AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA Infrormatyki

PROJEKT INŻYNIERSKI

Rozpoznawanie ekspresji twarzy w oparciu o sieci neuronowe konwolucyjne

Dokumentacja produktowa

Autorzy: Jacek Obłaza, Dmytro Petruk
Kierunek studiów: Informatyka
Opiekun pracy: Dr hab. Bogdan Kwolek

Kraków, 2016

Spis treści

1. Dziedzina problemu

Rozpoznawanie obrazów, a w szczególności wspólnych cech i podobieństw jest dosyć nowym zagadnieniem w informatyce. O ile samo rozpoznawanie i klasyfikowanie prostych obrazów takich jak cyfry było możliwe w latach 90tych, tak klasyfikacja obrazu twarzy i przypisania do niej emocji jest możliwa dopiero od około 2003 roku kiedy to zaczęły być rozwijane konwolucyjne sieci neuronowe.

2. Sieci konwolucyjne

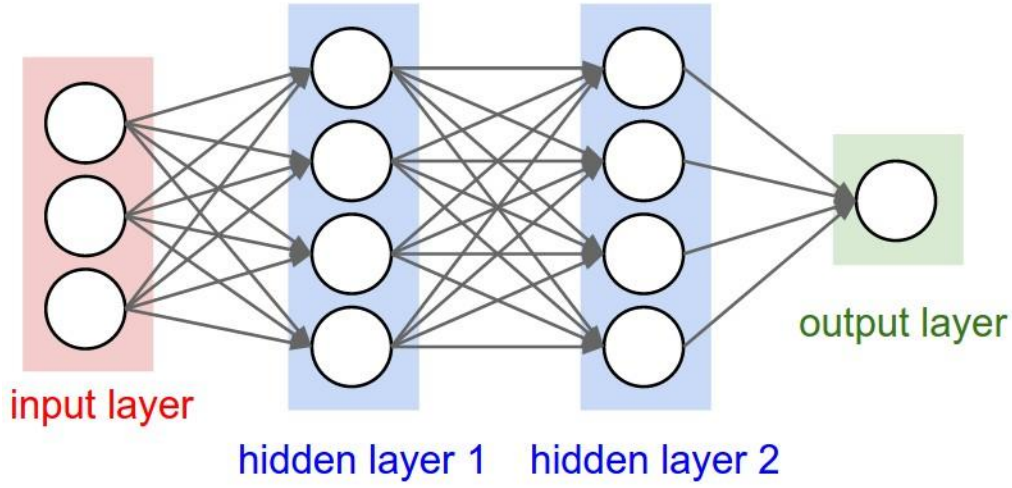
Koncepcyjnie, konwolucyjne sieci neuronowe działają w sposób bardzo podobny do zwykłych sieci neuronowych (Multilayer Perceptron, MLP). Każdy neuron ma swoje wejścia, oblicza ich iloczyn skalarny i opcjonalnie dodaje nieliniowość. Cała sieć nadal reprezentuje pojedynczą, różniczkowalną funkcję oceny: od surowych pikseli obrazu na jednym końcu do oceny klasy na drugim końcu. Na ostatniej warstwie taka sieć zwykle też zawiera loss-function (w fully-connected layer).

2.1. Różnica pomiędzy MLP i CNN

2.1.1 Topologia

Zwykły wielowarstwowy perceptron nie skaluje się dobrze podczas pracy z obrazami. Biorąc do przykładu zbiór CIFAR-10 [1], obrazy tam są wymiaru tylko 32x32x3 (32 - szerokość, 32 - wysokość, 3 - ilość kanałów koloru), więc pojedynczy neuron w pełnopołączeniowej sieci, w pierwszej warstwie ukrytej będzie miał 32*32*3 = 3072 wag. Taka liczba połączeń nie jest jeszcze na tyle duża, żeby sieć miała jakiekolwiek problemy z uczeniem się, ale łatwo zauważyć, że struktura pełnopołączeniowa nie skaluje się dobrze dla większych obrazów. Oczywistym jest to, że pełne połączenia są nieoptymalne i ogromna ilość parametrów szybko spowoduje przetrenowanie sieci (overfitting).

Z drugiej strony, CNN wykorzystują fakt, że wejście składa się z obrazów i to wymusza pewną, charakterystyczną architekturę. W odróżnieniu od MLP, warstwy sieci konwolucyjnych zawierają neurony w trzech wymiarach: szerokość, wysokość i głębokość. Na przykład, wejściowe obrazy w CIFAR-10 mają wymiary 32x32x3 (szerokość, wysokość i głębokość). Jak zostanie pokazane później, neurony w warstwie będą połączone tylko z małym regionem warstwy poprzedzającej, w odróżnieniu od neuronów w MLP, połączonych ze wszystkimi wcześniejszymi. Ponadto wynikowa warstwa wyjściowa dla CIFAR-10 będzie miała wymiary 1x1x10, ponieważ w ostatniej warstwie sieci konwolucyjnej pełny obraz zostanie zredukowany do pojedynczego wektora z nasyceniem klasami.



Rys. 1. Topologia sieci neuronowej

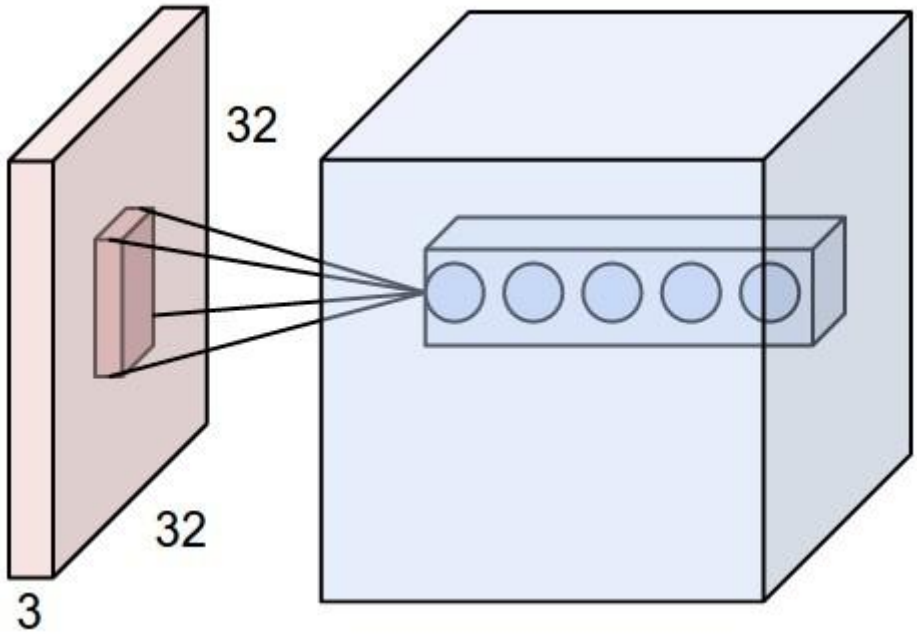
2.2 Warstwy CNN

Prosta sieć konwolucyjna jest sekwencją warstw, gdzie każda warstwa dokonuje transformacji jednej wymiarowości aktywacji do innej poprzez funkcją różniczkowalną. Zwykle są używane 3 podstawowe typy warstw dla budowania sieci konwolucyjnych: *convolutional layer*, *pooling layer* i *fully-connected layer* (taka sama jak jest używana w zwykłych sieciach neuronowych). Odpowiednie połączenie tych warstw skutkuje powstaniem sieci neuronowej konwolucyjnej.

2.2.1. Convolutional layer

Jest podstawową warstwą CNN, która wykonuje większość obliczeń. *Convolutional layer* składa się ze zbioru uczących się filtrów. Każdy filtr ma małe rozmiary (szerokość i wysokość) w porównaniu do obrazów wejściowych. Mimo, że filtr ma tylko 2 wymiary, to w pełni używa głębokości wejścia. Na przykład, typowym filtrem dla pierwszej *convolutional layer* jest rozmiar 5x5. Przy przejściu w przód (forward pass), my przesuwamy filtrem po całym obrazie wejściowym (wzdłuż i wszerz) i liczymy iloczyn skalarny z wartości danej komórki filtra i odpowiadającą jej komórką wejścia. W miarę przesuwania filtra po obrazie tworzona jest dwuwymiarowa mapa aktywacyjna (activation map), która jest reakcją konkretnego filtru na każdą rozpatrywaną pozycję. Intuitywnie, sieć nauczy się filtrów kiedy są one aktywowane widząc pewne cechy wizualne, takie jak krawędzi o pewnej orientacji, plamy pewnego koloru na pierwszej warstwie, lub bardziej złożone elementy na wyższych warstwach sieci (nos, ucho, oko).

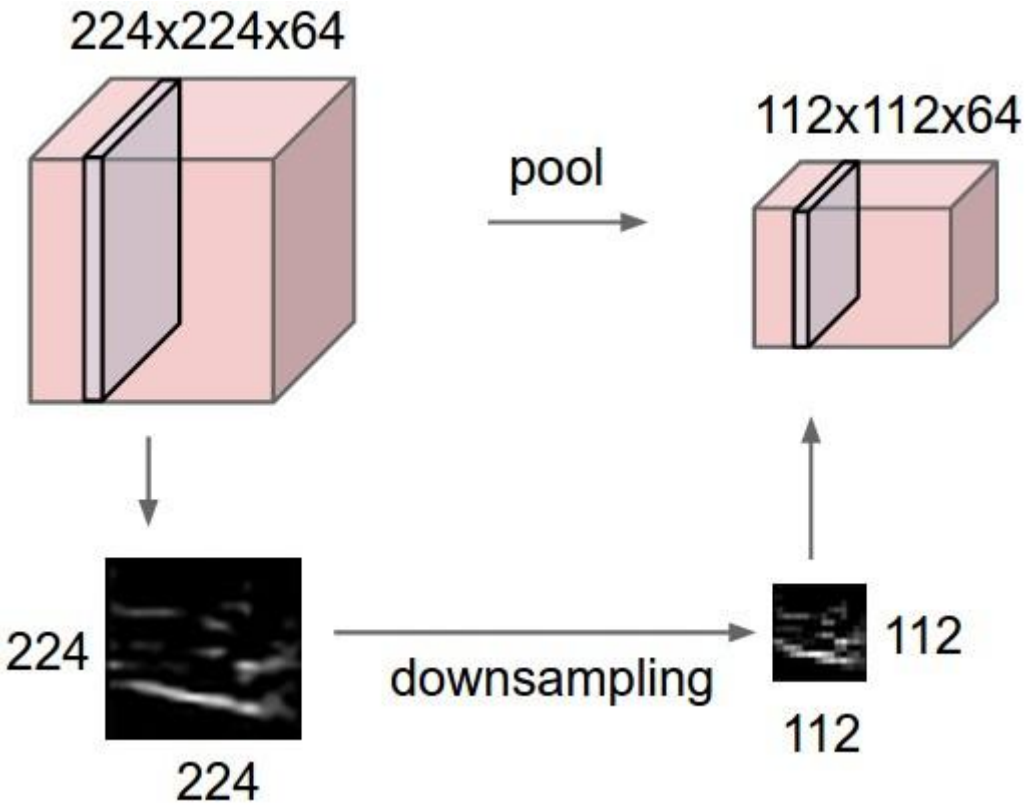
Mając cały zestaw filtrów na każdej *convolutional layer* (na przykład 32 filtry) każdy z nich wytworzy inną mapę aktywacyjną. Nakładając takie mapy aktywacyjne na siebie będziemy mieli wejście (feature map).



Rys. 2. Convolutional Layer

2.2.2. Pooling Layer

Powszechnie stosowanym zabiegiem jest wstawianie *pooling layer* pomiędzy warstwy konwolucyjne. Służy on do redukcji wymiarowości oraz zmniejszenia liczby parametrów. Zapobiega to również przeuczeniu sieci. *Pooling layer* operuje niezależnie na każdej mapie aktywacji wejścia i redukuje jej wymiarowość, zwykle wykorzystując operator MAX. Najbardziej powszechnie używaną formą jest *pooling layer* z rozmiarem filtru 2x2 i rozmiarem stride (odległość, na którą przesuwamy każdorazowo filtr) - 2. Tym sposobem usuwamy 75% aktywacji. Każda operacja MAX zwróci maksymalną wartość z 4-ch liczb (region 2x2 na każdym z poziomów feature map).



Rys. 3. Działanie pooling layer

2.2.3. Fully-connected Layer

Neurony w *fully-connected layer* mają połączenia z wszystkimi neuronami w warstwie poprzedzającej, tak jak to działa w zwykłych sieciach neuronowych. Ich aktwycje mogą być wyliczone poprzez mnożenie macierzy z przesunięciem o bias.

3. Stosowane narzędzia

3.1. Język programowania

Całość kodu projektu napisana jest w języku **Python**. Jest on prosty, szybko się w nim tworzy małe projekty oraz pozwala na szybkie prototypowanie. Co jednak było najważniejsze większość narzędzi do tworzenia sieci neuronowych jest właśnie dla niego stworzona.

Dużym minusem mogło okazać się jego wolne działanie, jednak jest ono wystarczająco szybkie dla potrzeb tego projektu.

3.2. Narzędzie do tworzenia sieci neuronowych

Na rynku znajduje się wiele narzędzi pozwalających na zbudowanie, wytrenowanie i testowanie własnej sieci neuronowej. Wiele z nich umożliwia także tworzenie głębokich sieci neuronowych, w tym sieci konwolucyjnych. Biblioteki brane pod uwagę podczas tworzenia tego projektu:

1) **TensorFlow** [2]
2) **Theano** [3]
3) **Caffe** [4]
4) **Lasagne** [5]
5) **Keras** [6]

TensorFlow i Theano pozwalają na wydajną definicję, optymalizację i ewaluację wyrażeń matematycznych używających wielowymiarowych macierzy. Obie pozwalają na współbieżne trenowanie sieci z wykorzystaniem GPU. TensorFlow do tego pozwala w prosty sposób wizualizować sieć za pomocą TensorBoard. Są dobrymi narzędziami do budowania całego projektu od podstaw. Ich niskopoziomowość jest ich dużym atutem, jak i dużą wadą ze względu na złożoność procesu tworzenia sieci neuronowej.

Caffe jest bardzo potężnym narzędziem, jednak po wstępnych testach i implementacjach przykładowych sieci konwolucyjnych okazało się, że jej API jest nieoczywiste. Tak samo jak poprzednie biblioteki pozwala na uczenie sieci przy pomocy GPU.

Keras i Lasagne są do siebie podobne pod względem API i możliwości. Wybór między nimi nie był jednak trudny. Obie tak samo są nakładką na Theano, która sprawia, że staje się ono wyskopoziomowym narzędziem. Do Lasagne można jednak znaleźć więcej poradników, przykładowych sieci i pomocy w sieci, co zaważyło o wyborze tej właśnie biblioteki.

3.3. Narzędzie do rozpoznawania obrazów

Przed przystąpieniem do trenowania i ewaluacji sieci należy przygotować obraz wejściowy. Zdjęcia czegoś więcej niż samej twarzy wprowadziłyby niepotrzebny element zaburzający wyniki klasyfikacji. Mogłoby się okazać, że sieć klasyfikuje emocję jako strach, jeśli na obrazie poza twarzą występuje jakiś inny konkretny obiekt. Aby ograniczyć obraz do samej twarzy należy najpierw ją na nim znaleźć, do czego posłużyła bilioteka **OpenCV**

OpenCV to biblioteka funkcji wykorzystywanych podczas obróbki obrazu, zapoczątkowana przez Intela. Biblioteka ta jest wieloplatformowa, można z niej korzystać zarówno na Mac OS X, Windows, jak i Linux. Jej autorzy skupiają się na przetwarzaniu obrazu w czasie rzeczywistym.

Z tej biblioteki był użyty głównie mechanizm rozpoznawania twarzy algorytmem Viola-Jones, który działa na podstawie cech kaskad Haar'a i został opisany w artykule z 2001 roku ("Rapid Object Detection using a Boosted Cascade of Simple Features" [7]). Jest bazowany na metodach uczenia maszynowego, gdzie funkcja kaskad jest trenowana na mnóstwie obrazów pozytywnych i negatywnych (obrazy pozytywne - zawierają szukany objekt, a negatywne - nie zawierają). Następnie jest wykorzystany do detekcji obiektu na nowych obrazach. OpenCV zawiera już wytrenowaną konfigurację detektora, więc została ona użyta do detekcji twarzy. [8]

4. Moduły

4.1. Moduł przygotowywania danych wejściowych

Moduł odpowiedzialny za przygotowanie danych wejściowych do sieci. Jego zadaniem jest wczytanie obrazu z podanej ścieżki, przekształcenie go do skali szarości, wykrycie na nim twarz, żeby w końcu przeskalować sam fragment zawierający twarz do macierzy rozmiaru 96x96x1.

Wszystkie przekształcenia oraz wczytanie obrazu realizowane jest przy pomocy wspomnianej biblioteki OpenCV. Obraz po wczytaniu reprezentowany jest przez macierz jego pikseli, więc te operacje są prostymi przekształceniami matematycznymi. Wykrywanie twarzy na obrazie realizowane jest przy pomocy kaskad Haar'a z tej samej biblioteki. Ważne jest, aby dostarczyć odpowiedni plik xml z zapisanymi parametrami kaskad. Założenia są takie, że plik ten znajduje się w tym samym katalogu, co moduł, a jego nazwa to "haarcascade_frontalface_default.xml".

Moduł ten udostępnia też funkcję wczytania danych do trenowania sieci. Funkcja ta przyjmuje 2 argumenty: datadir i labeldir. Pierwszy to ściezka do folderu z obrazami w formacie jpg, a drugi to ścieżka do podpisów każdego z najgłębszego podkatalogu z obrazami. Struktura katalogów w obu tych ścieżkach musi być identyczna. Obrazy, które nie mają odpowiadającego katalogu w drugiej ścieżce nie zostają wczytane.

4.2. Moduł tworzenia architektury sieci

Moduł służący do przygotowania odpowiedniej struktury sieci. Potrzebne jest jego wywołanie zarówno kiedy chcemy trenować sieć, jak też użyć jej do klasyfikacji przynajmniej jednego obrazu.

Wykorzystuje on bibliotekę Lasagne i Theano do stworzenia architektury sieci. Pozwala na stworzenie architektury sieci niewytrenowanej lub, jeśli za argument funkcji tworzącej poda się nazwę pliku, który przechowuje informacje o wagach krawędzi w sieci, utworzona zostanie sieć wytrenowana, której można używać do rozpoznawania ekspresji twarzy.

4.3. Moduł trenowania sieci

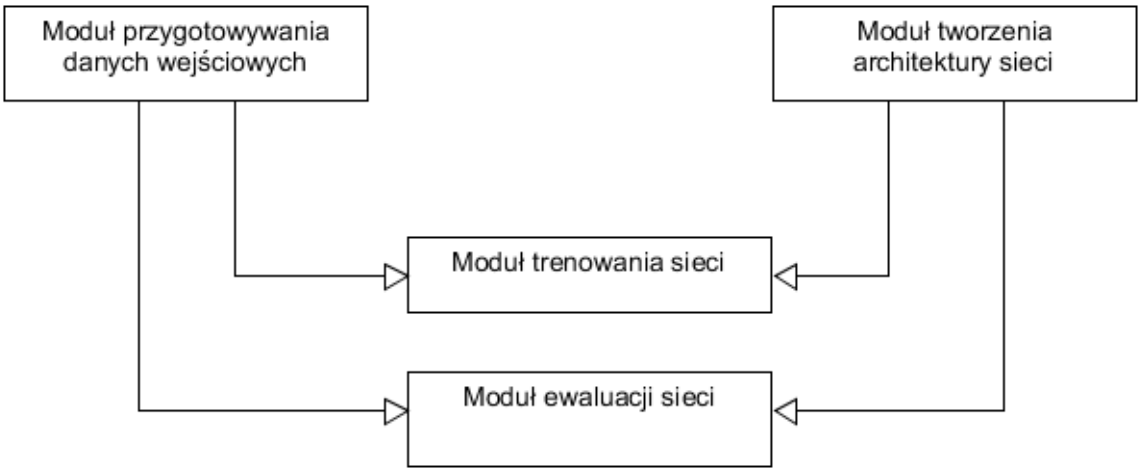
W tym module przeprowadzane jest trenowanie sieci. Mając już przygotowane dane wejściowe i sieć przez poprzednie moduły możemy przeprowadzić jej trenowanie. Trenowanie podzielone jest na epoki, których liczbę można podać jako argument funkcji trenującej. Jeżeli nie zostanie podany, funkcja będzie wykonywać się, dopóki różnica wartości funkcji błędu dla dwóch kolejnych epok nie będzie większa niż 0.0001.

Wytrenowana sieć zapisywana jest do pliku model.npz. Z tego pliku można wczytać parametry sieci i odtworzyć ją za pomocą tego modułu. Można też ustawić, aby zapisywanie odbywało się co każdą epokę. Zapobiega to utraceniu danych podczas nagłego przerwania procesu trenowaniu i umożliwia dalszy trening na podstawie zapisanych danych.

4.4. Moduł ewaluacji sieci

Moduł odpowiedzialny za przeprowadzenie klasyfikacji podanego przez nas obrazu w oparciu o wcześniej wytrenowaną sieć. Na wejściu przyjmuje obraz zawierający twarz, dla której chcemy rozpoznać ekspresję, a zwraca poziomy nasycenia każdą ze znanych mu ekspresji. Jest to najmniejszy, ale zarazem najczęściej używany moduł.

4.5. Relacje między modułami



Rys. 4. Relacje między modułami

Modułowi trenowania sieci potrzebne są zbiór treningowy, który dostaje od *modułu przygotowywania danych wejściowych*, oraz architektura sieci, którą dostarcza *moduł tworzenia architektury sieci*.

Moduł ewaluacji sieci potrzebuje pojedynczego obrazu, dostarczanego przez *moduł przygotowywania danych wejściowych*, i architektury wytrenowanej sieci, którą dostaje od *modułu tworzenia architektury sieci.*

5. Schemat wykrywania ekspresji twarzy

Rozpoznawania ekspresji na obrazie podzielone jest na następujące kroki:

1) Znalezienie wszystkich twarzy na danym obrazie
2) Przeskalowanie każdej znalezionej twarzy do ustalonego rozmiaru
3) Podanie na wejście sieci konwolucyjnej każdej twarzy
4) Otrzymanie stopni nasycenia twarzy każdą z ekspresji

1) **Znalezienie wszystkich twarzy na danym obrazie**

Do tego celu użyliśmy gotowej implementacji klasyfikatora kaskad Haar'a z biblioteki OpenCV. Klasyfikator ten przeprowadza analizę każdej części obrazu za pomocą przesuwanego okna i klasyfikuje ją jako zawierającą twarz bądź nie. Twarze nie zawsze są tej samej wielkości, więc algorytm musi wiele razy przeskanować obraz za każdym razem oknem innej wielkości.

2) **Przeskalowanie każdej znalezionej twarzy do ustalonego rozmiaru**

Klasyfikator z punktu 1) zwraca położenie prostokąta opisanego na znalezionej twarzy. Jest on różnych rozmiarów dla różnych obrazów i twarzy, a sieć neuronowa ma ustaloną ilość neuronów wejściowych. Co za tym idzie nie możemy podać na wejście sieci neuronowej obrazu innej wielkości niż tej, dla której sieć została nauczona. Obraz nie może też być za duży, aby proces trenowania i klasyfikowania nie był zbyt duży. Ustaliliśmy stały rozmiar obrazów twarzy na 96x96 px. Jest on jeszcze wystarczająco duży, aby człowiek był w stanie określić ekspresję twarzy.

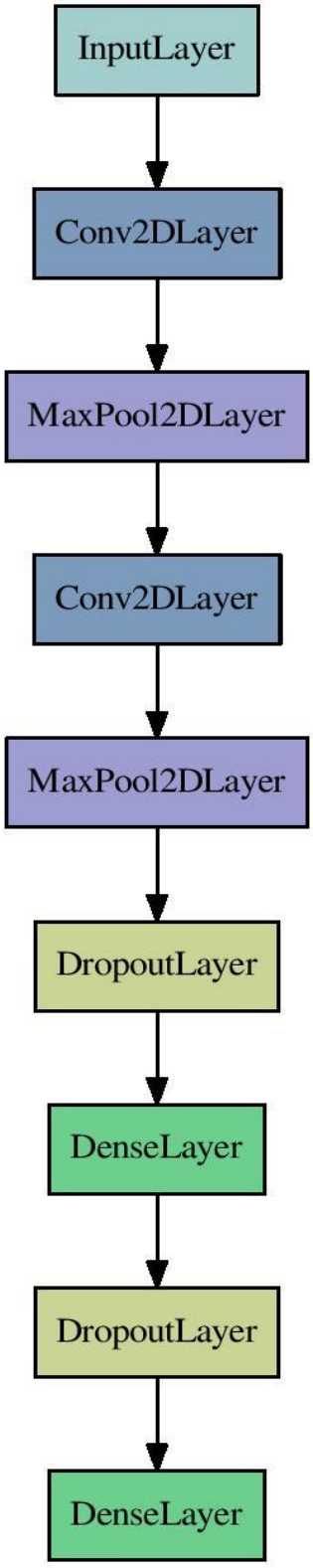
3) **Podanie na wejście sieci konwolucyjnej każdej twarzy**

Posiadając wytrenowaną sieć musimy podać na jej wejście macierz pikseli danej twarzy. Sieć następnie rozpocznie obliczenia dla każdej wartości z macierzy wejściowej.

4) **Otrzymanie stopni nasycenia twarzy każdą z ekspresji**

Wyniki działania sieci są przedstawiane w postaci macierzy zawierającej stopnie nasycenia twarzy daną emocją. Macierz ta jest jednowymiarowa, a każdy kolejny rząd odpowiada odpowiedniej emocji. 1 - gniew, 2 - pogarda, 3 - zniesmaczenie, 4 - strach, 5 - radość, 6 - smutek, 7 - zaskoczenie. Wszystkie warości są z przedziału 0 - 1, a ich suma wynosi 1.

Przykładowa macierz wynikowa: [0.0, 0.0, 0.01, 0.04, 0.0, 0.78, 0.0, 0.17]. Wynik ten jest równoznaczny z rozpoznaniem radości na danej twarzy.



Rys. 5. Topologia sieci konwolucyjnej

- Input Layer (InputLayer). Warstwa wejściowa. Jej neurony są reprezentacją obrazu wejściowego w postaci macierzy pikseli.

- Convolutional Layer (Conv2DLayer). Składa się z ustalonej ilości warstw, a dane przepuszczane są przez filtr o ustalonym rozmiarze, który wykonuje na nich konwolucję. Każda następna warstwa konwolucyjna powinna wykrywać bardziej wysokopoziomowe właściwości.

- Pooling Layer (MaxPool2DLayer). Służy do redukcji wymiarowości;

- Dropout (DropoutLayer). Warstwa, która usuwa część połączeń pomiędzy neuronami. Zapobiega to przeuczeniu sieci;

- Fully connected. Ma połączenia ze wszystkimi aktywacjami sieci z warstwy poprzedniej, tak jak i w zwykłym perceptronie.

Dokładniejszy opis wraz z parametrami znajduje się w dalszej części dokumentu

6. Zbiór treningowy

Do przeprowadzenia procesu uczenia sieci neuronowej konwolucyjnej niezbędny jest duży (rzędu paru tysięcy) zbiór danych. Znaleźliśmy zbiór, który odpowiada naszym potrzebom. Można go pobrać ze strony: http://www.consortium.ri.cmu.edu/ckagree/ . Znajduje się w nim około 10tys. zdjęć twarzy z podpisanymi na nich ekspresjami. Jednak nie wszystkie nadawały się do zastosowania w naszej sieci. Zbiór ten zawiera zdjęcia zmiany mimiki twarzy osób od ekspresji neutralnej, po zadaną. Dla każdej osoby jest to od kilku do kilkunastu zdjęć. Sieć neuronowa powinna dostawać tylko skrajne przypadki, więc dla każdego zbioru wzięliśmy tylko ostatnie 30% zdjęć. Dodając do tego fakt, że nie wszystkie zbiory okazały się opisane dało nam to 2200 nadających się zdjęć.

7. Tworzenie architektury sieci

Lasagne pozwala na proste definiowanie architektury sieci. W naszym przypadku sieć miała następującą postać:

1) *network = lasagne.layers.InputLayer(shape = (None, 1, 96, 96),*
input_var=input_var)

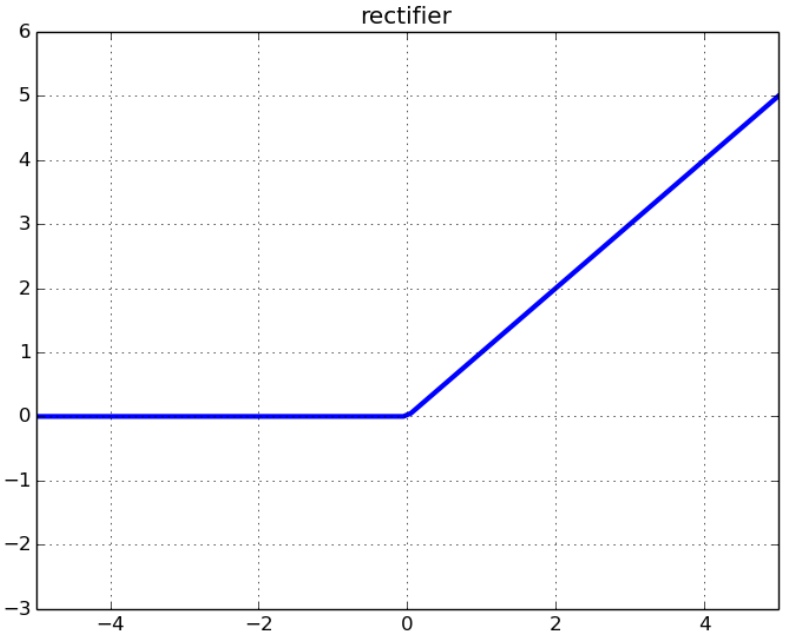
Warstwa wejściowa, z prostym mapowaniem jeden neuron - jeden piksel na obrazie wejściowym

2) *network = lasagne.layers.Conv2DLayer(*
network, num_filters=32, filter_size=(5, 5),
nonlinearity=lasagne.nonlinearities.rectify)

Pierwsza warstwa konwolucyjna. Ustawiamy rozmiar filtru jako macierz 5x5 i definiujemy że chcemy 32 filtry, więc wymiar na wyjściu tej warstwy będzie 92x92x32. 92 bierze się z tego, że biorąc tylko poprawne konwolucje (nie dopełniając zerami na brzegach - nie używamy zero-padding) otrzymujemy 92 przejścia. 32 macierze o rozmiarach 92x92 będą tworzyły tak zwany feature map.

Na wejściu tej warstwy dla każdego neuronu zastosowano funkcję ReLU, definiowaną jako:

ReLU(x) = max(0, x)

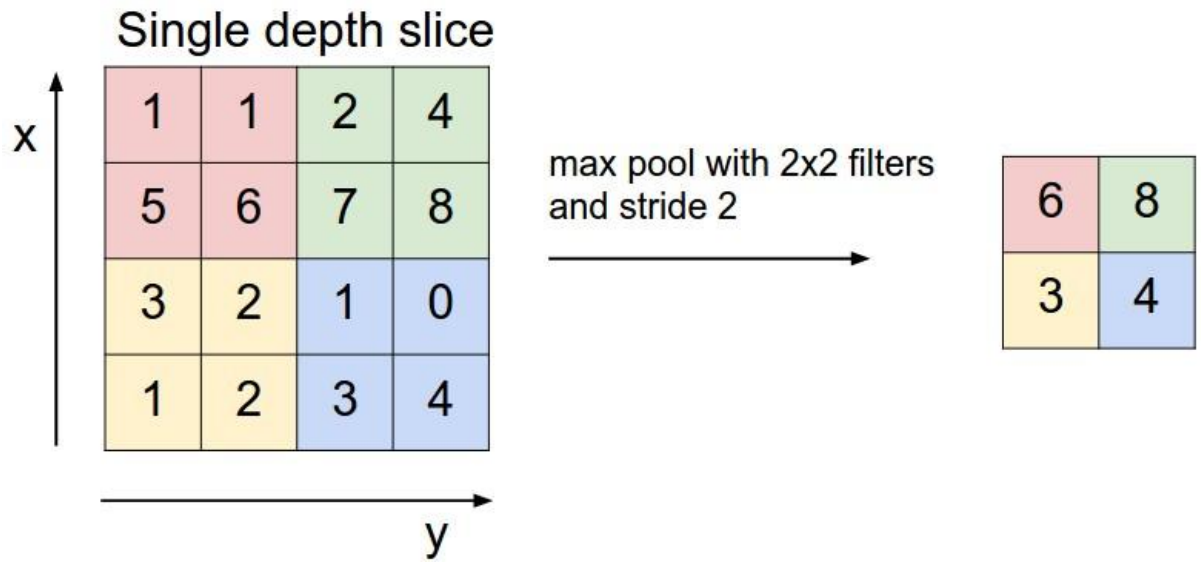


Rys. 6. Wykres funkcji ReLU

Użyto ReLU zamiast funkcji sigmoidalnej, ponieważ jest o wiele prostsza w obliczeniu i jej gradient nie zanika ze wzrostem/spadkiem x. Jak pokazano w pracy "ImageNet Classification with Deep Convolutional Neural Networks" [9] proces uczenia sieci neuronowej z zastosowaną funkcją ReLU przebiega kilka razy szybciej niż z innymi popularnymi funkcjami.

3) *network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2, 2))*

Pierwsza warstwa Pooling'owa. Używamy tu filtru o rozmiarze 2x2, stride (na jaką odległość przesuwamy filtr w każdym kroku w odpowiednim kierunku) o wartości 2 i funkcji max, która wyznacza wyjściowe wyniki. Ta warstwa służy do redukcji wymiarowości.



Rys. 7. Przykład działania pooling'u

4) *network = lasagne.layers.Conv2DLayer(*
network, num_filters=32, filter_size=(5, 5),
nonlinearity=lasagne.nonlinearities.rectify)

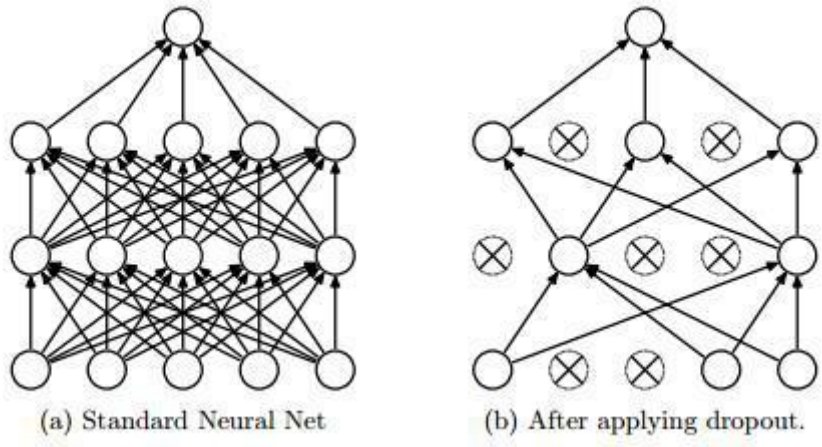
Jeszcze jedna warstwa konwolucyjna z takimi samymi parametrami jak poprzednia.

5) *network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2, 2))*

Kolejna warstwa służąca do redukcji wymiarowości danych o takich samych parametrach.

6) *network = lasagne.layers.DenseLayer(*
lasagne.layers.dropout(network, p=.5),
num_units=256,
nonlinearity=lasagne.nonlinearities.rectify)

Tworzymy jedną warstwę z dropout'em. Jest to technika, opisana w artykule "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" [10]. Pozwala ona na zmniejszenie prawdopodobieństwa przeuczenia sieci, które następuje poprzez zapamiętywanie dokładnego stanu neuronów wejściowych. Ta technika polega na "wyłączeniu" części neuronów w sposób losowy (w naszym przypadku prawdopodobieństwo wyłączenia neuronu wynosi 50%).



Rys. 8. Różnica między warstwą z dropout'em a bez dropout'u

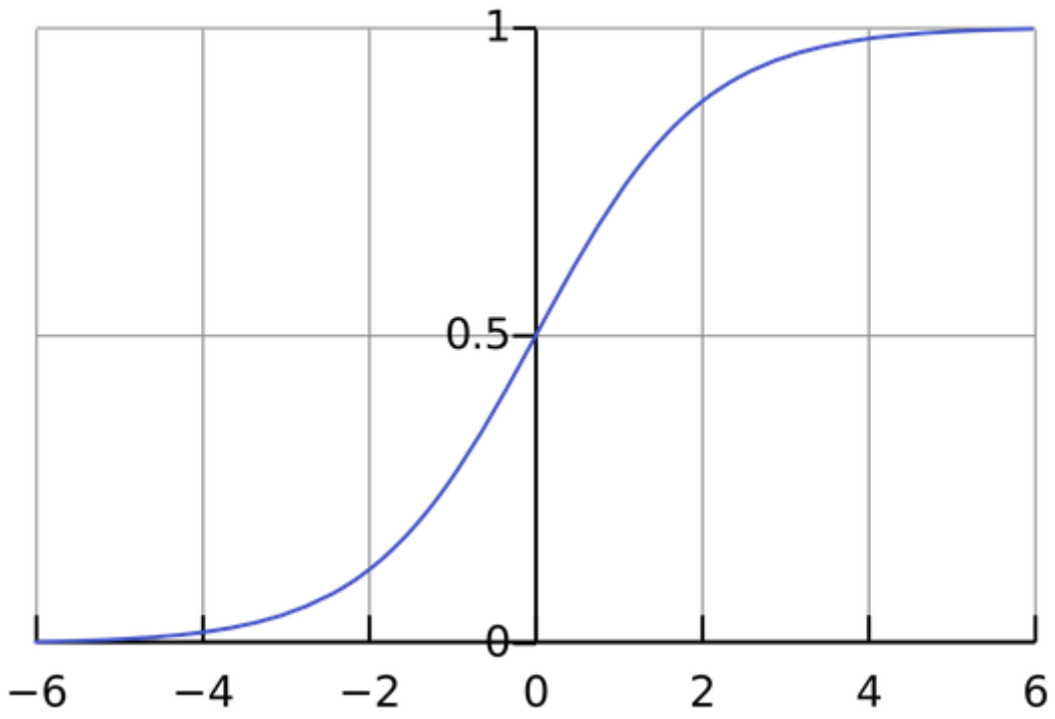
Następnie jest tworzona warstwa pełnopołączeniowa, która łączy wszystkie neurony warstwy poprzedniej z wszystkimi neuronami warstwy bieżącej, tak jak jest robione z zwykłym MLP (Perceptronie wielowarstwowym).

7) *network = lasagne.layers.DenseLayer(*

lasagne.layers.dropout(network, p=.5),
num_units=7,
nonlinearity=lasagne.nonlinearities.softmax)

Na końcu jeszcze jedna warstwa pełnopołączeniowa (ponownie z poprzedzającą warstwą z dropout'em) z 7 neuronami (mamy do rozpoznawania 7 klas emocji). Na wyjściu stosujemy funkcję softmax (co jest standartową praktyką dla ostatniej warstwy). Softmax jest definiowany jako:

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)},$$



Rys. 9. Wykres funkcji softmax

Funkcją tej warstwy jest przypisanie prawdopodobieństwa przynależności przykładu testowanego do poszczególnej klasy.

8. Trenowanie sieci

1. Definiujemy zmienne Theano:

```
input_var = T.tensor4('inputs')

target_var = T.ivector('targets')
```

2. Tworzymy model sieci z warstw opisanych poprzednio:

```
network = lasagne.layers.InputLayer(shape=(None, 1, size, size),
                    input_var=input_var)

network = lasagne.layers.Conv2DLayer(
                    network, num_filters=32, filter_size=(5, 5),
                    nonlinearity=lasagne.nonlinearities.rectify,
                    W=lasagne.init.GlorotUniform())

network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2, 2))

network = lasagne.layers.Conv2DLayer(
                    network, num_filters=32, filter_size=(5, 5),
                    nonlinearity=lasagne.nonlinearities.rectify)

network = lasagne.layers.MaxPool2DLayer(network, pool_size=(2, 2))

network = lasagne.layers.DenseLayer(
                    lasagne.layers.dropout(network, p=.5),
                    num_units=256,
                    nonlinearity=lasagne.nonlinearities.rectify)

network = lasagne.layers.DenseLayer(
                    lasagne.layers.dropout(network, p=.5),
                    num_units=7,
                    nonlinearity=lasagne.nonlinearities.softmax)
```

3. Definiujemy loss function dla trenowania (jest to wielkość skalarna, którą my chcemy zminimalizować). Jako loss function używamy cross-entropy loss.

Metryka cross-entropy jest definiowana jako [11]:

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)]$$

Wzór 1. Metryka cross-entropy

gdzie:
n - ilość przykładów w danych treningowych
y - pożądany wynik rozpatrywanego przykładu treningowego
a - wynik funkcji aktywacji na wejście ważonej sumy wejść

prediction = lasagne.layers.get_output(network)
loss = lasagne.objectives.categorical_crossentropy(prediction, target_var)
loss = loss.mean()

4. Tworzymy wyrażenie aktualizacji wag (w jaki sposób modyfikować parametry na każdym kroku trenowania). Tutaj użyliśmy Stochastic Gradient Descent (SGD) z Nesterov momentum.

Stochastic gradient descent - stochastyczna aproksymacja metody gradientów dla minimizacji funkcji celu, zapisanej jako suma funkcji różniczkowalnych [12]:

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n} Q_i(w)$$

Wzór 2. Stochastic gradient descent

gdzie:
Q_i - wartość loss function na i-tym przykładzie
n - liczba przykładów

Chcemy znaleźć wartość parametru w, dla którego Q(w) jest najmniejsze.

W przypadku standartowej metody gradientowej będą wykonywane następujące iteracje:

$$w := w - \eta \nabla Q(w) = w - \eta \sum_{i=1}^{n} \nabla Q_i(w)$$

Wzór 3. Standardowa metoda gradientowa

gdzie:
eta - rozmiar kroku (learning rate)

W przypadku stochastycznej metody gradientowej rzeczywisty gradient Q(w) jest aproksymowany gradientem pojedynczego przypadku:

$$w := w - \eta \nabla Q_i(w)$$

Wzór 4. Stochastyczna metoda gradientowa

Algorytm przechodzi przez dane treningowe i dokonuje aktualizacji dla każdego przykładu treningowego. Dokonuje on kilku przejść po zbiorze treningowym dopóki nie osiągnie zbieżności. Dane mogą być przetasowane przy każdym przejściu, aby zapobiec powstawaniu cykli.

Pseudokod stochastycznej metody gradientów wygląda następująco:

inicjalizujemy początkowy wektor parametrów *w* i learning rate (*eta*)
while not converged:
    train = shuffle(train)
    for i in range(n):
        w = w - eta*$dQ_i(w)$

params = lasagne.layers.get_all_params(network, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params, learning_rate=0.01, momentum=0.9)

5) Tworzymy loss function dla walidacji/testowania. Istotną różnicą jest to, że nie używamy warstw dropout, żeby uzyskać deterministyczne wyniki.

test_prediction = lasagne.layers.get_output(network, deterministic=True)
test_loss = lasagne.objectives.categorical_crossentropy(test_prediction, target_var)
test_loss = test_loss.mean()

Również definiujemy wyrażenie dla obliczenia dokładności (accuracy) klasyfikacji:

test_acc = T.mean(T.eq(T.argmax(test_prediction, axis=1), target_var),
dtype=theano.config.floatX)

6) Dokonujemy kompilacji funkcji trenującej:

train_fn = theano.function([input_var, target_var], loss, updates=updates)

7) Dokonujemy kompilacji funkcji wyliczającej validation loss i dokładność:

val_fn = theano.function([input_var, target_var], [test_loss, test_acc])

8) Wykonujemy właśnie trenowanie sieci. Do tego została napisana funkcja pomocnicza iterująca po danym zbiorze treningowym w mini-partiach o określonych wymiarach (opcjonalnie w dowolnym porządku).

Wykonujemy pewną ilość przejść po zbiorze danych. Jedno przejście stanowi jedną epokę. Dla każdej epoki robimy po jednym przejściu po zbiorze treningowych (na których właśnie trenowaliśmy) i zbiorze walidacyjnym (dla sprawdzenia bieżącego wyniku).

9)* Biblioteka pozwala również na dotrenowanie sieci bazując na naszym wytrenowanym modelu. Do tego jest potrzebny zbiór zdjęć (jedna osoba na jednym zdjęciu) z ustawionymi podpisami (jaka na danym zdjęciu jest emocja). Na wejście funkcji dotrenowania musi się podawać:

- macierz obrazów wejściowych o wymiarach N x K x H x W, gdzie:
  N - liczba przykładów
  K - głębokość przestrzeni kolorów. Może być 1 (dla obrazów w skali szarości) lub 3 (dla obrazów RGB, które następnie zostaną przekształcone do skali szarości)
  H - wysokość i-tego przykładu (znaleziona twarz zostanie przeskalowana do wysokości 96)
  W - szerokość i-tego przykładu (znaleziona twarz zostanie przeskalowana do szerokości 96)

- wektor klas (dla każdego przykładu numer emocji osoby na tym zdjęciu) (1 - gniew, 2 - pogarda, 3 - zniesmaczenie, 4 - strach, 5 - radość, 6 - smutek, 7 - zaskoczenie)

[1] https://www.cs.toronto.edu/~kriz/cifar.html
[2] https://www.tensorflow.org
[3] http://deeplearning.net/software/theano/
[4] http://caffe.berkeleyvision.org/
[5] https://lasagne.readthedocs.io
[6] https://keras.io
[7] https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf
[8] http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html
[9] https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
[10] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
[11] http://neuralnetworksanddeeplearning.com/chap3.html
[12] https://en.wikipedia.org/wiki/Stochastic_gradient_descent