zbioru treningowego:

$$J(w) = y \log a(w)^{[2]} + (1 - y) \log (1 - a(w)^{[2]})$$

oraz przyjmując sigmoid za funkcję aktywacji:

$$\rho = \frac{e^z}{e^z + 1},$$

wtedy:

$$1. \frac{dJ}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}},$$

$$2. \frac{da^{[2]}}{dz^{[2]}} = \frac{d\rho^{[2]}}{dz} = \frac{e^{z^{[2]}}}{\left(1 + e^{z^{[2]}}\right)^2} = \frac{e^{-z^{[2]}}}{\left(1 + e^{-z^{[2]}}\right)^2} = \frac{1}{1 + e^{-z^{[2]}}} \frac{1 + \left(e^{-z^{[2]}}\right) - 1}{\left(1 + e^{-z^{[2]}}\right)} = \frac{1}{1 + e^{-z^{[2]}}} (1 - \frac{1}{1 + e^{-z^{[2]}}}) = a^{[2]}(1 - a^{[2]}),$$

$$3. \frac{dJ}{dz^{[2]}} = \frac{dJ}{da^{[2]}} \frac{da^{[2]}}{dz^{[2]}} = \left[-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \right] a^{[2]} (1 - a^{[2]}) = - \left[\frac{y(1-a^{[2]}) + a^{[2]}(1-y)}{a^{[2]}(1-a^{[2]})} \right] a^{[2]} (1 - a^{[2]}) = a^{[2]} - y.$$

Dysponując $\frac{dJ}{dz^{[2]}}$, $a^{[2]}$ i $a^{[1]}$ (z fazy propagacji w przód) możemy zaktualizować $w^{[2]}$ i $b^{[2]}$,

$$4. \frac{dJ}{dw^{[2]}} = \frac{dJ}{dz^{[2]}} \frac{dz^{[2]}}{dw^{[2]}} = (a^{[2]} - y)a^{[1]},$$

$$5. \frac{dJ}{db^{[2]}} = \frac{dJ}{dz^{[2]}} \frac{dz^{[2]}}{db^{[2]}} = (a^{[2]} - y).$$

Powyzsze obliczenia pokazują, w jaki sposób obliczyć nową wartość wag między funkcją kosztu a ostatnią warstwą sieci. Kolejne kroki pokażą, w jaki sposób obliczyć nową wartość wag pomiędzy dwiema warstwami (można te kroki uogólniać na dowolną liczbę warstw).

$$6. \frac{dJ}{da^{[1]}} = \frac{dJ}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}} = (a^{[2]} - y)w^{[2]},$$

$$7. \frac{dJ}{dz^{[1]}} = \frac{dJ}{da^{[1]}} \frac{da^{[1]}}{dz^{[1]}} = w^{[2]}(a^{[2]} - y)a^{[1]}(1 - a^{[1]}).$$

$(\frac{da^{[1]}}{dz^{[1]}})$ obliczone analogicznie jak w kroku numer. 2),

Stąd wiadomo jak obliczyć wagę $w^{[1]}$ i $b^{[1]}$,

$$8. \frac{dJ}{dw^{[1]}} = \frac{dJ}{dz^{[1]}} \frac{dz^{[1]}}{dw^{[1]}} = w^{[2]}(a^{[2]} - y)a^{[1]}(1 - a^{[1]})x,$$

$$9. \frac{dJ}{db^{[1]}} = \frac{dJ}{dz^{[1]}} \frac{dz^{[1]}}{dw^{[1]}} = w^{[2]}(a^{[2]} - y)a^{[1]}(1 - a^{[1]}).$$

Po takim cyklu, należy wybrać kolejny przykład z zestawu treningowego, przeliczyć propagację w przód, funkcję kosztu a następnie znów propagację w tył, aż kryterium stabilności metody gradientu prostego zostanie spełnione.

3. Neuronowa sieć konwolucyjna

3.1. Wstęp

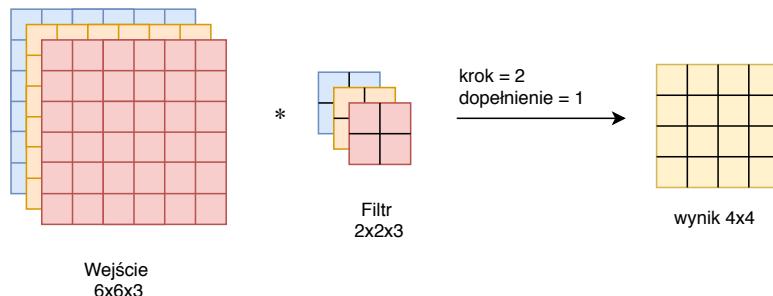
W przypadku zagadnień z dziedziny analizy obrazu, same sieci gęste robią się niepraktyczne. Teoretycznie możliwe jest przetworzenie obrazu w taki sposób, że każdy piksel traktowany jest jako jeden węzeł sieci, ale już przy obrazkach o wymiarach 100x100x3 (100 na 100 pikseli, 3 kanały RGB) daje to 30000 węzłów na warstwie pierwszej. Nawet gdyby warstwa druga miała być dwa razy mniejsza tj. 15000 węzłów macierz $w^{[1]}$ miałaby wymiar (30000, 15000). Przy założeniu, że rozmiar liczby typu double to 64 bity, to sama taka pojedyncza macierz miałaby rozmiar $64 [b] \times 30000 \times 15000 \times \frac{1}{8} \times \frac{1}{1024} \frac{1}{1024} \approx 3433 [MB]$. Poza problemami z pamięcią, leży wziąć pod uwagę wysoki koszt obliczeniowy operacji na tak dużych macierzach, a przecież rozdrobnienie współczesnych obrazów jest wielokrotnie wyższa. By poradzić sobie z tym problemem, należało wprowadzić inny rodzaj architektury sieci neuronowej – konwolucyjną sieć neuronową (ang. *convolutional neural network*, w skrócie *CNN*).

3.2. Opis elementów składających się na konwolucyjną sieć neuronową

3.2.1. Filtry i splot w konwolucyjnej sieci neuronowej

By ograniczyć liczbę węzłów, definiuje się tzw. filtry. Są to macierze kwadratowe, wypełnione wagami w których wartość jest ustalana w procesie propagacji wstecznej. Macierze te są układane w warstwy a ilość tych warstw determinowana jest przez ilość kanałów obrazu wejściowego lub poprzedniej warstwy sieci konwolucyjnej.

By uzyskać wynik, stosuje się na obrazie dyskretną operację splotu z filtrem lub filtrami. Taka operacja wykonywana jest z zadanym *krokiem* i *dopełnieniem*. Przykładową operację splotu macierzy i filtra prezentuje rysunek 3.1.



Rysunek 3.1: Przykładowa operacja splotu wykonywana między dwiema warstwami sieci CNN.

Operację wykonaną pomiędzy pojedynczą warstwą wejściową a odpowiadającym jej filtrem można zapisać:

$$\text{niech: } k = \left\lfloor \frac{n+2p}{s} \right\rfloor, k \in Z$$

$$\sum_{q=0}^{n_w} \sum_{p=0}^k \sum_{i=0}^{n_g-1} \sum_{j=0}^{n_g-1} \rho[f_q(ps+i, ps+j) g_q(i, j) + b],$$

gdzie:

- n_w – wielkość trzeciego wymiaru wejściowej macierzy,
- n – wymiar wejściowej macierzy kwadratowej, w tym przypadku równy 6,
- p – dopełnienie, wymiar dodatkowej „obwódki” wokół oryginalnego obrazu w celu wyeliminowania tzw. problemu brzegu; w tym przypadku równe 1.
- s – krok, z którym dopasowujemy filtr na macierz wejściową, w tym wypadku równy 2, to znaczy macierz filtra jest aplikowana co dwa elementy w pionie i poziomie (każdy piksel jest zakryty wyłącznie raz),
- n_g – wymiar macierzy filtra,
- $f_q(i, j)$ – funkcja reprezentująca wartości macierzy wejściowej warstwy q ,
- $g_q(i, j)$ – funkcja reprezentująca wartości macierzy filtra warstwy q ,
- ρ - funkcja aktywacji,
- b – bias, reprezentuje wartość neuronu gdy wszystkie aktywacje na jego wejściu są równe zero. Dokładna jego wartość jest ustalana razem z wagami podczas propagacji wstecznej,
- k – krok z jakim aplikowany jest filter na obrazie.

Wzór zaproponowany powyżej jest analogiczny do gęstego modelu wzoru na propagację w przód $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$, $a^{[l]} = \rho(z_n^{[l]})$, jeżeli traktować w , a i b jak macierze, a ρ jako operację działającą na każdym z elementów macierzy z osobna.

Uzyskane tym sposobem macierze wynikowe oraz wagowe są wielokrotnie mniejsze, kosztem tego, że te same wagi są ustalane przy pomocy różnych części obrazu. Można za to wprowadzić wiele różnych zestawów filtrów. Każdy z nich może nauczyć się wykrywać inne zależności w obrazie np. jeden może odpowiadać za wykrywanie krawędzi w pionie, inny w poziomie. W tym wypadku każdy z tych filtrów miałby wymiary $2 \times 2 \times 3$. Stosując wiele różnych filtrów na raz uzyskujemy wielowymiarowy wynik a wymiar trójwymiarowej macierzy wynikowej wynosi wtedy:

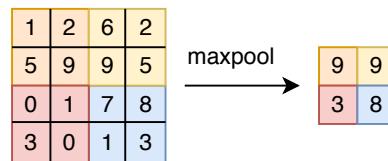
$$n_x \times n_y \times n_z$$

Gdzie $n_x = n_y = \left\lfloor \frac{n+2p-n_f}{s} \right\rfloor + 1$, n , p , s są zgodne z opisem wyżej, n_f to wymiar macierzy kwadratowej filtra a n_z – to liczba użytych filtrów wielkości $n_f \times n_f \times n_c$, gdzie n_c to ilość poprzednich warstw.

Należy jednocześnie nadmienić, że filtry nie muszą być stosowane tylko na danych wejściowych. Wynik konwolucji może i często jest, przekazywany do głębszych warstw, gdzie uczone są filtry pracujące na wynikach warstw poprzednich. O wizualizacji tego, czego się te filtry nauczyły, będzie dalej w tej pracy.

3.2.2. Warstwy łączące

Warstwy łączące (ang. *pooling layers*) są często wykorzystywane w celu zmniejszania wymiarów warstwy sieci CNN. Do najpopularniejszej należy zdecydowanie tzw. *max pooling*. Polega na tym, że z obszaru w macierzy reprezentującej wartości danej warstwy w sieci konwolucyjnej wybieramy element z najwyższą wartością.



Rysunek 3.2: Zasada działania *maxpool* dla macierzy 4×4 przy użyciu filtra 2×2 , bez dopełnienia i z krokiem $s = 2$.

Istnieje jeszcze wariacja sumująca wszystkie wartości z obszaru (ang. *sum pooling*) i licząca średnią z obszaru (ang. *avg pooling*). Zasady odnośnie liczenia wielkości macierzy wynikowej dotyczącej każdego z wariantów *poolingu* i są analogiczne jak w przypadku obliczania splotu, z

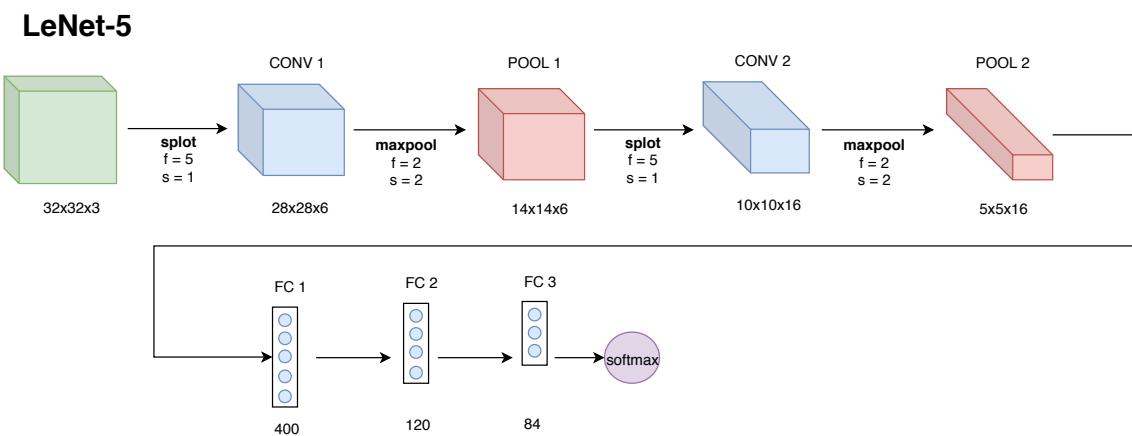
tym wyjątkiem, że *pooling* stosowany jest do każdej z warstw osobno i nie wpływa na wielkość trzeciego wymiaru macierzy wynikowej.

Popularnym schematem jest stawianie warstwy *poolingowej* zaraz po warstwie konwolucyjnej (wraz z wykonaniem aktywacji na wyniku). Warstwy *splot-pooling* często są łączone ze sobą w różnych wariantach architektury sieci CNN.

3.3. Przykład sieci konwolucyjnej LeNet-5

3.3.1. Schemat sieci LeNet-5

Przy użyciu splotu, warstw łączących i sieci gęsto połączonych, można zaprezentować kompletny model sieci konwolucyjnej. Posłużę się przykładem klasycznej sieci nazwanej LeNet-5. Oryginalnie służyła do rozpoznawania odręcznie pisanej pisma. Schemat sieci LeNet-5 przedstawia rysunek 3.3.



Rysunek 3.3: Schemat sieci konwolucyjnej LeNet-5. Jako s oznaczono krok konwolucji a f to wymiar macierzy kwadratowych składających się na filtr.

Przyjmuje ona na wejściu obrazki o wymiarach 32×32 pikseli z trzema kanałami (RGB), stosuje operacje splotu przy użyciu 6 filtrów o wymiarach $5 \times 5 \times 3$ dając, zgodnie ze wzorem, wynik o wymiarach $28 \times 28 \times 6$. Do tak uzyskanych macierzy wprowadzany jest element nielinowości przy pomocy funkcji aktywacji, w tym przypadku tangens hiperboliczny. Następnie stosowana jest warstwa *maxpool* o wymiarze filtra $2 \times 2 \times 6$. Tak uzyskany wynik trafia na splot tym razem 16 filtrów $5 \times 5 \times 6$ i *maxpool* $2 \times 2 \times 16$. Następnie przekazywany jest do w pełni połączonej sieci neuronowej, trójwarstwowej, posiadającej 400, 120 i 84 węzły na kolejnych warstwach. W wersji, którą chce zaprezentować wynik uzyskam przy pomocy *softmax* i będzie to 10 elementowy wektor.

3.3.2. Opis implementacji sieci LeNet-5

By zaimplementować zaproponowany model, posłużę się biblioteką do głębokiego uczenia maszynowego – Keras. Model będzie zaimplementowany w języku Python (wersja 3). Z racji niewielkich rozmiarów sieć będzie trenowana przy pomocy procesora. Do treningu użyję publicznie dostępnej bazy danych MNIST [1]. Zawiera ona zbiór odręcznie pisanych cyfr wraz z przyporządkowaną wartością od 0 do 9. Dane zostaną podzielone w sposób, jaki został zaproponowany przez autorów tj. 60 000 przykładów będzie użytych do treningu, a 10 000 do ewaluacji sprawności sieci. Często pracuje się na trzech zbiorach danych: do uczenia, do regulacji parametrów oraz do testów. W tym wypadku architektura sieci nie będzie jednak zmieniana w trakcie treningu, więc zestaw służący do regulacji zostanie pominięty.

Dane dostępne w postaci binarnej zostaną wczytane naraz do pamięci RAM. Następnie wartości pikseli obrazu zostaną znormalizowane z przedziału $[0, 255]$ do $[0, 1]$. Etykiety symbolizujące jaką cyfrę przedstawia obraz początkowo wczytywane są jako cyfra z przedziału $[0, 9]$. W przypadku użycia *softmax*, istnieje potrzeba dostosowania ich do postaci wektora składającego się z zer i jedynek w którym, jedynka wstawiona na odpowiedniej pozycji pokazuje jaka jest wartość etykiety (ang. *onehot vector*).

Niestety dostępne obrazy nie pasują formatem do zaproponowanej oryginalnie sieci LeNet-5. Zamiast posiadać trzy kanały, obrazy wejściowe są czarno białe i w wymiarach 28×28 pikseli, więc filtry pierwszej warstwy konwolucyjnej będą miały odpowiednio zredukowany trzeci wymiar. By nie zmieniać zbytnio oryginalnej architektury, obrazy zostały dopełnione przy pomocy zer na brzegach do wymiaru 32×32 .

Sieć trenowana jest przy pomocy wariacji metody gradientu – *Adam* wspomnianej w rozdziale 2.6. Użyta funkcja kosztu została opisana w rozdziale 2.5. Gdy sieć jest trenowana przy pomocy *Adam*, dane użyte do treningu grupowane są w mniejsze porcje (w tym przypadku 256 elementów), na których przeliczana jest propagacja w przód i w tył naraz. Następnie, na sieci wynikowej uczona jest następna porcja, aż do wyczerpania elementów ze zbioru uczącego. Taki jeden obieg nazywamy *epochem*. Podczas treningu tej sieci wykonam 5 takich *epochów*,

W celu uzyskania lepszej sprawności w przewidywaniach sieci, zastosowałem w implementacji, nie opisane do tej pory praktyki zapobiegające przeuczeniu sieci tzw. *dropout* i normalizację porcji (ang. *batch normalization*). Dropout, losowo wyłącza poszczególne neurony podczas uczenia, wymuszając równomierne rozłożenie odpowiedzialności za wykrywanie poszczególnych cech zbioru danych. *Batch normalization* natomiast skaluje wagę warstw ukrytych tak, by mediana i odchylenie standardowe nie zmieniały się zbyt drastycznie. Daje to pozytywne efekty dla czasu treningu sieci i w pewnym stopniu, tak samo jak *dropout*, normalizuje wagę sieci.

3.3.3. Implementacja modelu przy pomocy *Keras*

Ten niewielki model jest idealnym przykładem, w jaki sposób modeluje się sieci neuronowe przy pomocy Kerasa. Kod do wczytywania danych został pominięty, ponieważ nie jest istotny z punktu widzenia treningu. Po wczytaniu danych, obrazy użyte do treningu są dopełniane do wielkości 32×32 przy pomocy metody *ZeroPadding2D*, następnie przy pomocy *Conv2D* definiuję warstwę konwolucyjną, normalizuję jej wagę przy pomocy *BatchNormalization* i aktywuję przy pomocy tangensa hiperbolicznego. Następnie stosuję warstwę *MaxPooling2D*, analogiczne warstwy *Conv2D* oraz *BatchNormalization* i całość wprowadzam do warstw gęstej sieci (warstwy *Dense*), stosując uprzednio dwa razy *Dropout*.

```

1 #loading the dataset
2 train_labels, test_labels, train_images, test_images,
3             train_images_orig, test_images_orig = loadTrainingData();
4 ## model fitting below ##
5 X_input = Input((28, 28, 1))
6 # Padding to the 32x32
7 X = ZeroPadding2D((4, 4))(X_input)
8 # CONV -> BN -> RELU
9 X = Conv2D(6, (5, 5), strides = (1, 1), name = 'conv0')(X)
10 X = BatchNormalization(axis = 3, name = 'bn0')(X)
11 X = Activation('tanh')(X)
12 # MAXPOOL
13 X = MaxPooling2D((2, 2), name='max_pool_1', strides=2)(X)
14 # CONV -> BN -> RELU
15 X = Conv2D(16, (5, 5), strides = (1, 1), name = 'conv1')(X)
16 X = BatchNormalization(axis = 3, name = 'bn1')(X)
17 X = Dropout(0.4, name='drop_1')(X)
18 X = Dropout(0.3, name="drop_2")(X)
19 X = Dense(400, activation='tanh', name='fc_in')(X)
20 X = Dense(120, activation='tanh', name='fc_mid')(X)
21 X = Dense(84, activation='tanh', name='fc_out')(X)
22 X = Dense(10, activation='softmax', name='fc_softmax')(X)
23 # Create model
24 model = Model(inputs = X_input, outputs = X, name='ResNet-5')
25 model.compile(optimizer="Adam",
26                 loss="binary_crossentropy", metrics=["accuracy"])
27 # Fit the model
28 model.fit(x=train_images, y=train_labels, epochs=5, batch_size=256)
29 # Evaluate
30 preds = model.evaluate(x = test_images, y = test_labels)
31 print ("Loss = " + str(preds[0]))

```

```
32 print ("Test Accuracy = " + str(preds[1]))
```

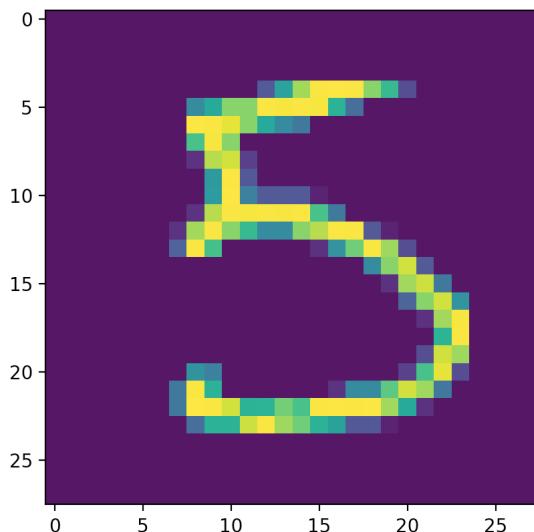
Listing 3.1: Model sieci LeNet-5 w Keras.

Model zdefiniowany przy pomocy klasy *Model*, trenowany jest przy pomocy funkcji optymalizującej *Adam* przy użyciu funkcji kosztu binary crossentropy zdefiniowanych przy pomocy metody *compile*. Trening modelu zaczyna się po wywołaniu metody *fit*, przyjmującej dane, na których ma być wytrenowana. Metoda *evaluate* testuje skuteczność modelu na danych testowych.

```
Epoch 1/5
60000/60000 [=====] - 40s 667us/step - loss: 0.0600 - acc: 0.9790
Epoch 2/5
60000/60000 [=====] - 39s 654us/step - loss: 0.0247 - acc: 0.9914
Epoch 3/5
60000/60000 [=====] - 44s 733us/step - loss: 0.0200 - acc: 0.9932
Epoch 4/5
60000/60000 [=====] - 45s 745us/step - loss: 0.0172 - acc: 0.9941
Epoch 5/5
60000/60000 [=====] - 45s 748us/step - loss: 0.0152 - acc: 0.9948
10000/10000 [=====] - 2s 180us/step
Loss = 0.015972366375336423
Test Accuracy = 0.9943899975776672
>>> |
```

Rysunek 3.4: Wynik treningu sieci LeNet-5.

Trening trwał niecałe 4 minuty i uzyskał skuteczność powyżej 99%, co jest bardzo zadawającym wynikiem.



Rysunek 3.5: Przykładowy obraz ze zbioru testowego.

```

>>> x = test_images[15]
>>> x = np.expand_dims(x, axis=0)
>>> x = model.predict(x)
>>> np.where(x == np.max(x))[1]
array([5])
>>> []

```

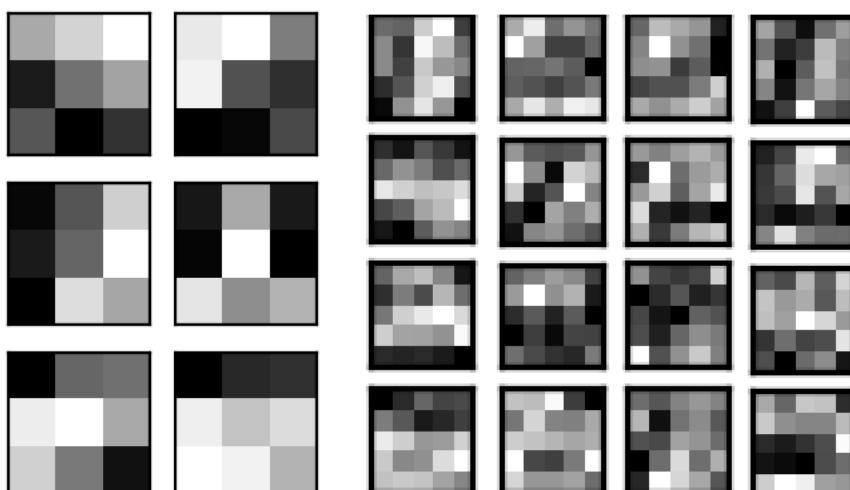
Rysunek 3.6: Odpowiedź modelu na pytanie, jaką cyfrę rozpoznaje.

3.3.4. Wizualizacja neuronów warstw konwolucyjnych sieci LeNet-5.

Mając do dyspozycji wytrenowany model, można przystąpić do próby wizualizacji tego, czego nauczyła się sieć. Przy modelu tak płytkim jak LeNet-5 uzyskane odpowiedzi powinny być mniej imponujące wizualnie, ale przez to łatwiejsze w interpretacji.

Wizualizacja poprzez wyświetlenie wag warstw konwolucyjnych.

Pierwszy ze sposobów, który zaprezentuję będzie też najprostszy koncepcyjnie. Wyrysowanie filtrów warstw konwolucyjnych powinno dać ogólny zarys tego, jakie cechy obrazu (gradient, linie poziome/pionowe/ukośne) były brane pod uwagę przez sieć. W celu łatwiejszej interpretacji zestawię ze sobą filtry 5×5 oryginalnie występujące w sieci LeNet-5 z wariacją sieci wytrenowanej na filtrach 3×3 , którą wytrenałem tylko po to, by mieć punkt odniesienia do filtrów klasycznie używanych w analizie obrazu.



Rysunek 3.7: Wizualizacja filtrów sieci LeNet-5. Od lewej filtry warstwy pierwszej wersji zmodyfikowanej 3×3 . Z prawej filtry warstwy drugiej wersji oryginalnej.

Na filtrach 3×3 widać tendencję w wykrywaniu krawędzi pionowych, poziomych i ukośnych. Dodatkowo duży nacisk położony jest na wykrywanie gradientu po skosie (dwa pierwsze filtry od góry i dwa ostatnie od dołu). Widoczne jest, że macierze filtrów odbiegają trochę war-

tościami od klasycznie używanych macierzy w analizie obrazów. Mogło być to spowodowane przedewszystkim charakterem używanego zbioru danych, sieć mogła dostosować używane filtry by te pasowały jak najbardziej do danych wejściowych. Dodatkowo, użyte dopełnienie z 28 do 32 pikseli nie było obojętne na uzyskane wartości w macierzach.

W przypadku filtrów 5×5 kształty, które próbuje dopasować sieć robią się bardziej skomplikowane, przez co trudniejsze w interpretacji. Podobnie jak w przypadku filtrów warstwy pierwszej, można znaleźć filtry szukające ukośnych gradientów oraz linii w pionie i poziomie, ale oprócz tego znajdują się tu filtry, które intepretowałbym jako elementy strukturalne próbujące dopasować się do konkretnych kształtów występujących w poszczególnych elementach. Należy jednak podkreślić, że wyświetcone filtry 5×5 operują na wyniku konwolucji i *maxpoolingu* warstwy pierwszej – są przez to trudne w interpretacji.

Wniosek jest taki, że tym wypadku najprostsze rozwiążanie nie jest najlepsze. Nie będę więc dalej analizował tej metody ani stosował jej dla modeli o większej złożoności, bo im głębszą warstwę sieci wybiorę tym trudniejsze w interpretacji będą wyniki.

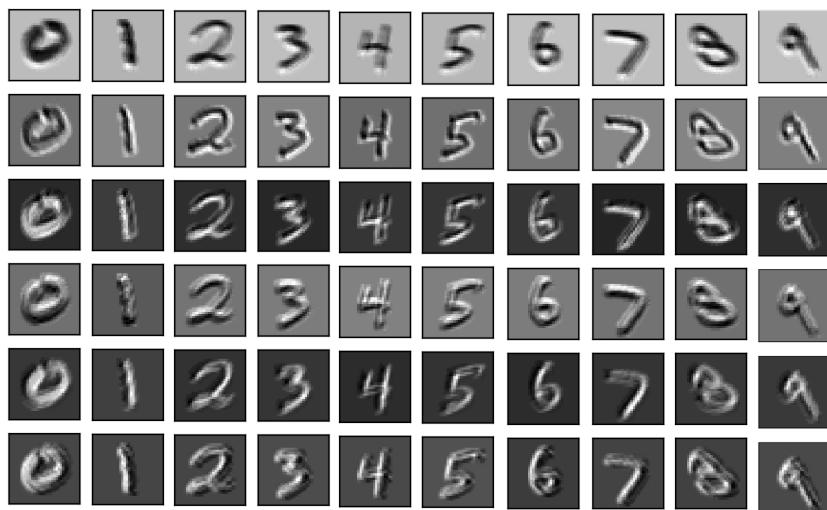
Wizualizacja poprzez mapy cech.

Mapy cech czy inaczej mapy aktywacji, wizualizują rezultat zastosowania filtrów na danych wejściowych – np. obrazu wejściowego czy innej mapy cech. Przykładowo, dla danego obrazu można wyrysować jakie, konkretnie jego cechy zostały wyszczególnione przez warstwę sieci CNN. By uzyskać mapę cech dla danego obrazu, należy zmodyfikować model w taki sposób, by jego warstwą wyjściową była ta warstwa konwolucyjna, której cechy chcemy uzyskać

```
1 img = np.expand_dims(img, axis=0)
2 vmodel = Model(inputs=model.inputs, outputs=model.layers[layer_no].output)
3 feature_maps = vmodel.predict(img)
```

Listing 3.2: Uzyskiwanie mapy aktywacji dla danego modelu i obrazu w Keras.

W ogólności, warstwy bliżej źródła danych powinny zawierać informacje o dużej ilości szczegółów, a warstwy głębsze, wykrywać cechy bardziej ogólne. I tak, dla pierwszej warstwy konwolucyjnej sieci LeNet-5 wyrysowane cechy dla przykładu z każdej klas bardzo mocno przypominają oryginalne obrazy. Dostrzegalne różnice pomiędzy warstwami polegają na dostrzeganiu innego „konturu” obrazu każda z warstw akcentuje inny jego rodzaj, przez co liczby dają iluzję bycia „oświetlonymi” z różnych stron. Zgadza się to z obserwacją poczynioną w przypadku analizy wartości filtrów. Dodatkowo, każda aktywacja różni się trochę kolorem tła co nasuwa logiczny wniosek, że aktywacja nie jest zależna od tła otoczenia (choć prawdopodobnie niejednolite tło mogłoby negatywnie wpływać na wydajność sieci).

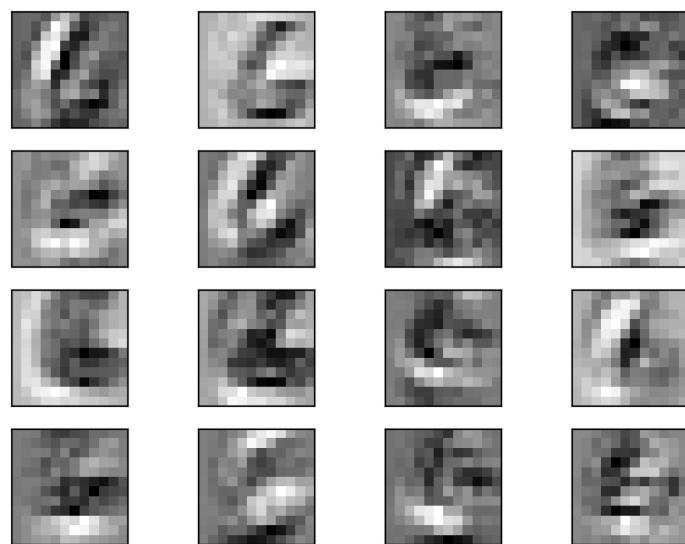


Rysunek 3.8: Mapy aktywacji dla 6 filtrów warstwy pierwszej, przy użyciu przykładu dla każdej z klas. Filtry użyte do aktywacji ustawione są w rzędach tak, że indeks rzędu odpowiada numerowi filtra. Kolejne kolumny odpowiadają użytej klasie.

Sytuacja robi się jeszcze ciekawsza w przypadku warstwy drugiej. Tam jest już 16 filtrów, więc nie zostaną zaprezentowane dla każdej z klas z osobna. Niemniej jednak, sam wynik dla pojedynczej klasy jest w stanie wiele powiedzieć na temat branych pod uwagę cech.

Uzyskane aktywacje wciąż przypominają oryginalną szóstkę, lecz tym razem, zgodnie z oczekiwaniami, cechy są bardziej zgeneralizowane. Wyraźnie widać skupienie filtrów na dolnej „pętli” szóstki oraz na tym, by posiadała odpowiednie łuki na brzegach (wyraźnie zarysowany jest tam gradient, widoczny szczególnie na filtrach w ostatnim rzędzie).

Niestety sieć LeNet-5 jest na tyle płytką, a zestaw odręcznie pisanych cyfr na tyle mało skomplikowany, że nie uda mi się przy ich pomocy wyekstrahować bardziej złożonych wzorów. W tym celu będę musiał posłużyć się bardziej złożonym modelem – siecią VGG-19.

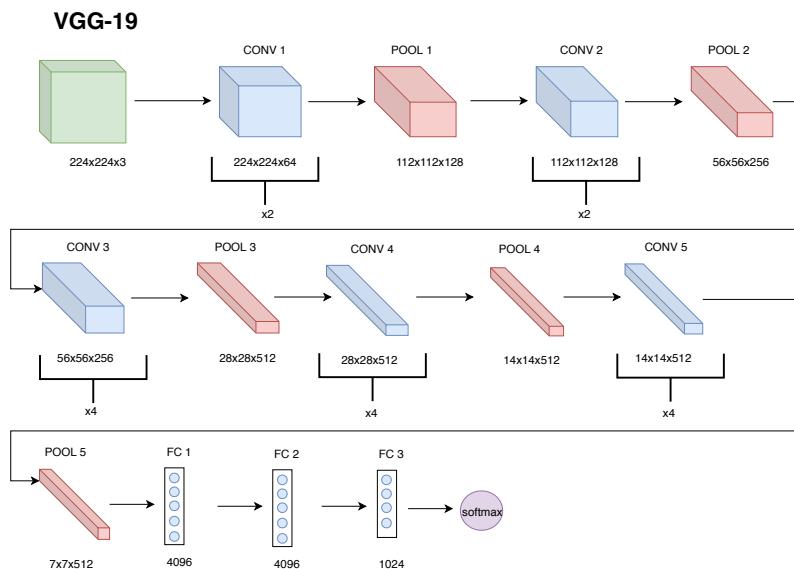


Rysunek 3.9: Mapy aktywacji dla 16 filtrów warstwy drugiej, przy użyciu przykładowej reprezentacji liczby 6.

4. Wizualizacja przy pomocy warstw ukrytych sieci VGG-19

4.1. Model sieci VGG-19

Do bardziej złożonych wizualizacji posłużę się siecią VGG, której architekturę zaprezentowano po raz pierwszy w 2015 roku w publikacji autorstwa Karen Simonyan i Andrew Zisserman[2]. W publikacji zaprezentowano kilka wariantów tej sieci. Ja z uwagi na większą ilość warstw użyję wariantu VGG-19.



Rysunek 4.1: Schemat modelu sieci VGG-19.

Dopełnienie w przypadku warstw konwolucyjnych wynosi $p = 1$ przy skoku $s = 1$ co sprawia, że warstwy konwolucyjne mogą być ze sobą łączone w łańcuchy niezmieniające wymiarów tensora. W przypadku warstw *poolingowych* $p = 1$ przy skoku $s = 2$ tym samym, dzielą one dwa pierwsze wymiary przez pół. Sumarycznie daje to 16 warstw możliwych do wykorzystania w wizualizacjach.

4.1.1. Wizualizacja poprzez rekonstrukcję obrazu przy pomocy maksymalizacji aktywacji neuronu

Z racji swoich rozmiarów, trening sieci VGG zajmuje dni, nawet przy pomocy GPU. Użyję więc wcześniej wytrenowanej sieci dostępnej bezpośrednio w Kerasie.

```
1 from keras.applications import VGG19
2 model = VGG19(include_top=False, weights='imagenet')
```

Listing 4.1: Wczytywanie wag VGG-19 w Keras.

Zastosowane opcje to przede wszystkim zestaw danych na, których była trenowana sieć, w tym wypadku zbiór ImageNet[3], oraz wyłączenie z ładowanego modelu warstw gęstych. Ta ostatnia opcja pozwoli na rekonstrukcję obrazu o dowolnej liczbie pikseli (zamiast standardowego dla architektury VGG 224×224 pikseli).

Mając załadowany model wraz z wytrenowanymi wagami, wybieram poszczególne filtry z kolejnych warstw konwolucyjnych. Następnie, generuję losowy (wartość pikseli losowana jest z rozkładu jednorodnego) obrazek o niskiej rozdzielcości, początkowo szary obraz zaburzam nieznacznie różnymi kolorami i wyliczam aktywacje wybranej wcześniej warstwy. Przykładowy obraz wejściowy przedstawia rysunek 4.2



Rysunek 4.2: Obraz wejściowy dla skryptu maksymalizującego medianę warstwy.

Medianę z tych aktywacji traktuję jako wartość mojej funkcji kosztu i modyfikuję wylosowany uprzednio obrazek w celu jej maksymalizacji.

Maksymalizacja aktywacji jest możliwa poprzez wywołanie metody o nazwie *gradients* z biblioteki Keras. Wykorzystując ją można przy pomocy tensora wejściowego, w tym wypadku zdjęcia, oraz funkcji kosztu (zdefiniowanej powyżej) uzyskać gradient funkcji kosztu dla danych z tensora. Następnie, moim celem będzie wykonać operację odwrotną niż w podczas treningu sieci – maksymalizację funkcji kosztu. Uzyskany gradient wykorzystywany jest do przekształcania obrazu.

Modyfikacja obrazu polega na dodaniu odpowiednio znormalizowanego gradientu do wartości poszczególnych pikseli. Cały proces treningu składa się z kilkukrotnego skalowania obrazu do wyższych „rozdzielcości” i ponawiania procesu optymalizacji.

Skalowanie jest konieczne, ponieważ struktury generowane tą metodą charakteryzują się wysoką częstotliwością (są małe i powtarzalne). Uprzednio wygenerowanie fragmentu wzoru w niskiej rozdzielczości i skalowanie, sprawia, że często wzór jest niejako „dobudowywany” zamiast powtarzany, przez co jest większy – co czasem daje lepsze zrozumienie na co patrzymy, choć nie jest to niestety reguła.

```

1 output = layers_by_name[layer_name].output
2 loss = K.mean(output[:, :, :, filter_index])
3 gradients = K.gradients(loss, input_tensor)[0]
4 img = generate_grayscale_img();
5 for interpolation_step in image_resize_steps:
6     for i in range(EPOCHS_NUMBER):
7         loss_val, gradients_val = step([img])
8         img += gradients_val
9     resize_img(img);

```

Listing 4.2: Wizualizowanie poprzez maksymalizację mediany wybranej warstwy.

Listing 4.1.1 zawiera wysokopoziomowy zarys tego, w jaki sposób działa skrypt generujący poniższe wizualizacje. Wszystkie szczegóły implementacyjne, takie jak normalizacja gradientu czy konwersja obrazu z postaci nadającej się do treningu na postać zdatną do wyświetlenia zostały pominięte.

Zgodnie z oczekiwaniemi, złożoność uzyskanych wizualizacji rośnie wraz numerem warstwy, na podstawie której je uzyskano. Uzyskane obrazy nie prezentują żadnej ze znanych mi klas na których trenowano tę konkretną instancję VGG19, a raczej „teksturę” obserwowanego przedmiotu.

Pierwsze warstwy nie są zbytnio bogate w informacje. Kodują podstawowe informacje o kolorze, nie posiadając zbyt wiele informacji o strukturze klasyfikowanego przedmiotu. Choć i już tu trafiają się ciekawe wizualizacje jak np. filtra 5 warstwy conv1 bloku 1 - przypominającej trochę księżyca.

U części z tych wizualizacji (szczególnie na dalszych początkowych warstwach) można zaobserwować struktury przypominające korę drzew. Kolejne wizualizacje robią się bardziej kolorowe i przechodzą w bardziej abstrakcyjne wzory, które wciąż możnaby wziąć za tkaninę czy chmurę. Niestety uchwycone zależności na warstwach głębszych, choć fascynujące, są poza moimi możliwościami interpretacji.

Choć przytoczony w pracy zestaw wizualizacji nie posiada dużej ich reprezentacji, podczas tworzenia wizualizacji natknąłem się na wiele podobnych do siebie struktur, nieznacznie tylko obróconych o jakiś kąt. Za przykład może posłużyć wizualizacja z warstw conv5 bloku 1, filtry 30 i 3 na rysunku 4.7.

Biorąc pod uwagę to, że w sieciach CNN filtry aplikowane są cały czas w taki sam sposób, czyli od lewej do prawej, z góry na dół; a przedmioty występujące w danych wejściowych często występują w różnych położeniach nie wydaje się być zaskakujące, że istnieje potrzeba stosowania takich samych filtrów o różnych rotacjach. Jest to zaskakująco analogiczna sytuacja jak w klasycznej analizie obrazu, gdzie stosowane filtry do wykrywania krawędzi pionowych i poziomych są takimi samymi, po uprzedniej operacji transponowania, macierzami. Pozostawia to pole do poprawy działania takich sieci. Odpowiednie kadrowanie, rotacja danych wejściowych lub jakakolwiek forma adaptacyjnej aplikacji filtrów mogłaby sprawić, że trening tych samych, obróconych filtrów byłby zbędny co w rezultacie sprawiłoby, że nie potrzeba byłoby ich tak dużo, co mogłoby skrócić trening sieci przy braku negatywnego wpływu na sprawność predykcji.

4.1.2. Wizualizacja maksymalnej aktywacji danej klasy

VGG19 w inny od człowieka sposób uchwycia to czym jest dany na obrazie przedmiot. By zwizualizować w jaki sposób sieć odróżnia poszczególne klasy, można posłużyć się analogicznym sposobem postępowania jak z podrozdziału 4.1.1. Jeżeli do modelu przywrócićby warstwy gęste i próbować zmaksymalizować aktywację dla jednej z klas poprzez modyfikację obrazu wejściowego, powiniennem otrzymać obraz, który sieć bezwątpliwie traktowałaby jako reprezentanta danej klasy. Gdyby sieć miała w sobie zakodowaną informację o tym co tak naprawdę jest na obrazie powiniennem otrzymać coś przynajmniej odlegle przypominającego oryginalną klasę.

Weźmy przykładowo klasę numer 319 w przypadku tego konkretnego modelu odpowiada ona klasie ważki.

```

1 img = image.load_img('resources/vgg_mean_topincluded/dragonfly.jpeg',
2                      target_size=(224, 224))
3 x = image.img_to_array(img)
4 x = np.expand_dims(x, axis=0)
5 x = preprocess_input(x)
6 preds = model.predict(x)
7 print('Predicted:', decode_predictions(preds, top=3)[0])

```

Listing 4.3: Predykcja klasy zdjęcia ważki4.8.

```

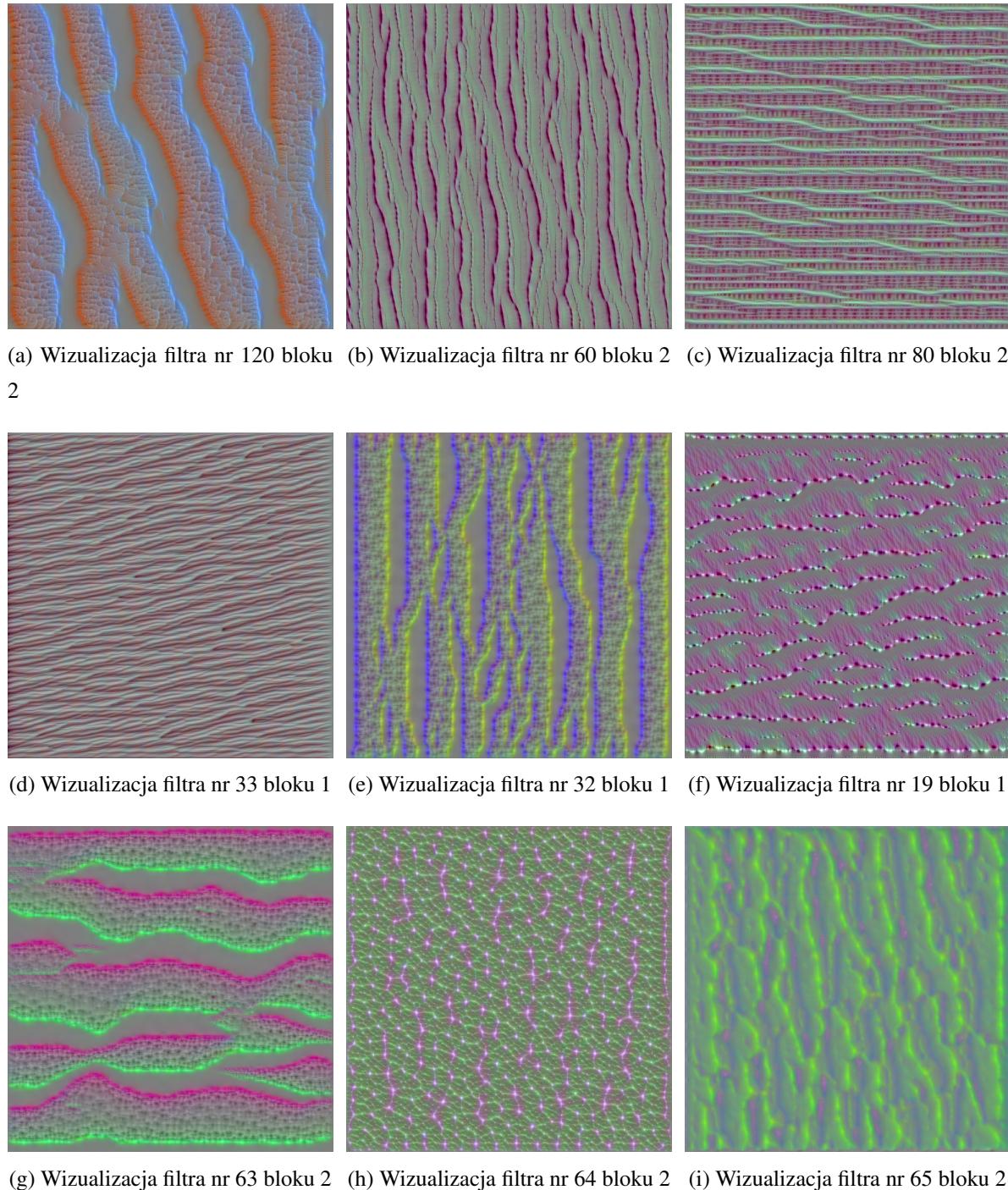
1 [ ('n02268443', 'dragonfly', 0.96805733), ('n02268853', 'damselfly',
2      0.01810956), ('n02264363', 'lacewing', 0.012446008) ]

```

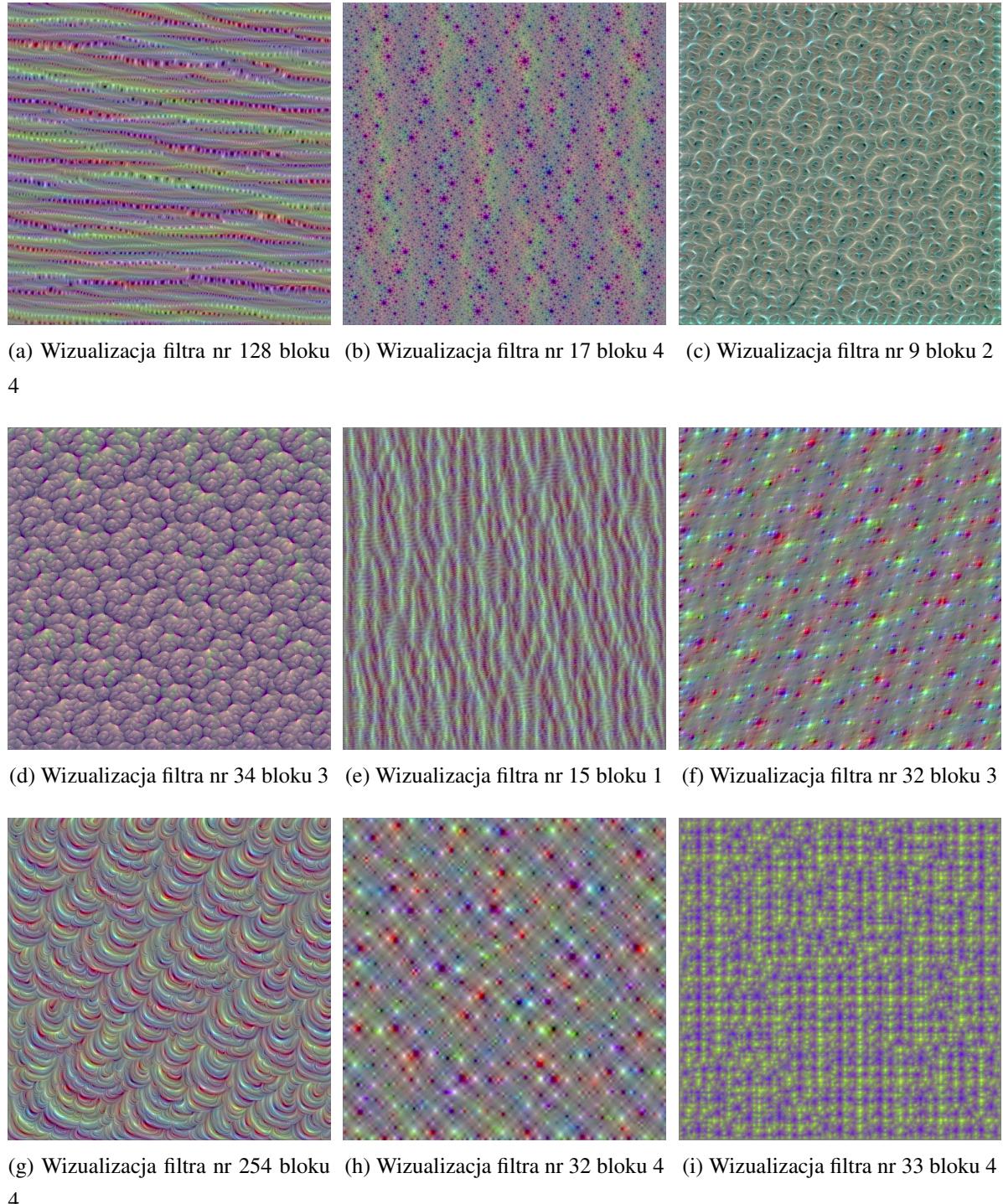
Listing 4.4: Wynik skryptu 4.1.2.



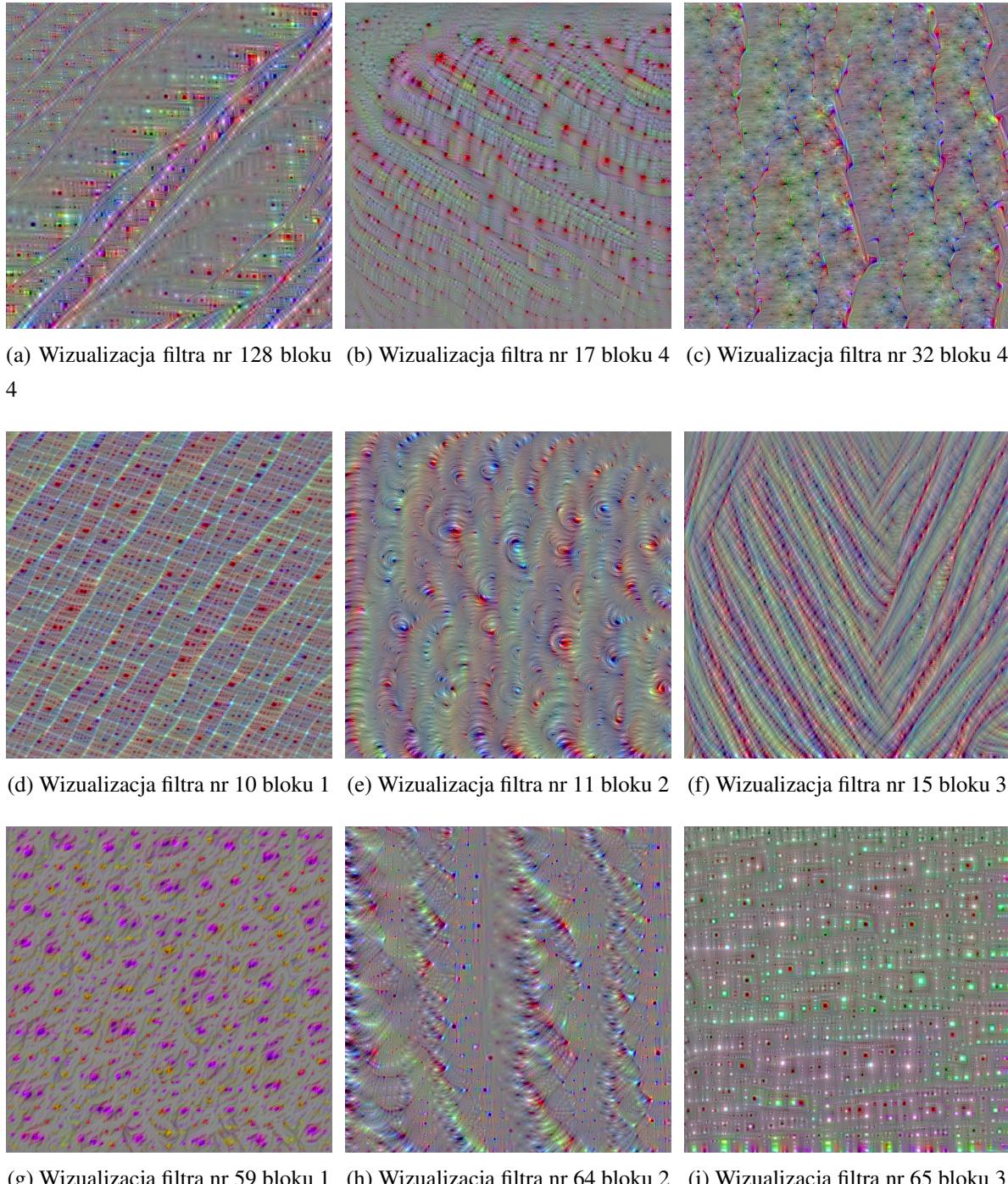
Rysunek 4.3: Wybrane wizualizacje warstwy sieci VGG-19 oznaczonej na rysunku 4.1 jako conv1



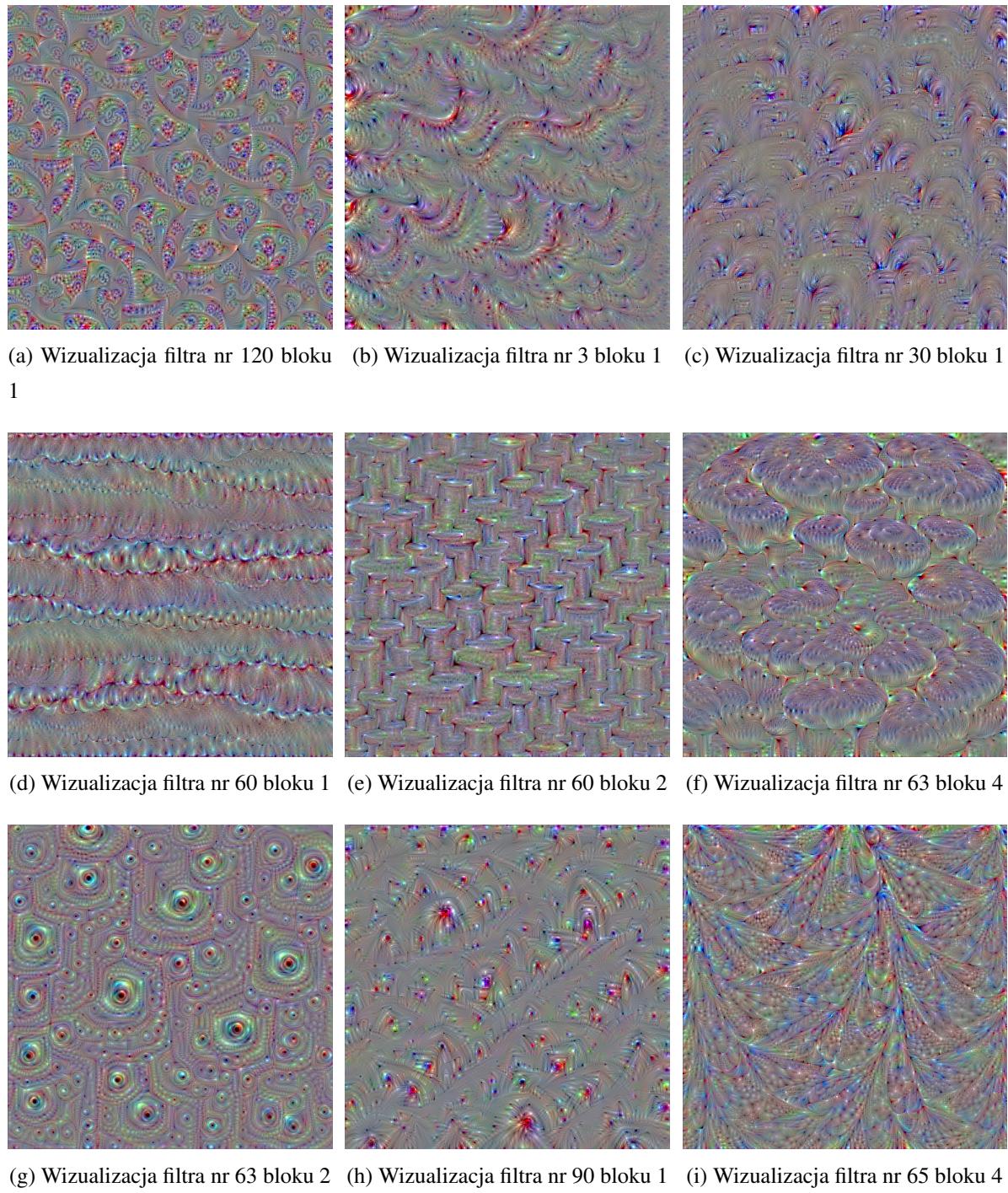
Rysunek 4.4: Wybrane wizualizacje warstwy sieci VGG-19 oznaczonej na rysunku 4.1 jako conv2



Rysunek 4.5: Wybrane wizualizacje warstwy sieci VGG-19 oznaczonej na rysunku 4.1 jako conv3



Rysunek 4.6: Wybrane wizualizacje początkowej warstwy sieci VGG-19 (oznaczonej na rysunku 4.1 jako conv4)



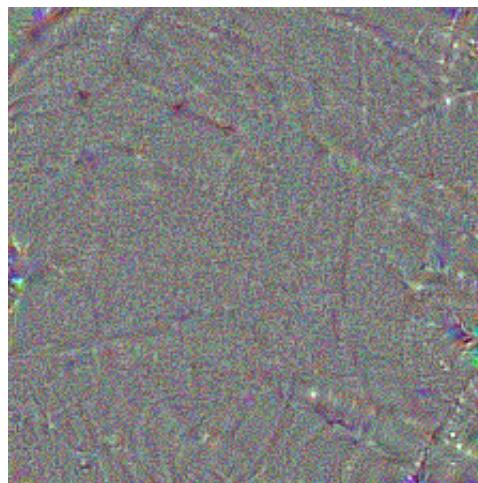
Rysunek 4.7: Wybrane wizualizacje warstwy sieci VGG-19 oznaczonej na rysunku 4.1 jako conv5



Rysunek 4.8: Przykładowe zdjęcie ważki.

Oznacza to nie mniej, że sieć którą dysponuję jest przekonana, w zaokrągleniu, na 97% mamy do czynienia z ważką. Dwie kolejne najbardziej prawdopodobne klasy przedstawiają inny rodzaj ważki - ważkę równoskrzydłą i owada siatkoskrzydłego.

Wszystko wydaje się być w porządku dopóki nie wygenerujemy obrazu na podstawie wyżej wymienionej klasy. Uprzednio modyfikując funkcję kosztu, tak by teraz zwracała wartość jaką przypisuje klasie ważki softmax, możemy zmodyfikować szary szum tak by maksymalizować (wg. sieci) prawdopodobieństwo tego, że na obrazie jest ważka. Tym nie skaluje obrazu podczas treningu i modyfikuję obraz 224×224 .



Rysunek 4.9: Ważka uzyskana w odwróconym procesie treningu.

Mimo krótkiego czasu treningu taki obraz klasyfikowany jest przez sieć VGG19 jako ważka z pewnością sięgającą 96%. Podobny wynik uzyskałem na wszystkich testowanych przeze mnie klasach. VGG rozpoznaje przedmioty na podstawie prawdopodobieństwa wystąpienia pewnej kombinacji filtrów - niestety samo nie nadaje się do generowania realistycznych obrazów danej klasy. By uzyskać obrazy zdolne do oszukania ludzkiej percepcji należy posłużyć się architekturą typu GAN (*ang. Generative Adversarial Networks*).

Niezwyczajnym zajmującym tematem wydaje się być próba wprowadzenia uzyskanych wizualizacji do sieci VGG i uzyskanie odpowiedzi samej sieci co jest na danym obrazie. Zmuszony jednak jestem zostawić temat wizualizacji uzyskanych za pomocą maksymalizacji aktywacji warstwy na rzecz innego zajmującego tematu - neuronowego transferu stylu.

4.2. Neural Style Transfer przy pomocy sieci VGG

Neural Style Transfer został pierwszy raz zaprezentowany w pracy autorstwa Leon A. Gatys, Alexander S. Ecker, Matthias Bethge [4]. Jest to technika mająca na celu uzyskanie obrazu G w stylu obrazu S , przy pomocy obrazu bazowego B 4.10.

Jej oryginalni autorzy sami podają kluczową obserwację, umożliwiającą mieszanie obrazu. Jest to spostrzeżenie, że możliwe jest odesparowanie stylu i treści z zakodowanej w aktywacjach sieci VGG.

Założymy, że chcemy za pomocą wybranej warstwy konwolucyjnej sieci VGG odzwierciedlić treść obrazu C na obrazie G . Mamy do wyboru warstwy oznaczone conv1...5. Gdy wybierzemy jedną warstwę i wykonamy dla obrazów C i G kroki propagacji w przód otrzymamy mapy aktywacji a_C i a_G . Korzystając z nich, można zdefiniować funkcję kosztu:

$$J_c^{[l]}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_F} \sum (a_C - a_G)^2$$

Gdzie:

- n_W – szerokość filtra wybranej warstwy,
- n_H – wysokość filtra wybranej warstwy,
- n_F – ilość filtrów wybranej warstwy,
- l – numer zastosowanej warstwy,

A sumowanie odbywa się po wszystkich elementach mapy cech (uzyskana suma jest sumą aktywacji wszystkich warstw).

Analogicznie można potraktować problem odzwierciedlenia stylu obrazu. Po wyborze warstwy uzyskujemy aktywację dla obrazów S i G oraz oznaczamy je kolejno a_S i a_G . Następnie zwijamy tensorzy o wymiarach $n_H \times n_W \times n_F$ w dwuwymiarowe macierze o wymiarach $(n_H + n_W) \times n_F$ i wyznaczamy macierze Grama tych macierzy, oznaczamy je jako $G^{(S)}$ i $G^{(G)}$. Wtedy można zdefiniować następującą funkcję kosztu stylu.

$$J_s^{[l]}(S, G) = \frac{1}{4 \times n_F^2 \times (n_H \times n_W)^2} \sum_{i=1}^{n_F} \sum_{j=1}^{n_F} (G_{ij}^{(S)} - G_{ij}^{(G)})^2 \quad (2)$$

Gdy zdefiniujemy zbiorczą funkcję kosztu jako:

$$J(G) = \alpha J_c(C, G) + \beta J_s(S, G)$$

Gdzie:

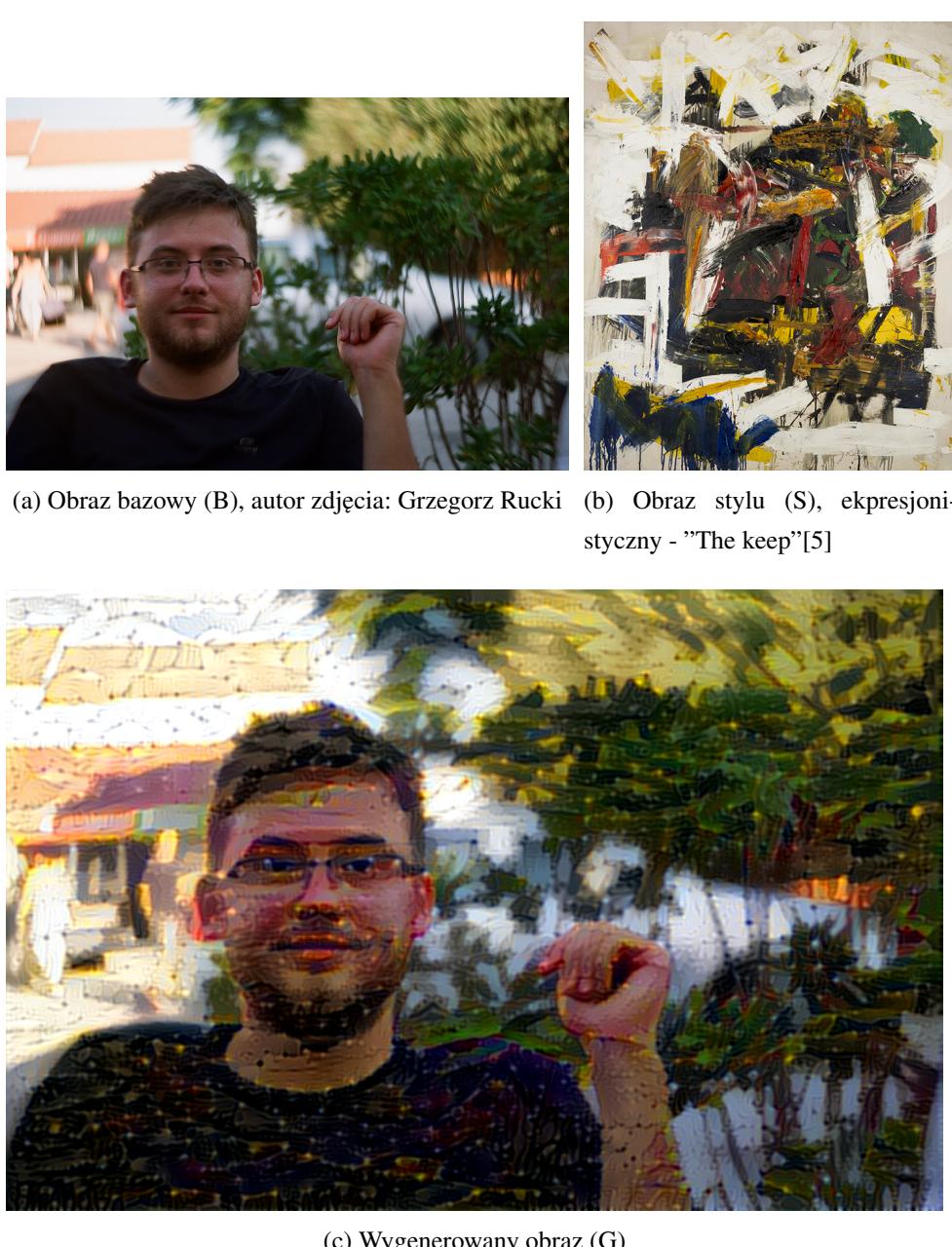
- α – współczynnik kosztu treści,
- β – współczynnik kosztu stylu,

Sterując α i β można wpływać na to jak bardzo obraz będzie wierny orginałowi C w stosunku do jak bardzo będzie namalowany w stylu S . Używając tak zdefiniowanej funkcji kosztu dla jednej z warstw, można zapisać funkcję biorącą pod uwagę wyjście z kilku warstw konwolucyjnych jednocześnie:

$$J_s(S, G) = \sum_l \lambda^{[l]} J_s^{[l]}(S, G)$$

Gdzie $\lambda^{[l]}$ jest współczynnikiem wykorzystania danej warstwy l .

Przy pomocy tej funkcji, można zastosować sposób działania znany z podrozdziału 4.1.1. Używając ją zamiast mediany aktywacji danej warstwy uzyskamy efekt widoczny na rysunku 4.10. Szczegół implementacyjny: zaprezentowany obraz został wygenerowany przy pomocy skryptu napisanego w *tensorflow* a nie na podstawie modyfikacji opisanego wyżej skryptu. Powód jest ściśle praktyczny, dysponowałem już takim skryptem mojego autorstwa - nie było praktyczne jednak wprowadzanie nowej biblioteki uczenia maszynowego, a jak było napisane wyżej, identyczne efekty można uzyskać przy pomocy Kerasa.



(c) Wygenerowany obraz (G)

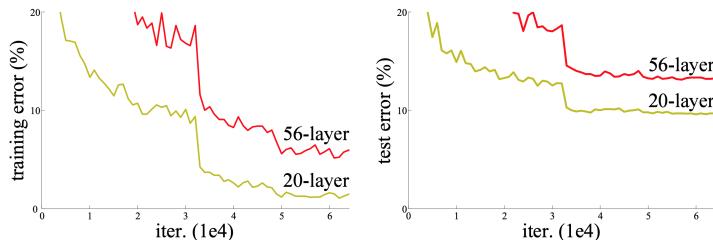
Rysunek 4.10: Transfer stylu na przykładzie zdjęcia z wakacji.

5. Konwolucyjna sieć incepcja

5.1. Architektura sieci incepcja

5.1.1. Wstęp

Siec Incepcja (*ang. Inception Network*) została zaprezentowana w 2015 roku w publikacji *ang. Going deeper with convolutions* [6]. Nazwa sieci (wraz z tytułem publikacji) nawiązuje do popularnego w tamtym czasie internetowego mema z Leonardo DiCaprio. Incepcja wychodzi na przeciw problemom z treningiem bardzo głębokich sieci neuronowych. Jednym z nich jest rosnący margines błędu wraz rosnącą ilością warstw co przedstawia rysunek 5.1.



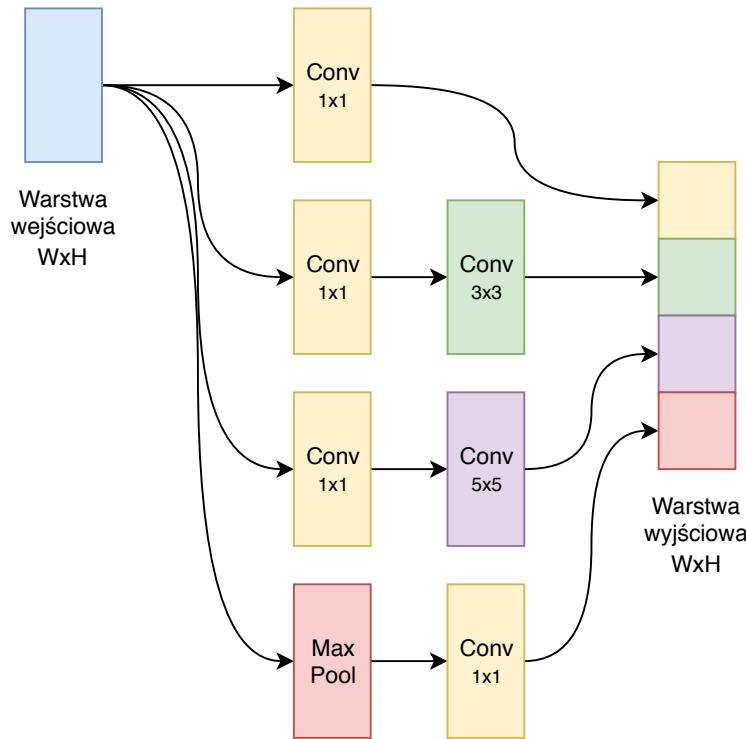
Rysunek 5.1: Błąd predykcji w stosunku do zastosowanej ilości iteracji treningowych przy uwzględnieniu ilości użytych warstw ukrytych dla standardowej architektury sieci neuronowej [7].

Incepcja rozwiązuje ten problem rozbudowując się o dodatkowe operacje splotu w obrębie jednej warstwy-bloku (opisanego w podrozdziale 5.1.2). To wraz ze zastosowaniem operacji splotu o wymiarach filtra 1×1 (rysunek 5.3), sprawiło, że można było zbudować sieć o jeszcze większej ilości warstw ukrytych, która jednocześnie zyskiwała na jakości predykcji.

5.1.2. Blok sieci incepcja

Główną cechą sieci typu incepcja jest to, że jej architektura została rozbudowana o dodatkowe operacje splotu w obrębie jednej warstwy 5.2. Mając do dyspozycji warstwę (lub N

warstw) o wymiarach filtra $W \times H$ stosujemy równocześnie splot o wymiarach filtra 1×1 , 3×3 , 5×5 oraz warstwę MaxPool. Wynik tych operacji konkatenowany jest i zapisywany jako warstwa wyjściowa. Wykonywane operacje splotu mają tak dobrane krok i dopełnienia by wymiar W i H nie uległy zmianie. Taki dobór parametrów ma na celu łączenie jednego bloku sieci incepcja bezpośrednio w drugi podobnie jak ma to miejsce w przypadku warstw sieci VGG. Podobnie jak w przypadku VGG w celu modyfikacji dwóch pierwszych wymiarów warstwy stosowane są warstwy *MaxPool* bezpośrednio między blokami incepcji.

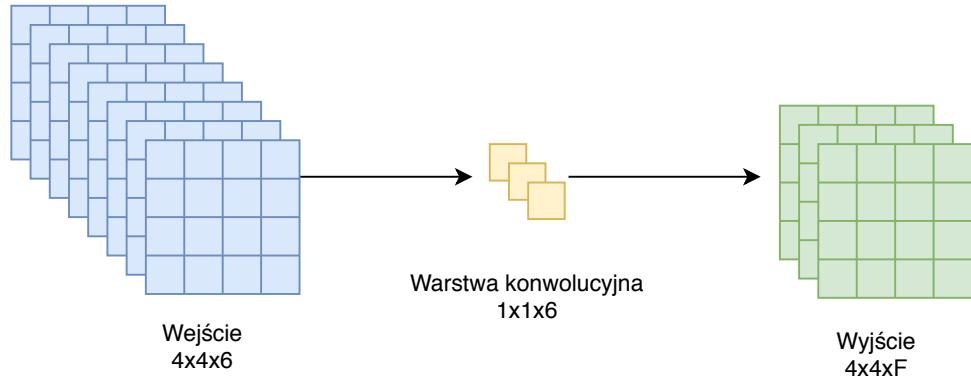


Rysunek 5.2: Schemat bloku z których składa się sieć incepcja.

Operacje splotu o wymiarach 1×1 są stosowane przed 3×3 i 5×5 w celu redukcji liczby wartości filtrów co ma pozytywny wpływ na koszt obliczeniowy tej operacji 5.3. Stosując F zestawów filtrów można dowolnie modyfikować trzeci wymiar uzyskanej warstwy. Dzięki temu, razem z MaxPool, splot 1×1 pozwala na dowolną modyfikację wymiarów uzyskiwanych konwolucji.

5.1.3. Schemat sieci

Zaprezentowany wariant sieci Incepcja, tak zwany GoogleLeNet 5.5 to 22 warstwowa sieć konwolucyjna. Oprócz widocznych dla Incepcji bloków, GoogleLeNet posiada 3 osobne warstwy wyjściowe typu softmax. Każde z tych wyjść używane jest w czasie treningu do obliczania funkcji kosztu (ich koszt dodawany jest do całkowitego kosztu z wagą 0,3). Autorzy twierdzą,



Rysunek 5.3: Modyfikacja ilości warstw przy pomocy splotu 1×1 .

że taka operacja ma właściwości regularyzujące wagę oraz pomaga z problemem zanikającego gradientu występującego przy tak głębokich sieciach neuronowych. Te dodatkowe wyjścia nie biorą udziału w procesie testowania ani podczas pracy z danymi produkcyjnymi. Między warstwami konwolucyjnymi a gęstopołączonymi znajduje się pierwszy w tej pracy przykład użycia *AveragePooling*. Autorzy podają, że takie stosowanie *AveragePooling* zwiększył dokładność najlepszych predykcji o około 0.6%. Jako, że z racji wielkości nie sposób zamieścić wszystkich informacji na temat sieci w postaci znanych do tej pory z tej pracy diagramów, dokładne informacje na temat użytych warstw zawiera tabela na rysunku 5.4.

type	patch size/ stride	output size	depth	# 1×1	# 3×3 reduce	# 3×3	# 5×5 reduce	# 5×5	pool proj	params	ops
convolution	$7 \times 7 / 2$	$112 \times 112 \times 64$	1							2.7K	34M
max pool	$3 \times 3 / 2$	$56 \times 56 \times 64$	0								
convolution	$3 \times 3 / 1$	$56 \times 56 \times 192$	2		64	192				112K	360M
max pool	$3 \times 3 / 2$	$28 \times 28 \times 192$	0								
inception (3a)		$28 \times 28 \times 256$	2	64	96	128	16	32	32	159K	128M
inception (3b)		$28 \times 28 \times 480$	2	128	128	192	32	96	64	380K	304M
max pool	$3 \times 3 / 2$	$14 \times 14 \times 480$	0								
inception (4a)		$14 \times 14 \times 512$	2	192	96	208	16	48	64	364K	73M
inception (4b)		$14 \times 14 \times 512$	2	160	112	224	24	64	64	437K	88M
inception (4c)		$14 \times 14 \times 512$	2	128	128	256	24	64	64	463K	100M
inception (4d)		$14 \times 14 \times 528$	2	112	144	288	32	64	64	580K	119M
inception (4e)		$14 \times 14 \times 832$	2	256	160	320	32	128	128	840K	170M
max pool	$3 \times 3 / 2$	$7 \times 7 \times 832$	0								
inception (5a)		$7 \times 7 \times 832$	2	256	160	320	32	128	128	1072K	54M
inception (5b)		$7 \times 7 \times 1024$	2	384	192	384	48	128	128	1388K	71M
avg pool	$7 \times 7 / 1$	$1 \times 1 \times 1024$	0								
dropout (40%)		$1 \times 1 \times 1024$	0								
linear		$1 \times 1 \times 1000$	1							1000K	1M
softmax		$1 \times 1 \times 1000$	0								

Rysunek 5.4: Szczegółowe dane na temat GoogleLeNet zaprezentowane w ang. *Going deeper with convolutions* [6].

5.2. DeepDream i wizualizacje warstw ukrytych GoogleLeNet

5.2.1. Opis i zasada działania

DeepDream to program komputerowy stworzony przez Aleksandra Mordvintseva. Został opublikowany w 2015 roku i opisany w formie posta na blogu [8].

Co do zasady działania zarówno *Deepdream* jak i wizualizacje neuronów są modyfikacją programu zaprezentowanego w podrozdziale 4.1.1. Podobnie jak w wyżej wymienionym programie, używając wcześniej wytrenowanej sieci neuronowej modyfikujemy obraz by zmaksymalizować aktywację neuronów danej sieci konwolucyjnej.

Poza oczywistą zamianą VGG na GoogleLeNet, *DeepDream* stosuje znany już z podrozdziału 4.2 sposób obliczania aktywacji kilku neuronów na raz przy użyciu parametru stopnia użycia danej warstwy λ . Obliczając aktywację dla kilku warstw na raz uzyskuje się bogatsze w treść efekty wizualne. Podobnie jak w przypadku wizualizacji uzyskanych na VGG, w celu uzyskania obrazu, modyfikować wcześniejszy spreparowany, wylosowany obraz-szum. W swoim poście oryginalny autor posłużył się kolorowym obrazem (RGB). Ciekawe efekty uzyskano również przy pomocy modyfikacji już istniejących zdjęć. Tak samo jak poprzednio, obraz jest wielokrotnie skalowany podczas treningu, ale z racji swoich rozmiarów sieć GoogleLeNet jest w stanie uchwycić struktury nieuchwytne dla VGG co umożliwia generowanie nieco psychodelicznych obrazów.

5.2.2. Wizualizacja warstw ukrytych modelu GoogleLeNet

W przypadku wizualizacji samych warstw modelu GoogleLeNet zdecydowałem się na opublikowaną przez Google, opartą na *tensorflow*, bibliotekę *lucid* [9]. Jej głównym zastosowaniem jest wizualizacja neuronów, ale pozwala też m.in na *neural style transfer*, nawet na obiektach 3D.

Podobnie jak w przypadku wizualizacji na VGG, nie będę trenował sieci Incepcja samodzielnie, ponieważ taki trening trwałby bardzo długo. Zamiast tego załaduję model przygotowany przez autorów biblioteki. Podobnie jak w przypadku VGG model został wytrenowany na zbiorze *ImageNet*.

```
1 model = models.InceptionV1()  
2 model.load_graphdef()
```

Listing 5.1: Wczytywanie modelu GoogleLeNet w *lucid*.

Po wczytaniu modelu jedyne co pozostało to wybrać warstwy które chcemy aktywować i możemy generować wizualizacje.

```
1 channel = lambda n: objectives.channel("mixed4a_pre_relu", n)
2 pf = lambda: param.image(256)
3 obj = channel(122) + channel(155)
4 _ = render.render_vis(model, param_f = pf, objective_f = obj)
```

Listing 5.2: Generowanie wizualizacji neuronów przy użyciu *lucid*.

Listing 5.2.2 jest przykładem implementacji wizualizacji aktywacji filtrów 122 i 155 warstwy *mixed4a_pre_relu*. Efekty skryptu można zobaczyć na rysunku 5.6. Przykładowe wizualizacje innych filtrów i warstwy pokazuje rysunek 5.2.2.

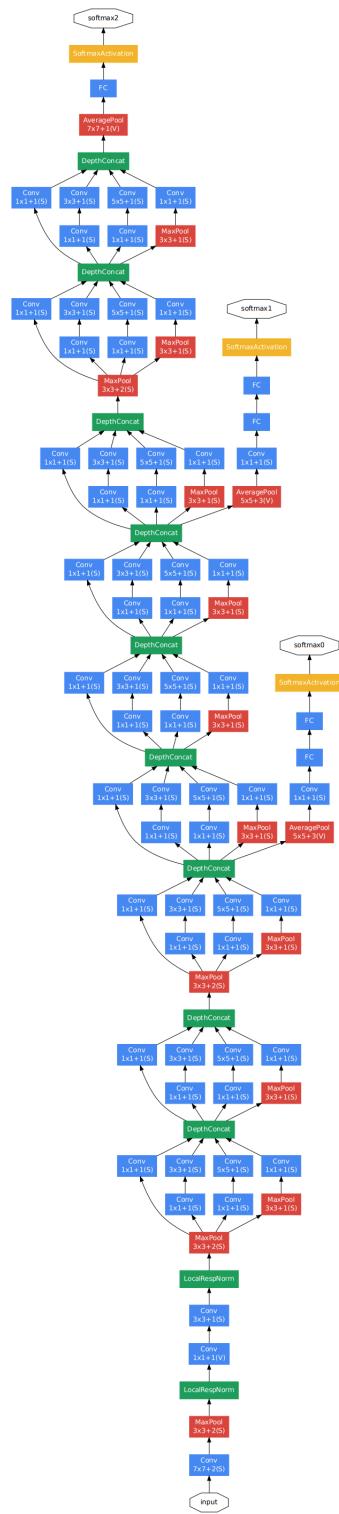
Na załączonych wizualizacjach widać potencjał *lucid* jako biblioteki. Uzyskane wyniki pokazują w jaki sposób kolejne filtry oddziaływują na otrzymywany obraz - opisują przez to interakcje między neuronami.

Ilość kombinacji między warstwami czy filtrami niestety nie pozwoli na zamieszczenie podobnego przeglądu aktywacji warstw jaki został zamieszczony w przypadku sieci VGG.

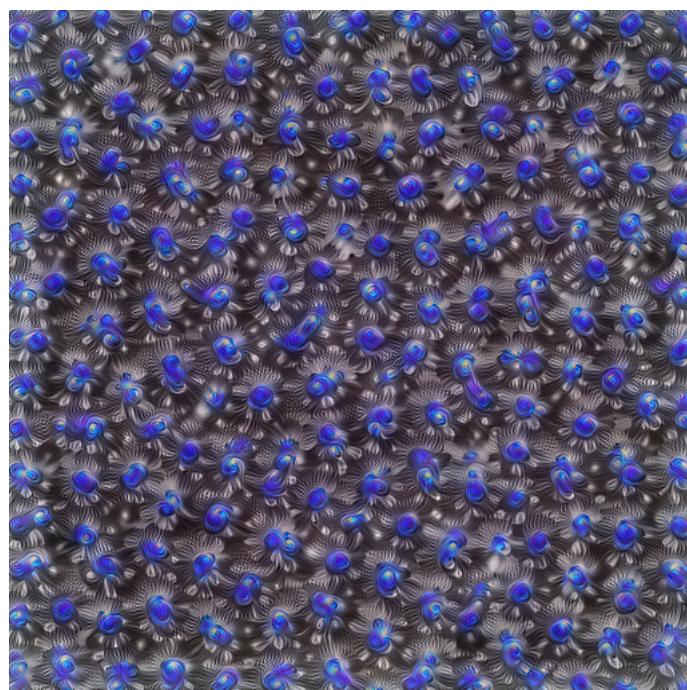
5.2.3. Wizualizacje uzyskane przy pomocy *DeepDream*

Zgodnie z opisem z podrozdziału 5.2.1 prezentowane wizualizacje zostały otrzymane poprzez modyfikację skryptu wizualizującego VGG. W procesie modyfikacji obrazu biorą 4 warstwy konwolucyjne (mixed od 2 do 4). Największy wpływ na funkcję kosztu mają 2 ostatnie warstwy. By uzyskać bardziej widoczny efekt należy modyfikować orginalny obraz przez większą ilość iteracji, jest to jednak kosztowne obliczeniowo.

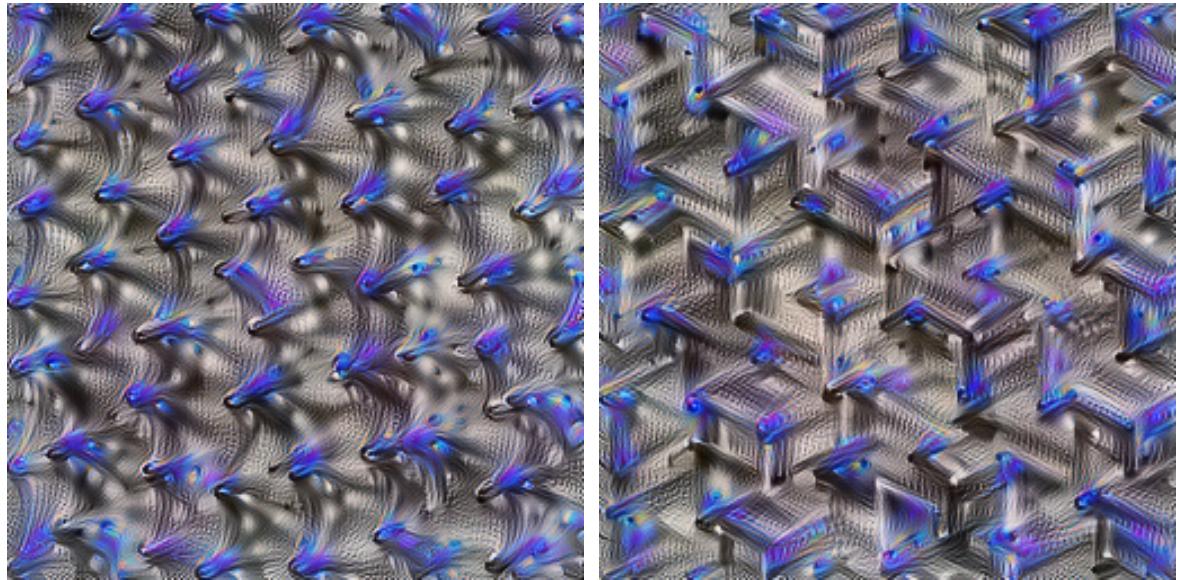
Uzyskane efekty zostały zaprezentowane na rysunku 5.2.3.



Rysunek 5.5: Schemat sieci GoogleLeNet zaprezentowany w ang. *Going deeper with convolutions* [6].



Rysunek 5.6: Wizualizacja aktywacji filtrów 122 i 155 warstwy *mixed4a_pre_relu*.

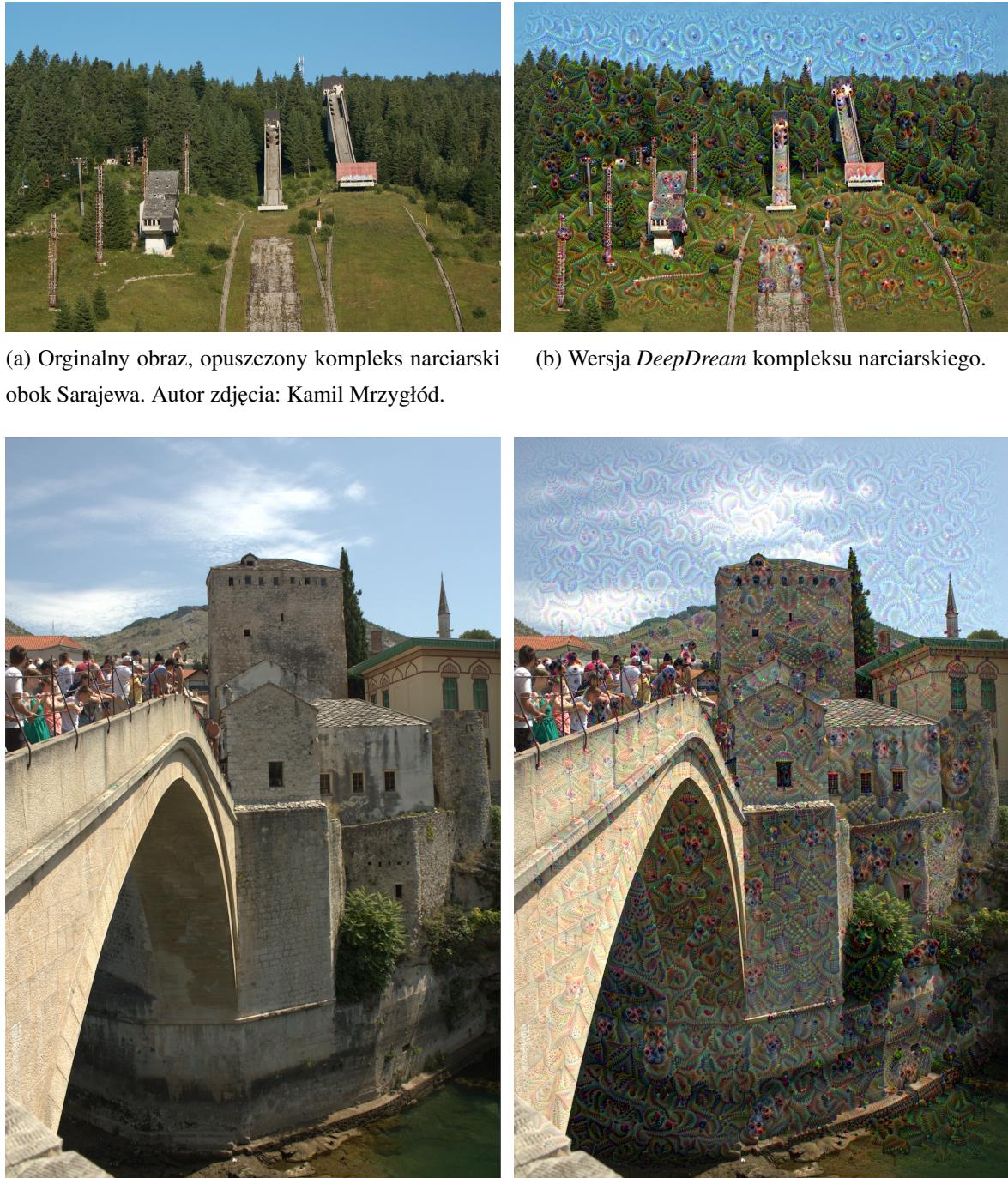


(a) Wizualizacja filtrów 122, 155 i 255 warstwy *mixed4a_pre_relu*. (b) Wizualizacja filtrów 122, 155, 255 i 455 warstwy *mixed4a_pre_relu*.



(c) Wizualizacja filtra 1 warstwy *mixed3a*. (d) Wizualizacja filtrów 1 i 102 warstwy *mixed3a*. (e) Wizualizacja filtrów 1, 102 i 203 warstwy *mixed3a*.

Rysunek 5.7: Przykładowe wizualizacje neuronów sieci GoogleLeNet.



(c) Oryginalny obraz, most w miejscowości Mostar.
Autor zdjęcia: Kamil Mrzygłód.

(d) Wersja DeepDream mostu w Mostarze.

Rysunek 5.8: Przykładowe wizualizacje neuronów sieci GoogleLeNet.

6. Podsumowanie

Niewątpliwym postępem w dziedzinie uczenia maszynowego i głębokich sieci konwolucyjnych doprowadził do przełomu na wielu płaszczyznach. Najnowsze doniesienia z branży, na które za-brakło niestety miejsca w tej pracy dotyczą przykładowo automatycznego generowania, nigdy wcześniejszej nie istniejących, realistycznych ludzkich twarzy. Dalszy rozwój tej dziedziny niewątpliwie doprowadzi do automatyzacji procesów, w których obecnie udział człowieka może wydawać się niezbędny. Jesteśmy dopiero na początku tego procesu co sprawia, że możemy spodziewać się kolejnych przełomowych osiągnięć w kolejnych latach.

Postęp w rozwoju sieci konwolucyjnych jest możliwy dzięki ich lepszemu zrozumieniu. Wizualizacje działania sieci neuronowych ułatwiają przyswajanie wiedzy na ich temat oraz często prowadzą do wynajdywania nowych dla nich zastosowań. Widzą to zarówno pracujący na uczelniach naukowcy i szerogowi pracownicy firm IT. Wspaniale jest widzieć, jak przy współpracy uczelnii i firm powstają takie narzędzia jak *lucid*. Świecone jest to, że uczelnie udostępniają wytrenowane wagi sieci neuronowych tak by każdy mógł dokonywać na nich swoich eksperymentów.

Oprócz twórczego aspektu, takie eksperimentowanie naprowadza nas na kolejne, lepsze architektury konwolucyjnych sieci neuronowych. O ile w przypadku analizy języka naturalnego, już dziś posiadamy odpowiednią ilość danych w stosunku do wydajności tam stosowanych architektur, tak w przypadku analizy obrazu przy użyciu sieci neuronowych wciąż jest ogromne pole do zagospodarowania dla innowacji.

Uważam, że uzyskane wizualizacje obok bezpośrednich efektów dydaktycznych dla mnie, były świetną okazją by zaznajomić się z dalszą teorią stojącą za sieciami neuronowymi i mam nadzieję na dalszy swój rozwój w tej dziedzinie.

Bibliografia

- [1] Yann LeCun, Corinna Cortes i Christopher J.C. Burges. *Baza danych zawierająca ręcznie pisane cyfry MNIST*. URL: <http://yann.lecun.com/exdb/mnist/>.
- [2] Karen Simonyan i Andrew Zisserman. *Very deep convolutional network for large-scale image recognition*. URL: <https://arxiv.org/pdf/1409.1556.pdf>.
- [3] <http://image-net.org/about-people>. *Zbiór danych imangenet*. URL: <http://www.image-net.org/>.
- [4] Leon A. Gatys, Alexander S. Ecker i Matthias Bethge. *A Neural Algorithm of Artistic Style*. URL: <https://arxiv.org/abs/1508.06576>.
- [5] Michael Goldberg. *The keep*. URL: <http://www.artnet.com/artists/michael-goldberg/the-keep-OATwKGwjhcYzfjurZZ7R4w2>.
- [6] Christian Szegedy i in. *Going Deeper with Convolutions*. URL: <https://arxiv.org/abs/1409.4842>.
- [7] Kaiming He i in. “Deep Residual Learning for Image Recognition”. W: *CoRR* abs/1512.03385 (2015). arXiv: 1512 . 03385. URL: <http://arxiv.org/abs/1512.03385>.
- [8] Alexander Mordvintsev, Christopher Olah i Mike Tyka. *Inceptionism: Going Deeper into Neural Networks*. URL: <https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>.
- [9] <https://github.com/tensorflow/lucid/graphs/contributors>. *Repozytorium z biblioteką lucid*. URL: <https://github.com/tensorflow/lucid>.