Systemy Sztucznej Inteligencji

dokumentacja projektu Movie Classifier

Chłąd Paweł
Grupa 2D

Matula Kamil
Grupa 2D

Meller Bartłomiej
Grupa 2D

28 maja 2020

Część I

Opis programu

Movie Classifier to sieć neuronowa służąca do klasyfikacji kadrów z danej puli filmów. Aplikacja jest podzielona na dwie części, a) Program uczący oraz b) Program kliencki.

Instrukcja obsługi

Aby uruchomić program uczący należy przejść do folderu `MovieClassifierLearner` oraz uruchomić program za pomocą `dotnet run arg1 arg2 arg3 itp` lub uruchomić go bezpośrednio z pliku wykonywalnego [1].

Aby uruchomić program kliencki należy przejść do folderu `MovieClassifierClient` oraz uruchomić program za pomocą `dotnet run path_to_image` lub uruchomić go bezpośrednio z pliku wykonywalnego. (Gdzie path_to_image to ścieżka do zdjęcia/klatki z filmu)

Argumenty wejściowe

- Program uczący
  - args[0] - Współczynnik uczenia (LR)
  - args[1] - Alpha w Bipolarnej Linearnej Funkcji
  - args[2] - Minimalna wartość wagi
  - args[3] - Maksymalna wartość wagi
  - args[4+] - Wielkość ukrytych warstw (w neuronach)
- Klient
  - args[0] - Ścieżka do klatki z filmu
  - args[1] - (Opcjonalny) - ścieżka do modelu
  - args[2] - (Opcjonalny) - ścieżka do etykiet

Dodatkowe informacje

Projekt został skompilowany za pomocą .NET Core 3.1

[1] Ze względu na ilość i rozmiary plików, które są tworzone podczas kompilacji, w repozytorium nie umieściliśmy plików wykonywalnych

Część II

Opis działania sieci neuronowej

Jak zostało wcześniej wspomniane program opiera się na sztucznej sieci neuronowej (SSN), czyli matematycznym modelu sieci nerwowej działającej w mózgu. Podobnie jak ludzka sieć neuronowa, SSN zbudowana jest z neuronów ułożonych w warstwy. Każda komórka nerwowa danej warstwy połączona jest ze wszystkimi komórkami warstwy poprzedniej i warstwy następnej za pomocą synaps posiadających pewne losowo zainicjowane wagi w postaci liczb. Są one modyfikowane w procesie uczenia sieci neuronowej.

Pierwszą warstwę sieci, odpowiedzialną za przyjmowanie danych wejściowych, nazywamy warstwą wejściową. Analogicznie ostatnia warstwa sieci to warstwa wyjściowa, odpowiadająca za zwracanie wyniku. Pomiędzy nimi mogą (lecz nie muszą) znajdować się tzw. warstwy ukryte. Zadaniem projektanta sieci neuronowej jest znalezienie optymalnej ilości i wielkości tych warstw, dzięki czemu nauczanie będzie przebiegało efektywnie. Z kolei ilość neuronów na warstwach skrajnych zależy od tego, ile cech posiada obiekt wejściowy oraz do ilu klas można go zaklasyfikować na wyjściu - w przypadku tego projektu jest to 3750 neuronów wejściowych (przetwarzane obrazy mają wymiary 50x25x3) oraz 6 neuronów wyjściowych (do tylu różnych filmów może zostać zakwalifikowany analizowany kadr).

Każdy z neuronów przyjmuje pewną wartość na wejściu, a następnie przetwarza ją dzięki funkcji aktywacji. Sygnał wejściowy i-tego neuronu k-tej warstwy można opisać równaniem:

$$s_i^k = \sum_{j=1}^{n} w_{ij}^k y_j^{k-1} + b,$$

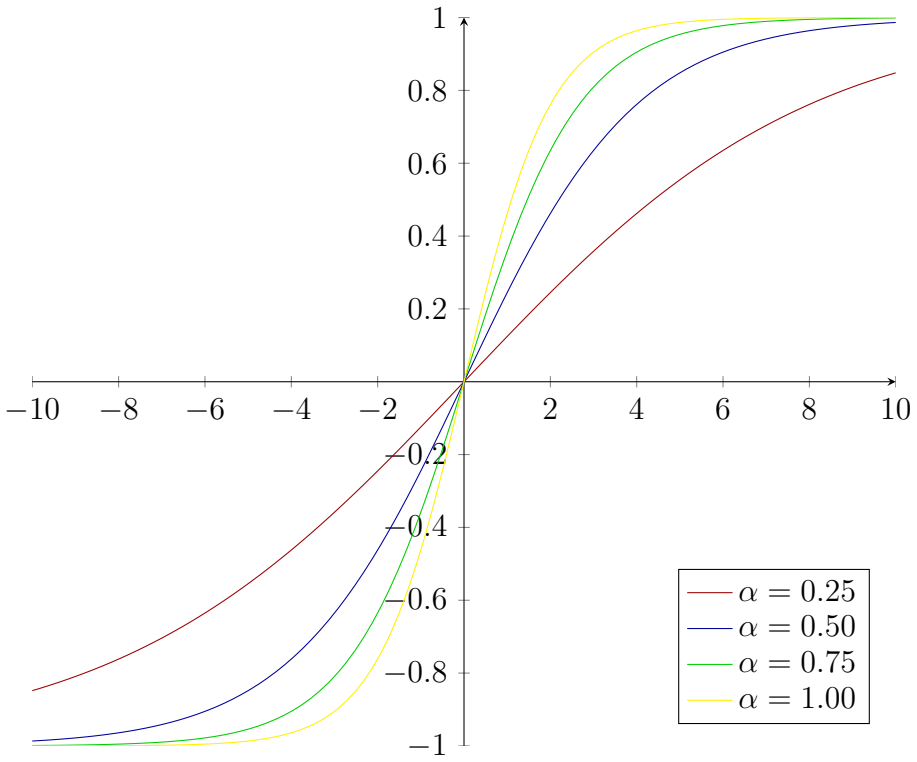
gdzie w_{ij}^k - waga synapsy pomiędzy i-tym neuronem k-tej warstwy a j-tym neuronem warstwy poprzedniej, y_j^{k-1} - wartość sygnału wyjściowego j-tego neuronu warstwy poprzedniej, b - zakłócenia sieci (tzw. bias). Najczęściej we wzorze tym nie uwzględnia się ostatniego czynnika (zakłada się, że sieć nie posiada zakłóceń tj. b = 0). Z kolei sygnał wyjściowy i-tego neuronu to:

$$y_i^k = f(s_i^k) = f(\sum_{j=1}^{n} w_{ij}^k y_j^{k-1} + b).$$

Wyróżniamy wiele funkcji aktywacji, jednak najczęściej wykorzystywaną (i wykorzystaną również w tym projekcie) jest funkcja bipolarna liniowa, której wzór wygląda następująco:

$$f(s_i^k) = \frac{2}{1 + e^{-\alpha s_i^k}} - 1 = \frac{1 - e^{-\alpha s_i^k}}{1 + e^{-\alpha s_i^k}}$$

gdzie α jest współczynnikiem korygującym rozpiętość funkcji aktywacji w przestrzeni decyzyjnej. Jej wykres zamieszczono na następnej stronie.



Bipolarna liniowa funkcja aktywacji

Kiedy sztuczna sieć neuronowa jest już odpowiednio zbudowana, należy ją nauczyć tego, czego od niej oczekujemy. Polega to na modyfikowaniu wag synaps w ściśle określony sposób. Jest wiele metod uczenia z czego część wymaga nauczyciela w postaci zbioru treningowego z danymi wejściowymi i oczekiwanymi danymi wyjściowymi, a część nie - wtedy sieć dostaje tylko dane wejściowe. W przypadku uczenia rozpoznawania obiektów stosuje się metody uczenia z nauczycielem. Jedną z takich strategii jest algorytm wstecznej propagacji. Jej zadaniem jest zminimalizowanie wartości funkcji błędu dla wszystkich elementów zbioru treningowego T, co opisuje wzór:

$$B(T) = \sum_{T} \sum_{i=1}^{n} (d_i - y_i)^2,$$

gdzie n to wymiar wektora wyjściowego / liczba neuronów wyjściowych, d_i to wartość oczekiwana na i-tej pozycji wektora wyjściowego, a y_i to wartość uzyskana na i-tej pozycji wektora wyjściowego. Korekcja wag synaps wejściowych poszczególnych neuronów zaczyna się w warstwie wyjściowej oznaczanej literą K i przebiega wstecz przez wszystkie wcześniejsze warstwy aż dotrze do warstwy wejściowej. Równanie korekcji wag wygląda następująco:

$$w_{ij}^k = w_{ij}^k + \eta \nabla w_{ij}^k,$$

gdzie η jest współczynnikiem korekcji powszechnie nazywanym „Learning Rate”, a ∇w_{ij}^k to wartość gradientu błędu wagi synapsy opisywana wzorem:

$$\nabla w_{ij}^k = \frac{\partial B(T)}{\partial w_{ij}^k} = \frac{1}{2} \cdot \frac{\partial B(T)}{\partial s_i^k} \cdot 2 \cdot \frac{\partial s_i^k}{\partial w_{ij}^k} = 2\delta_i^k y_j^{k-1},$$

gdzie δ_i^k to zmiana funkcji błędu dla sygnału wejściowego i-tego neuronu k-tej warstwy, a y_j^{k-1} to sygnał wyjściowy j-tego neuronu warstwy poprzedniej. Wspomniana wartość δ liczona jest inaczej na warstwie wyjściowej i inaczej na pozostałych. Na K-tej warstwie wynosi:

$$\delta_i^K = \frac{1}{2} \cdot \frac{\partial B(T)}{\partial s_i^k} = \frac{1}{2} \cdot \frac{\partial (d_i^K - y_i^K)^2}{\partial s_i^k} = f'(s_i^K) \cdot (d_i^K - y_i^K),$$

gdzie $f'(s_i^K)$ to pochodna funkcja aktywacji na K-tej warstwie (wyjściowej). Wartość zmiany funkcji błędu na pozostałych warstwach jest zależna od wartości uzyskanej na warstwie następnej i jest równa:

$$\delta_i^k = \frac{1}{2} \cdot \frac{\partial B(T)}{\partial s_i^k} = \frac{1}{2} \cdot \sum_{j=1}^{N_{k+1}} \frac{\partial B(T)}{\partial s_j^{k+1}} \frac{\partial s_j^{k+1}}{\partial s_i^k} = f'(s_i^k) \sum_{j=1}^{N_{k+1}} \delta_j^{k+1} w_{ij}^{k+1},$$

gdzie N_{k+1} to liczba neuronów warstwy następnej.

Algorytm

Uczenie sieci

Data: ilość iteracji - *EpochsCount*, dane wejściowe zbioru treningowego - $Inputs$, oczekiwane dane wyjściowe zbioru treningowego - *ExpectedOutputs*
Result: Większa dokładność sieci
$L :=$ ilość warstw sieci neuronowej;
$Deltas :=$ pusta tablica poszarpana o L wierszach i tylu kolumnach w danym wierszu, ile neuronów ma dana warstwa; będzie przetrzymywać wartości δ;
for $i = 0$ **to** *EpochsCount* **do**
  for $j = 0$ **to** *wielkość zbioru treningowego* **do**
    Wprowadź j-ty wektor wejściowy zbioru treningowego ($Inputs[j]$) do synaps wchodzących neuronów pierwszej warstwy;
    for $k = 0$ **to** L **do**
      Wyznacz s^k na wszystkich neuronach k-tej warstwy, sumując iloczyny wag synaps wchodzących i y^k neuronów warstwy poprzedniej (lub synaps w przypadku pierwszej warstwy);
      Wyznacz y^k na wszystkich neuronach k-tej warstwy poprzez zastosowanie funkcji aktywacji;
    end
    $Output :=$ wektor złożony z wartości wyjściowych ostatniej warstwy;
    for $n = 0$ **to** *ilość neuronów wyjściowych* **do**
      $Deltas[L-1][n] = (ExpectedOutputs[j][n] - Output[j]) \cdot f'(s_n^{L-1})$;
    end
    for $k = L-2$ **to** 0 **by** -1 **do**
      for $n = 0$ **to** *ilość neuronów na k-tej warstwie* **do**
        $Deltas[k][n] = 0$;
        for $m = 0$ **to** *ilość neuronów na (k+1)-tej warstwie* **do**
          $Deltas[k][n] = Deltas[k][n] + Deltas[k+1][m] \cdot w_{mn}^{k+1}$;
        end
        $Deltas[k][n] = Deltas[k][n] \cdot f'(s_n^k)$;
      end
    end
    for $k = L-2$ **to** 0 **by** -1 **do**
      for $n = 0$ **to** *ilość neuronów na k-tej warstwie* **do**
        for $m = 0$ **to** *ilość neuronów na (k-1)-tej warstwie* **do**
          $w_{nm}^k = 2 \cdot LR \cdot Deltas[k][n] \cdot y_m^{k-1}$;
        end
      end
    end
  end
end

Algorithm 1: Algorytm trenowania sztucznej sieci neuronowej.

Widoczny na poprzedniej stronie algorytm przedstawia pełny proces trenowania sieci neuronowej z wykorzystaniem zbioru treningowego i algorytmu wstecznej propagacji. Przedziały liczbowe, przez które przebiegają zapisane pętle **for** są jednostronnie domknięte (liczba po **to** nie jest brana pod uwagę). Pojawiają się też zapisy LR, s_i^k, y_i^k i w_{ij}^k - są to kolejno: wartość współczynnika nauczania (Learning Rate), wartość wejściowa i wartość wyjściowa i-tego neuronu na k-tej warstwie oraz waga synapsy pomiędzy i-tym neuronem k-tej warstwy a j-tym neuronem warstwy poprzedniej. Ponadto $f'(\cdot)$ oznacza wartość pochodnej funkcji aktywacji - dla funkcji bipolarnej liniowej o wzorze $f(x) = \frac{1-e^{-\alpha x}}{1+e^{-\alpha x}}$ pochodna wynosi $f'(x) = \frac{2\alpha e^{-\alpha x}}{(1+e^{-\alpha x})^2}$.

Zbiór danych

Generowanie datasetu wyglądało następująco. Filmy odtwarzane były na platformie Netflix. Skrypt napisany w języku Python3, poruszając i klikając kursorem myszki, przesuwał film o stałą ilość czasu do przodu. Czekał aż interfejs zniknie, wykonywał i zapisywał zrzut ekranu. W ten sposób powstawało ponad 250 zrzutów na film. Z tak powstałego zbioru danych eliminowane były zrzuty ekranu będące nieczytelnymi oraz zwierające jedynie monolityczny czarny kolor.

Następnie obrazy zostały wczytane do aplikacji, której zadaniem było ich ustandaryzowanie oraz zmniejszenie. SSN, która została zaakceptowana posiada wektor wejściowy o wielkości 50x25x3 (50x25 pikseli, każdy po 3 kolory BGR). Aplikacja więc zmniejszy obraz do żądanej wielkości. Proces zmniejszania wygląda następująco:

1. Zdjęcie jest przycinane, aby stosunek długości do wysokości zdjęcia był taki sam jak stosunek długości do wysokości oczekiwanego obrazu wejściowego (bieżące AspectRatio: 2) - jest to wymagane, aby proces zmniejszania nie rozciągał zdjęcia, co mogłoby sprawić problemy przy uczeniu.

2. Często klatki z filmów posiadają tzw. letterboxa (czarne paski u góry i na dole, czasami po bokach też). Czarne piksele mogą wpływać na proces uczenia się SSN, więc zdjęcie jest ponownie przycinane; wycinane jest $\frac{1}{12}$ zdjęcia z każdej strony.

3. Obrobioną już klatkę, zmniejszamy do wymiarów 50x25.

4. Klatka zostaje zamieniona na tablicę typu `double`.

Przykładowy kadr z filmu „Lego Przygoda”

Implementacja

Ogólna struktura

Cały projekt składa się z wielu mniejszych projektów (każdy w swoim osobnym folderze):

- DataPreparer - projekt biblioteki do ładowania zdjęć i przerabiania ich na format czytelny dla sieci neuronowej,
- DataPreparerTests - projekt zawierający testy jednostkowe dla biblioteki DataPreparer,
- DataSetGenerator - zestaw skryptów Python, które generują dataset z wybranych filmów platformy Netflix,
- MovieClassifierClient - projekt klienta aplikacji,
- MovieClassifierLearner - projekt programu uczącego,
- NeuralNetwork - projekt biblioteki sieci neuronowej,
- NeuralNetworkTests - projekt testów jednostkowych biblioteki sieci neuronowej.

DataPreparer

Biblioteka DataPreparer zawiera dwie klasy:

- `static DataPreparer` - statyczna klasa zawierająca metody obróbki i ładowania zdjęć,
- `struct ImageLearningData` - struktura trzymająca dane o zdjęciu i danych do nauki.

DataPreparer - Metody

- `ImageLearningData PrepareImage(string path, int imageWidth, int imageHeight)` - odczytuje obraz i zmienia go w obsługiwany format dla aplikacji uczącej
- `ImageLearningData[] PrepareImages(string path, int imageWidth, int imageHeight` - odczytuje wszystkie obrazy w podanej ścieżce i zmienia je w obsługiwany format dla aplikacji uczącej

ImageLearningData - Pola i właściwości

- `ReadOnlyCollection<double> data` - kolekcja zawierająca informacje o obrazie w formacie BGR
- `string label` - etykieta (używana przy procesie uczenia)
- `int number` - numer próbki
- `readonly int width` - szerokość obrazu
- `readonly int height` - wysokość obrazu

NeuralNetwork

Biblioteka zawiera następujące klasy:

- `static class ArrayExtensions` - klasa zawierająca metody rozszerzające dla tablic
- `class ConvertUtil` - klasa zawierająca metody pomocnicze do konwersji argumentów
- `static class Functions` - klasa zawierająca funkcje aktywacji
- `class Layer` - klasa reprezentująca warstwę neuronów
- `static class ListExtensions` - klasa zawierające metody rozszerzeń dla list
- `class Network` - główna klasa biblioteki reprezentująca sieć neuronową
- `class Neuron` - klasa reprezentująca neuron
- `static class Normalizator` - klasa zawierające metodę normalizacji danych
- `static class Shuffler` - klasa zawierająca metodę przetasowywania kolekcji
- `class Synapse` - klasa reprezentująca połączenie między neuronami
- `interface ITestStrategy` - interfejs strategii testów
- `class MeanErrorTest : ITestStrategy` - test sieci: błąd średniokwadratowy
- `class HighestHitTest : ITestStrategy` - test sieci: celność

ArrayExtensions - Metody

- `T[] GetColumn(int index)` - zwraca kolumnę w tablicy
- `SetColumn(T[] data, int index)` - ustawia kolumnę w tablicy
- `int MaxAt()` - zwraca indeks elementu który posiada najwyższą wartość

ConvertUtils - Metody

- `static double ConvertArg(string s)` - konwertuje ciąg znaków do typu double, bez względu na aktualny język wątku

Functions - Metody

- `static double CalculateError(List<double> outputs, int row, double[][] expectedOutputs )` - oblicza błąd średniokwadratowy
- `static double InputSumFunction(List<Synapse> Inputs)` - oblicza ważoną sumę dla listy wchodzących połączeń
- `static double BipolarLinearFunction(double input)` - oblicza wartość funkcji bipolarnej liniowej (funkcja aktywacji)

- `static double BipolarDifferential(double input)` - oblicza wartość pochodnej funkcji bipolarnej liniowej

Functions - Pola i Właściwości

- `static double Alpha` - współczynnik dla funkcji bipolarnej liniowej

Layer - Metody

- `Layer(int numberOfNeurons)` - konstruktor - tworzy instancję `Layer` wraz z `numberOfNeurons` neuronami
- `void ConnectLayers(Layer outputLayer)` - łączy neurony wywołującego z neuronami `outputLayer` za pomocą instancji `Synapse`
- `void CalculateOutputOnLayer` - wywołuje `CalculateOutput` dla każdego neuronu w tej warstwie

Layer - Pola i Właściwości

- `List<Neuron> Neurons` - kolekcja przechowująca instancje klasy `Neuron`

ListExtensions - Metody

- `int MaxAt()` - zwraca indeks elementu który posiada najwyższą wartość

Network - Metody

- `Network(double learningrate, double alpha, double mininitweight, double maxinitweight , int numInputNeurons, int[] hiddenLayerSizes, int numOutputNeurons, bool testHaltEnabled = false, bool testingEnabled = true, bool recordSaveEnabled = true)` - tworzy nową instancję sieci z następującymi parametrami:
  - `double learningRate` - modyfikator nauczania
  - `double alpha` - współczynnik alpha funkcji sigmoidalnej bipolarnej
  - `double minInitWeight` - minimalna waga która może zostać przypisana połączeniu podczas inicjalizacji
  - `double maxInitWeight` - maksymalna waga która może zostać przypisana połączeniu podczas inicjalizacji
  - `int numInputNeurons` - ilość neuronów w warstwie 0
  - `int[] hiddenLayerSizes` - ilość neuronów w poszczególnych warstwach ukrytych
  - `int numOutputNeurons` - ilość neuronów w ostatniej warstwie
  - `bool testHaltEnabled` - flaga, jeśli ustawiona na `true` to nauczenie automatycznie zatrzyma się gdy test wykryje pogorszenie wyników.
  - `bool testingEnabled` - flaga, jeśli ustawiona na `true` to sieć będzie testowana co epoch

  - `bool recordSaveEnabled` - flaga, jeśli ustawiona na `true` to topologia i parametry sieci zostanie zapisana do pliku po osiągnięciu rekordowego (dotychczas) wyniku

- `void PushInputValues(double[] inputs)` - ustawia wartości synaps wchodzących do warstwy 0
- `void PushExpectedValues(double[] values)` - ustawia wartości oczekiwane na ostatniej warstwie (używane w procesie uczenia)
- `List<double> GetOutputs()` - oblicza wynik dla tej sieci, z wartości przekazanych przez `PushInputValues`
- `void Train(double[][][] data, int epochCount)` - uczy sieć używając algorytmu propagacji wstecznej, przez `epochCount` epochów. Wartości argumentu data oznaczają:
  - `data[0]` - dane wejściowe
  - `data[1]` - oczekiwane wartości wyjściowe
  - `data[2]` - testowe wartości wejściowe
  - `data[3]` - testowe wartości oczekiwane

  Aby proces uczenia odbył się prawidłowo należy upewnić się czy długości `data[0]` i `data[1]` są równe, analogicznie z `data[2]` i `data[3]`
- `void RandomizeWeights()` - Nadaje losowe wartości wszystkim wagom
- `void SaveNetworkToFile(string path)` - Zapisuje sieć do pliku w `path`
- `static Network LoadNetwrokFromFile(string path)` - Tworzy nową istancję Network i odczytuje z pliku topologie oraz parametry sieci
- `int GetLayerSize(int layerIndex)` - zwraca wielkość `layerIndex`-tej warstwy

Network - Pola i Właściwości

- `ITestStrategy testStrategy;` - strategia testowania sieci
- `bool TestHaltEnabled { get; set; }` - flaga, jeśli ustawiona na `true` to nauczenie automatycznie zatrzyma się gdy test wykryje pogorszenie wyników.
- `bool TestingEnabled { get; set; }` - flaga, jeśli ustawiona na `true` to sieć będzie testowana co epoch
- `bool RecordSaveEnabled { get; set; }` - flaga, jeśli ustawiona na `true` to topologia i parametry sieci zostanie zapisana do pliku po osiągnięciu rekordowego (dotychczas) wyniku

Neuron - Metody

- `Neuron()` - tworzy nową instancję klasy `Neuron`
- `void AddOutputNeuron(Neuron outputNeuron)` - łączy wywołującego z outputNeuron, nową instancją klasy `Synapse`

- `void CalculateOutput()` - oblicza wartość neuronu korzystając z wchodzących połączeń i ustawia OutputValue na obliczoną wartość
- `void AddInputSynapse()` - dodaje nową instancję klasy `Synapse` do listy wchodzących połączeń

Neuron - Pola i Właściwości

- `List<Synapse> Inputs { get; set; }` - lista połączeń wchodzących
- `List<Synapse> Outputs { get; set; }` - lista połączeń wychodzących
- `double InputValue { get; set; }` - ostatnia wartość wchodząca
- `double OutputValue { get; set; }` - ostatnia wartość wychodząca

Normalizator - Metody

- `static double Normalize(double[] input, double nmin, double nmax)` - normalizuje `input` w granicach od `nmin` do `nmax`

Shuffler - Metody

- `void Shuffle(T[])` - miesza tablicę

Synapse - Metody

- `Synapse(Neuron fromNeuron, Neuron toNeuron)` - tworzy synapsę pomiędzy dwoma neuronami
- `Synapse(Neuron toNeuron, double data)` - tworzy synapsę bez nadawcy, metoda używana do tworzenia zerowej warstwy
- `double GetOutput()` - oblicza wartość na wywołującym połączeniu

Synapse - Pola i Właściwości

- `double Weight { get; set; }` - Waga połączenia
- `double PushedData { get; set; }` - Aktualna wartość przychodząca
- `static int SynapsesCount { get; set; } = 0;` - Informacja o topologii sieci
- `static double MaxInitWeight { get; set; }` - Maksymalna wartość wagi, która może być nadana przy inicjalizacji
- `static double MinInitWeight { get; set; }` - Minimalna wartość wagi, która może być nadana przy inicjalizacji

ITestStrategy

- `double CurrentRecord {get;}` - bierzący rekord
- `double Test(double[][] input, double[][] expectedOutput)` - metoda testująca sieć

- `bool CheckHalt()` - sprawdzenie warunku zatrzymania uczenia
- `bool CheckRecord()` - sprawdzenie rekordu

HighestHitTest - Metody

- `HighestHitTest(Network network, double minDelta = 0.001)` - tworzy nową instancję strategii testu
- `double Test(double[][] inputs, double[][] expectedOutputs)` - zwraca wynik testu dla danego zestawu testowego
- `bool CheckHalt()` - zwraca `true`, jeśli `recentPercentage` jest większe bądź równe `maximumPercentageHalt`
- `bool CheckRecord()` - zwraca `true`, jeśli `recentPercentage` przekracza dotychczas odnotowany rekord i ustawia `CurrentRecord` na `recentPercentage`

HighestHitTest - Pola i Właściwości

- `private double maximumPercentageHalt` - maksymalna zadana celność, po osiągnięciu której `Train` w `Network` zostaje przerwane.
- `private double recentPercentage` - ostatnia zarejestrowana celność

MeanErrorTest - Metody

- `HighestHitTest(Network network, double minDelta = 0.001)` - tworzy nową instancję strategii testu
- `double Test(double[][] inputs, double[][] expectedOutputs)` - zwraca wynik testu dla danego zestawu testowego
- `bool CheckHalt()` - zwraca `true` jeśli `recentError` jest mniejsze bądź równe `MinError`
- `bool CheckRecord()` - zwraca `true` jeśli `recentError` przekracza dotychczas odnotowany rekord i ustawia `CurrentRecord` na `recentError`

MeanErrorTest - Pola i Właściwości

- `private Network network` - instancja klasy `Network` obsługiwana przez ten test
- `private double minError` - zaplecze dla `MinError`
- `public double MinError{ get; set; }` - minimalny błąd, do którego ma się zbliżyć sieć
- `public double CurrentRecord { get; private set; }` - bieżący rekord
- `private double recentError` - ostatni zarejestrowany błąd

Testy

Początkowo planowaliśmy nauczyć sieć neuronową rozpoznawania trzech filmów animowanych: *Shrek 2*, *Madagaskar* oraz *Rybki z ferajny*. Niestety ze względu na duże podobieństwo dokładność rozpoznawania kadrów była dosyć niska - dla zbioru testowego złożonego z 25 kadrów przypadających na jeden film (i zbioru treningowego złożonego ze 225 kadrów / film) w szczytowych momentach osiągała zaledwie 65% skuteczności. Zdecydowaliśmy powiększyć pulę do sześciu filmów poprzez dodanie tytułów bardziej różniących się od siebie: *Indiana Jones*, *Twój Vincent* oraz *Lego Przygoda*. Zarówno to jak i zwiększenie liczby testów poskutkowało znalezieniem optymalnej architektury sieci neuronowej, a co za tym idzie zwiększeniem jej skuteczności, co przedstawia poniższa tabela:

Architektura: 1 warstwa ukryta, 50 neuronów	
Liczba filmów	Maks. dokładność
4	76.0 %
5	71.2 %
6	52.7 %

Powyższe wyniki uzyskano przy zastosowaniu współczynnika nauczania η na poziomie 0.05 oraz współczynnika korekcji α w bipolarnej liniowej funkcji aktywacji na poziomie 0.5. Co ciekawe sieć o jednej 10-neuronowej warstwie ukrytej również charakteryzowała się wysokimi osiągami:

Architektura: 1 warstwa ukryta, 10 neuronów	
Liczba filmów	Maks. dokładność
4	75.0 %
5	68.8 %
6	58.0 %

W obu powyższych tabelach w testach dotyczących 5 filmów zrezygnowano z filmu „Shrek 2", a w przypadku 4 filmów dodatkowo nie uwzględniono filmu „Madagaskar". Przetestowano także uczenie się sieci dla zestawu 4 filmów, w którym w miejsce tytułu „Rybki z ferajny" znalazł się „Madagaskar". W większości przypadków sieć ta miała skuteczność o 10 punktów procentowych niższą niż gdy w zestawie znajdował się pierwszy z tytułów.

Ze względu na fakt wykorzystania propagacji wstecznej jako algorytmu uczącego, często wzrost trafności predykcji sieci „utykał" na nieakceptowalnym poziomie i tylko w niewielkiej ilości testów osiągała ona zadowalający poziom. Z tego powodu w późniejszych testach zdecydowaliśmy się na douczanie wstępnie nauczonej sieci neuronowej poprzez randomizację wag. Ponadto zastosowaliśmy strategię polegającą na stopniowym zwiększaniu ilości rozpoznawanych klas dla jednego modelu. Więcej na ten temat w rozdziale „Eksperymenty".

Architektura: 3 warstwy ukryte, kolejno: 800, 200 i 50 neuronów	
Liczba filmów	Maksymalna dokładność
4	82.0 %
5	88.8 %
6	84.6 %

Eksperymenty

Głównym problemem, z którym zmagaliśmy się podczas uczenia sieci, było dobranie odpowiedniej architektury. Architektura ta zakłada względnie dużą dokładność oraz pojemność wystarczającą do nauczenia sieci rozpoznawania kilku klas, przy minimalnej ilości neuronów zapewniającej stosunkowo krótki czas uczenia. Początkowo, testy były wykonywane dla 4 klas.

Do pewnego momentu, najlepsze rezultaty (76% trafności) wydawała się przynosić architektura wyposażona w tylko jedną warstwę ukrytą, zawierającą zaledwie 50 neuronów. Z uwagi na bardzo niewielką pojemność takiej sieci postanowiliśmy poszukać większego modelu, który zapewni odpowiednią pojemność, przy jednoczesnej akceptowalnej szybkości uczenia. Finalnie obraliśmy model posiadający 3 warstwy ukryte, zawierające kolejno 800, 200 i 50 neuronów.

Uczenie algorytmem propagacji wstecznej ma jedną znaczącą wadę. Nie używa on żadnej heurystyki, która zabezpieczałaby sieć przed utknięciem w minimach lokalnych. Postanowiliśmy zaradzić jakoś temu problemowi. Zaskakująco skuteczne okazało się dodawanie losowej liczby z przedziału $[-0.002; 0.002]$ do wag wszystkich połączeń. Operacja ta będzie zwana dalej „randomizacją". W ten sposób udało nam się znacząco zwiększyć dokładność. Trudno jest jednak mówić o konkretnej wartości, gdyż metoda ta była używana jednocześnie ze stopniowym zwiększaniem ilości klas rozpoznawanych przez sieć.

Zwiększanie ilości klas, paradoksalnie przyniosło skok trafności predykcji wykonywanych przez naszą sieć. Skok ten nie zachodził jednak od razu. Potrzebne było kilka epok nauki, aby sieć dostosowała się do nowych warunków.

W chwili pisania tej dokumentacji, bezwzględna trafność dla sześciu klas, przewyższyła trafność uzyskiwaną w przypadku, gdy sieć „znała" tylko cztery. Taki stan rzeczy jest wytłumaczalny przez fakt rosnącej ogólności klasyfikatora wytworzonego przez model dla zwiększającej się liczby klas.

Metodologia uczenia dla obranego przez nas modelu wyglądała następująco. Model uczony był do momentu gdy kilka lub kilkanaście kolejnych epok nie przynosiło żadnego postępu. Kolejnym etapem była randomizacja jego wag. W ten sposób powstawało trzech „potomków" nauczonej wstępnie sieci. W następnym kroku nauka „potomków" oraz wyjściowej sieci (lub w niektórych wypadkach czterech potomków) prowadzona była równolegle do momentu, gdy każda z instancji nie przestała robić postępów. Na końcu wybierany był najlepszy z nich, a cały proces, począwszy od randomizacji, był powtarzany.

Jednym z bardziej interesujących problemów, z którymi spotkaliśmy się w trakcie początkowej fazy poszukiwań, była znacząco spadająca dokładność w momencie dodania filmu „Shrek 2" do klas problemu. Co ciekawe problem ten występował tylko dla małych, konkretnie klasyfikujących modeli. Po wstępnym nauczeniu na innych filmach, przy większej architekturze, problem został wyeliminowany.

Pełen kod aplikacji

Kod znajduje się poniżej jak i również w repozytorium pod adresem: `https://github.com/Madoxen/MLProject_secondary`.

DataPreparerTests.cs

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using DataPreparer;
using System.Diagnostics;
using System.Drawing;
using System.Drawing.Imaging;


namespace DataPreparerTests
{
    [TestClass]
    public class DataPreparerTests
    {
        [TestMethod]
        public void TestDataCount()
        {
            ImageLearningData ld = DataPreparer.ImageDataPreparer.
               PrepareImage("Resources/test_1.jpg",200,100);

            Assert.AreEqual(200, ld.width);
            Assert.AreEqual(100, ld.height);
            Assert.AreEqual(60000, ld.data.Count); //200 * 100 * 3 (BGR)

        }

        [TestMethod]
        public void TestDataCorrectness()
        {
            var a = DataPreparer.ImageDataPreparer.PrepareImage("
               Resources/test_1.jpg",200,100);
            Bitmap b = new Bitmap("Resources/target_1.bmp");
            int dataPos = 0;
            for (int i = 0; i < b.Height; i++)
            {
                for (int j = 0; j < b.Width; j++)
                {
                    Color c = b.GetPixel(j,i);
                    Assert.AreEqual(c.B, a.data[dataPos] * 255, 1);
                    Assert.AreEqual(c.G, a.data[dataPos + 1] * 255, 1);
                    Assert.AreEqual(c.R, a.data[dataPos + 2] * 255, 1);
                    dataPos += 3;
                }
            }
        }
    }
}
```

DataPreparer.cs

```
1  using System.Drawing;
2  using System.Drawing.Imaging;
3  using System.Runtime.InteropServices;
4  using System.IO;
5  using System;
6
7  namespace DataPreparer
8  {
9      //Prepares data from images
10     public static class ImageDataPreparer
11     {
12
13
14         ///<summary>
15         ///Prepares one image
16         ///Uses file name as a label, label search terminates at '_'
               after '_' signifies sample number
17         ///</summary>
18         public static ImageLearningData PrepareImage(string path, int
               targetWidth, int targetHeight)
19         {
20
21             int paddingRatio = 12;
22             //Extract data
23             Bitmap original = new Bitmap(path);
24
25
26             //Perform cropping and resize
27             Bitmap croppedToRatio = CropToRatio(original, 2.0);
28
29             Rectangle rect = new Rectangle(croppedToRatio.Width /
                   paddingRatio,
30     croppedToRatio.Height / paddingRatio,
31     croppedToRatio.Width - (2 * (croppedToRatio.Width / paddingRatio)),
32     croppedToRatio.Height - (2 * (croppedToRatio.Height / paddingRatio)))
           ;
33
34
35             Bitmap cropped = CropBitmap(croppedToRatio, rect);
36             Bitmap resized = new Bitmap(cropped, new Size(targetWidth,
                   targetHeight));
37             BitmapData data = resized.LockBits(new Rectangle(0, 0,
                   resized.Width, resized.Height), ImageLockMode.ReadOnly,
                   PixelFormat.Format24bppRgb);
38             int depth = 3; //bytes per pixel
39             byte[] buffer = new byte[data.Width * data.Height * depth];
40
41             //copy pixels to buffer
42             unsafe
43             {
44                 int Height = resized.Height;
45                 int Width = resized.Width;
46                 int pos = 0;
```

```
47                  byte* ptr = (byte*)data.Scan0;
48                  for (int y = 0; y < Height; y++)
49                  {
50                      byte* ptr2 = ptr;
51                      for (int x = 0; x < Width; x++)
52                      {
53                          buffer[pos++] = *(ptr2++); //B
54                          buffer[pos++] = *(ptr2++); //G
55                          buffer[pos++] = *(ptr2++); //R
56                      }
57                      ptr += data.Stride;
58                  }
59              }
60
61              resized.UnlockBits(data);
62
63              //Extract label
64              string fileName = Path.GetFileNameWithoutExtension(path);
65              string[] tokens = fileName.Split("_");
66              string label = tokens[0];
67              int number = Convert.ToInt32(tokens[1]);
68
69              //Free GDI handles
70              resized.Dispose();
71              croppedToRatio.Dispose();
72              cropped.Dispose();
73              original.Dispose();
74
75              return new ImageLearningData(data.Width, data.Height, buffer
                    , label, number);
76          }
77
78
79
80          ///<summary>
81          ///Prepares entire directory of images
82          ///</summary>
83          /// <param name="path">Path to directory that contains images</
                param>
84          /// <returns></returns>
85          public static ImageLearningData[] PrepareImages(string path, int
                 width, int height)
86          {
87              string[] files = Directory.GetFiles(path, "*.png");
88              ImageLearningData[] result = new ImageLearningData[files.
                    Length];
89              for (int i = 0; i < files.Length; i++)
90              {
91                  result[i] = PrepareImage(files[i], width, height);
92              }
93              return result;
94          }
95
96          private static Bitmap CropBitmap(Bitmap img, Rectangle cropArea)
97          {
```

```
98              Bitmap bmpImage = new Bitmap(img);
99              return bmpImage.Clone(cropArea, bmpImage.PixelFormat);
100         }
101
102
103         private static Bitmap CropToRatio(Bitmap input, double
                expectedAR)
104         {
105             double AR = input.Width / input.Height;
106
107             if (AR > expectedAR) //cut width center wise
108             {
109                 int cropAmount = input.Width - (int)(expectedAR * input.
                        Height);
110                 Rectangle rect = new Rectangle(cropAmount / 2,
111                 0,
112                 input.Width - cropAmount,
113                 input.Height);
114
115                 Bitmap target = new Bitmap(rect.Width, rect.Height);
116
117                 using (Graphics g = Graphics.FromImage(target))
118                 {
119                     g.DrawImage(input, new Rectangle(0, 0, target.Width,
                            target.Height),
120                                 rect,
121                                 GraphicsUnit.Pixel);
122                 }
123                 return target;
124             }
125             else if (AR < expectedAR) //cut height center wise
126             {
127                 int cropAmount = input.Height - (int)((double)input.
                        Width / expectedAR);
128                 Rectangle rect = new Rectangle(0,
129                cropAmount / 2,
130                input.Width,
131                input.Height - cropAmount);
132
133                 Bitmap target = new Bitmap(rect.Width, rect.Height);
134
135                 using (Graphics g = Graphics.FromImage(target))
136                 {
137                     g.DrawImage(input, new Rectangle(0, 0, target.Width,
                            target.Height),
138                                 rect,
139                                 GraphicsUnit.Pixel);
140                 }
141                 return target;
142             }
143             else
144             {
145                 return input;
146             }
147
```

```
148             }
149
150
151
152         }
153
154
155
156
157 }
```

ImageLearningData.cs

```
1  using System.Collections.ObjectModel;
2
3
4  namespace DataPreparer
5  {
6      public struct ImageLearningData
7      {
8
9
10         /// <summary>
11         /// Array containing raw data
12         /// in BGR format
13         /// </summary>
14         public ReadOnlyCollection<double> data;
15
16
17         /// <summary>
18         /// Label for this image
19         /// </summary>
20         public string label;
21         /// <summary>
22         /// Sample number of this image
23         /// </summary>
24         public int number;
25
26         public readonly int width;
27         public readonly int height;
28
29
30         /// <summary>
31         /// Creates new instance of Image data
32         /// </summary>ImageData result =
33         /// in R = nth pixel; G = (n+1)th pixel; B = (n+2)th pixel</
               param>
34         public ImageLearningData(int Width, int Height, byte[] rawData,
               string label, int number)
35         {
36             this.width = Width;
37             this.height = Height;
38             double[] d = new double[rawData.Length];
39
40             for(int i = 0; i < rawData.Length; i++)
41             {
42                 d[i] = ((double)rawData[i]/255.0);
43             }
44
45             this.data = new ReadOnlyCollection<double>(d);
46             //Assign label
47             this.label = label;
48             this.number = number;
49         }
50
51     }
```

```
52 }
```

Program.cs

```
1  using System;
2  using System.IO;
3  using NeuralNetwork;
4  using DataPreparer;
5  using System.Collections.Generic;
6
7  namespace MovieClassifierClient
8  {
9      class Program
10     {
11         /// <summary>
12         /// Args:
13         /// 0 - image path
14         /// 1 (optional, default: model.txt) - explicit model path
15         /// </summary>
16         /// <param name="args"></param>
17         static void Main(string[] args)
18         {
19
20             //Argument load stuff
21
22             if (!File.Exists(args[0]))
23                 throw new ArgumentException("Provided image path is not
                       valid");
24
25             string imagePath = args[0];
26             string modelPath = "model.txt";
27             string labelPath = "labels.txt";
28
29             if (args.Length > 1)
30             {
31                 if (File.Exists(args[1]))
32                 {
33                     modelPath = args[1];
34                 }
35                 else
36                 {
37                     throw new ArgumentException("Provided model path is
                           not valid");
38                 }
39             }
40
41             if (args.Length > 2)
42             {
43                 if (File.Exists(args[2]))
44                 {
45                     labelPath = args[2];
46                 }
47                 else
48                 {
49                     throw new ArgumentException("Provided model path is
                           not valid");
50                 }
```

```
51              }
52
53
54
55              string[] labels = File.ReadAllLines(labelPath);
56
57
58
59              //We assume that we use depth 3 images (RGB)
60              Network net = Network.LoadNetworkFromFile(modelPath);
61              double[] imageData = LoadImage(imagePath, 50, 25);
62              net.PushInputValues(imageData);
63              List<double> output = net.GetOutput();
64              int predictedIndex = output.MaxAt();
65
66              Console.WriteLine("Predicted movie: " + labels[
                    predictedIndex] + " with " + output[predictedIndex] + "%
                    positiveness");
67
68
69
70          }
71
72          private static double[] LoadImage(string path, int targetWidth,
                int targetHeight)
73          {
74              double[] data = new double[targetWidth * targetHeight * 3];
75              DataPreparer.ImageDataPreparer.PrepareImage(path,
                    targetWidth, targetHeight).data.CopyTo(data, 0);
76              return data;
77          }
78
79
80
81      }
82  }
```

Loader.cs

```
using System.IO;
using System;
using System.Linq;
using NeuralNetwork;
using System.Collections.Generic;

namespace NeuralNetwork.Tests
{

    public class Loader
    {
        /// <summary>
        /// Test loader, loads data.csv file
        /// Use only for simple tests
        /// </summary>
        /// <param name="path"></param>
        /// <returns></returns>
        public static double[][][] Load(string path)
        {
            /*data:
            [0] -> Input Data to be evaluated
            [1] -> Expected Output Data
            [2] -> Test Input Data
            [3] -> Test Output Data*/
            double[][][] finalData = new double[4][][];

            List<double[]> learningInputData = new List<double[]>();
            List<double[]> learningOutputData = new List<double[]>();
            List<double[]> testInputData = new List<double[]>();
            List<double[]> testOutputData = new List<double[]>();

            string[] lines = File.ReadAllLines(path).Skip(1).ToArray();
                 //Start from second line
            Shuffler.Shuffle(lines); //randomize data order

            for (int i = 0; i < lines.Length; i++)
            {
                string[] tokens = lines[i].Split(",");
                double[] data = new double[4];
                double[] output = new double[2];


                //Load data
                for (int j = 0; j < 4; j++)
                {
                    data[j] = Convert.ToDouble(tokens[j]);
                }

                //Load class
                if (tokens[4] == "0")
                {
                    output = new double[] { 1.0, 0.0 };
                }
```

```
53                  else if (tokens[4] == "1")
54                  {
55                      output = new double[] { 0.0, 1.0 };
56                  }
57                  else
58                  {
59                      throw new Exception("Error while reading data file:
                            Unrecognized object class");
60                  }
61
62
63                  if (i % 3 == 0) //take 30% of data as test data
64                  {
65                      testInputData.Add(data);
66                      testOutputData.Add(output);
67                  }
68                  else //take 70% as learning data
69                  {
70                      learningInputData.Add(data);
71                      learningOutputData.Add(output);
72                  }
73              }
74
75
76              //Pack everything
77              finalData[0] = learningInputData.ToArray();
78              learningInputData.Clear();
79              finalData[1] = learningOutputData.ToArray();
80              learningOutputData.Clear();
81              finalData[2] = testInputData.ToArray();
82              testInputData.Clear();
83              finalData[3] = testOutputData.ToArray();
84              testOutputData.Clear();
85
86              //Normalize data arrays (not output arrays as those are
                    already normalized)
87              for (int i = 0; i < 4; i++)
88              {
89               //   finalData[0].SetColumn(Normalizator.Normalize(
                        finalData[0].GetColumn(i), 0.0, 1.0), i);
90                 // finalData[2].SetColumn(Normalizator.Normalize(
                        finalData[2].GetColumn(i), 0.0, 1.0), i);
91              }
92
93
94
95
96              return finalData;
97          }
98      }
99  }
```

TestRecordTaking.cs

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using NeuralNetwork;

namespace NeuralNetwork.Tests
{

    [TestClass]
    public class TestRecordTaking
    {
        [TestMethod]
        public void TestHighestHitTest()
        {
            Network net = new Network(0.05, 0.5, -1.0, 1.0, 4, new int[]
                { 4, 4, 4 }, 2);
            //    net.testStrategy = new HighestHitTest(net);
            net.testStrategy = new HighestHitTest(net);

            double[][][] data = Loader.Load("data.csv");
            net.Train(data, 3000);

        }

        [TestMethod]
        public void TestMeanErrorTest()
        {
            Network net = new Network(0.05, 0.5, -1.0, 1.0, 4, new int[]
                { 4, 4, 4 }, 2);
            //    net.testStrategy = new HighestHitTest(net);
            net.testStrategy = new MeanErrorTest(net);

            double[][][] data = Loader.Load("data.csv");
            net.Train(data, 3000);

        }
    }
}
```

Loader.cs

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using System.IO;
4  using DataPreparer;
5  using NeuralNetwork;
6
7  namespace MovieClassifierLearner
8  {
9
10     public static class Loader
11     {
12         public static double[][][] Load(string path, int outputCount,
               int imageWidth, int imageHeight)
13         {
14             /*data:
15             [0] -> Input Data to be evaluated
16             [1] -> Expected Output Data
17             [2] -> Test Input Data
18             [3] -> Test Output Data*/
19             double[][][] finalData = new double[4][][];
20
21             List<double[]> inputData = new List<double[]>();
22             List<double[]> expectedOutputData = new List<double[]>();
23             List<double[]> testInputData = new List<double[]>();
24             List<double[]> testOutputData = new List<double[]>();
25             List<string> uniqueLabels = new List<string>();
26
27             { //Ensure that ImageLearningData[] will be disposed after
                   scope exit
28                 ImageLearningData[] data = ImageDataPreparer.
                       PrepareImages("Resources", imageWidth, imageHeight);
29                 //Pack data into double Data table
30
31
32                 for (int i = 0; i < data.Length; i++)
33                 {
34                     int labelIndex = uniqueLabels.IndexOf(data[i].label)
                           ;
35                     if (labelIndex == -1)
36                     {
37                         uniqueLabels.Add(data[i].label);
38                         labelIndex = uniqueLabels.Count - 1;
39                     }
40
41                     //Assign expected output
42                     double[] output = new double[outputCount];
43                     output[labelIndex] = 1.0;
44
45                     //Assign input values
46                     double[] input = data[i].data.ToArray();
47
48                     //Decide between test set and learning set
49                     if (i % 10 == 0)
```

```
                    {
                        testInputData.Add(input);
                        testOutputData.Add(output);
                    }
                    else
                    {
                        inputData.Add(input);
                        expectedOutputData.Add(output);
                    }
                }
            }

            //Shuffling
            int[] numbers = new int[inputData.Count];
            for (int i = 0; i < numbers.Length; i++) numbers[i] = i;
            Shuffler.Shuffle(numbers);
            List<double[]> tmpInputData = new List<double[]>();
            List<double[]> tmpOutputData = new List<double[]>();
            for (int i = 0; i < numbers.Length; i++)
            {
                tmpInputData.Add(inputData[numbers[i]]);
                tmpOutputData.Add(expectedOutputData[numbers[i]]);
            }
            inputData = tmpInputData;
            expectedOutputData = tmpOutputData;

            //Pack everything
            finalData[0] = inputData.ToArray();
            inputData.Clear();
            finalData[1] = expectedOutputData.ToArray();
            expectedOutputData.Clear();
            finalData[2] = testInputData.ToArray();
            testInputData.Clear();
            finalData[3] = testOutputData.ToArray();
            testOutputData.Clear();

            //Output labels
            File.WriteAllLines("labels.txt", uniqueLabels);

            return finalData;
        }

    }
}
```

Program.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.IO;
4  using DataPreparer;
5  using NeuralNetwork;
6  using System.Linq;
7  using System.Globalization;
8
9  namespace MovieClassifierLearner
10 {
11     class Program
12     {
13         static void Main(string[] args)
14         {
15             //Tensor dimensions
16             int imageWidth = 50;
17             int imageHeight = 25;
18             int imageDepth = 3; //number of colors
19
20             int outputCount = 5; // we need to know this in advance to
                   avoid back tracking through images
21
22             // args[0] - Learning Rate
23             // args[1] - Alpha in Bipolar Linear Function
24             // args[2] - Minimum Init Weight
25             // args[3] - Maximum Init Weight
26             // args[4+] - Hidden Neurons
27             int[] hiddenNeurons = new int[args.Length - 4];
28             for (int i = 4; i < args.Length; i++) hiddenNeurons[i - 4] =
                   Convert.ToInt32(args[i]);
29
30             Network net = new Network(ConvertUtil.ConvertArg(args[0]),
                   ConvertUtil.ConvertArg(args[1]),
31                 ConvertUtil.ConvertArg(args[2]), ConvertUtil.ConvertArg(
                       args[3]),
32                 imageWidth * imageHeight * imageDepth, hiddenNeurons,
                       outputCount);
33             //Network net = Network.LoadNetworkFromFile("
                   record_weights_HighestHitTest_0,74");
34             net.testStrategy = new HighestHitTest(net);
35
36             Console.WriteLine(" Loading data...");
37             double[][][] finalData = Loader.Load("Resources",
                   outputCount, imageWidth, imageHeight);
38
39             //net.RandomizeWeights();
40             ClassifyMovies(finalData, net);
41             net.Train(finalData, 2);
42             ClassifyMovies(finalData, net);
43         }
44
45         public static void ClassifyMovies(double[][][] finalData,
               Network network)
```

```
46          {
47              List<double> outputs; int correct = 0;
48              for (int i = 0; i < finalData[2].Length; i++)
49              {
50                  network.PushInputValues(finalData[2][i]);
51                  outputs = network.GetOutput();
52                  if (outputs.IndexOf(outputs.Max()) == finalData[3][i].
                        ToList().IndexOf(1)) correct += 1;
53              }
54              Console.WriteLine($" Correct ones: {correct}/{finalData[2].
                    Length} ");
55          }
56
57
58
59      }
60  }
```

Normalizator.cs

```
using System;

namespace ML.Lib
{
    public class Normalizator
    {
        static double Max(double[] input)
        {
            double result = double.MinValue;
            for (int i = 0; i < input.Length; i++)
            {
                if (result < input[i])
                    result = input[i];
            }
            return result;
        }

        static double Min(double[] input)
        {
            double result = double.MaxValue;
            for (int i = 0; i < input.Length; i++)
            {
                if (result > input[i])
                    result = input[i];
            }
            return result;
        }

        public static double[] Normalize(double[] input, double nmin,
            double nmax)
        {
            double[] result = new double[input.Length];
            double min = Min(input);
            double max = Max(input);

            for (int i = 0; i < input.Length; i++)
            {
                result[i] = ((input[i] - min) / (max - min)) * (nmax -
                    nmin) + nmin;
            }
            return result;
        }




    }
}
```

Shuffler.cs

```
using System;
using System.Collections.Generic;


namespace NeuralNetwork
{
    public class Shuffler
    {
        public static void Shuffle<T>(T[] input)
        {
            Random rand = new Random();
            for (int i = 0; i < input.Length; i++)
            {
                Swap<T>(input, i, rand.Next(0, input.Length - 1));
            }
        }

        static void Swap<T>(T[] input, int a, int b)
        {
            T buff = input[a];
            input[a] = input[b];
            input[b] = buff;
        }

    }
}
```

HighestHitTest.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace NeuralNetwork
5  {
6      public class HighestHitTest : ITestStrategy
7      {
8          private Network network;
9          private double maximumPercentageHalt;
10         private double recentPercentage;
11         public double CurrentRecord { get; private set; }
12
13
14         public HighestHitTest(Network network, double
               maximumPercentageHalt = 100)
15         {
16             this.network = network;
17             this.maximumPercentageHalt = maximumPercentageHalt;
18         }
19
20         public double Test(double[][] inputs, double[][] expectedOutputs
               )
21         {
22             double hitPercentage = 0;
23             int hits = 0;
24             List<double> outputs = new List<double>();
25             for (int i = 0; i < inputs.Length; i++)
26             {
27                 network.PushInputValues(inputs[i]);
28                 outputs = network.GetOutput();
29                 if (outputs.MaxAt() == expectedOutputs[i].MaxAt())
30                     hits++;
31             }
32             hitPercentage = (double)hits / (double)inputs.Length;
33             recentPercentage = hitPercentage;
34             Console.WriteLine($" Hit percentage : {Math.Round(
                   hitPercentage * 100.0, 3)}%");
35             return hitPercentage;
36         }
37
38         public bool CheckHalt()
39         {
40             return recentPercentage >= maximumPercentageHalt;
41         }
42
43         public bool CheckRecord()
44         {
45             if (CurrentRecord < recentPercentage)
46             {
47                 CurrentRecord = recentPercentage;
48                 return true;
49             }
50             return false;
```

```
51                 }
52             }
53
54 }
```

MeanErrorTest.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace NeuralNetwork
5  {
6      public class MeanErrorTest : ITestStrategy
7      {
8
9          private Network network;
10         private double minError;
11         public double MinError
12         {
13             get { return minError; }
14             set { minError = value; }
15         }
16
17         public double CurrentRecord { get; private set; }
18
19         private double recentError;
20
21
22         public MeanErrorTest(Network network, double minError = 0.001)
23         {
24             this.network = network;
25             this.minError = minError;
26             CurrentRecord = double.MaxValue;
27         }
28
29         public double Test(double[][] inputs, double[][] expectedOutputs
               )
30         {
31             double error = 0;
32             List<double> outputs = new List<double>();
33             for (int i = 0; i < inputs.Length; i++)
34             {
35                 network.PushInputValues(inputs[i]);
36                 outputs = network.GetOutput();
37                 error += Functions.CalculateError(outputs, i,
                       expectedOutputs);
38             }
39             error /= inputs.Length;
40             recentError = error;
41             Console.WriteLine($" Average mean square error: {Math.Round(
                   error, 5)}");
42
43             return error;
44         }
45
46         public bool CheckHalt()
47         {
48             return recentError <= minError;
49         }
50
```

```
51          public bool CheckRecord()
52          {
53              if (CurrentRecord > recentError)
54              {
55                  CurrentRecord = recentError;
56                  return true;
57              }
58              return false;
59          }
60      }
61
62  }
```

Network.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.IO;
5
6  namespace NeuralNetwork
7  {
8      public class Network
9      {
10         static double LearningRate { get; set; }
11         static double SynapsesCount { get; set; }
12         internal List<Layer> Layers;
13         internal double[][] ExpectedResult;
14         double[][] ErrorFunctionChanges;
15
16         public ITestStrategy testStrategy;
17         public bool TestHaltEnabled { get; set; }
18
19         public bool TestingEnabled { get; set; }
20
21         public bool RecordSaveEnabled { get; set; }
22
23         public Network(double learningrate, double alpha, double
               mininitweight, double maxinitweight, int numInputNeurons,
24         int[] hiddenLayerSizes, int numOutputNeurons, bool
               testHaltEnabled = false, bool testingEnabled = true, bool
               recordSaveEnabled = true)
25         {
26             Console.WriteLine("\n Building neural network...");
27             if (numInputNeurons < 1 || hiddenLayerSizes.Length < 1 ||
                   numOutputNeurons < 1)
28                 throw new Exception("Incorrect Network Parameters");
29
30             Functions.Alpha = alpha;
31             Synapse.MinInitWeight = mininitweight;
32             Synapse.MaxInitWeight = maxinitweight;
33             LearningRate = learningrate;
34             this.testStrategy = new MeanErrorTest(this);
35             this.TestHaltEnabled = testHaltEnabled;
36             this.TestingEnabled = testingEnabled;
37             this.RecordSaveEnabled = recordSaveEnabled;
38
39             Layers = new List<Layer>();
40             AddFirstLayer(numInputNeurons);
41             for (int i = 0; i < hiddenLayerSizes.Length; i++)
42                 AddNextLayer(new Layer(hiddenLayerSizes[i]));
43             AddNextLayer(new Layer(numOutputNeurons));
44
45             SynapsesCount = Synapse.SynapsesCount;
46
47             ErrorFunctionChanges = new double[Layers.Count][];
48             for (int i = 1; i < Layers.Count; i++)
49                 ErrorFunctionChanges[i] = new double[Layers[i].Neurons.
```

```
                Count];
50          }
51
52          private void AddFirstLayer(int inputneuronscount)
53          {
54              Layer inputlayer = new Layer(inputneuronscount);
55              foreach (Neuron neuron in inputlayer.Neurons)
56                  neuron.AddInputSynapse(0);
57              Layers.Add(inputlayer);
58          }
59
60          private void AddNextLayer(Layer newlayer)
61          {
62              Layer lastlayer = Layers[Layers.Count - 1];
63              lastlayer.ConnectLayers(newlayer);
64              Layers.Add(newlayer);
65          }
66
67          public void PushInputValues(double[] inputs)
68          {
69              if (inputs.Length != Layers[0].Neurons.Count)
70                  throw new Exception("Incorrect Input Size");
71
72              for (int i = 0; i < inputs.Length; i++)
73                  Layers[0].Neurons[i].PushValueOnInput(inputs[i]);
74          }
75
76          public void PushExpectedValues(double[][] expectedvalues)
77          {
78              if (expectedvalues[0].Length != Layers[Layers.Count - 1].
                    Neurons.Count)
79                  throw new Exception("Incorrect Expected Output Size");
80
81              ExpectedResult = expectedvalues;
82          }
83
84          public List<double> GetOutput()
85          {
86              List<double> output = new List<double>();
87              for (int i = 0; i < Layers.Count; i++)
88                  Layers[i].CalculateOutputOnLayer();
89              foreach (Neuron neuron in Layers[Layers.Count - 1].Neurons)
90                  output.Add(neuron.OutputValue);
91              return output;
92          }
93
94
95          /// <summary>
96          /// Trains network with given data
97          /// </summary>
98          /// <param name="data">
99          /// [0] -> Input Data to be evaluated
100         /// [1] -> Expected Output Data
101         /// [2] -> Test Input Data
102         /// [3] -> Test Output Data</param>
```

```
103         /// <param name="epochCount"></param>
104         public void Train(double[][][] data, int epochCount)
105         {
106             double[][] inputs = data[0], expectedOutputs = data[1];
107             double[][] testInputs = data[2], testOutputs = data[3];
108 
109             PushExpectedValues(expectedOutputs);
110 
111             Console.WriteLine(" Training neural network...");
112             for (int i = 0; i < epochCount; i++)
113             {
114                 List<double> outputs = new List<double>();
115                 for (int j = 0; j < inputs.Length; j++)
116                 {
117                     PushInputValues(inputs[j]);
118                     outputs = GetOutput();
119                     ChangeWeights(outputs, j);
120                 }
121 
122                 if (TestingEnabled == true)
123                 {
124                     testStrategy.Test(testInputs, testOutputs);
125                     if (testStrategy.CheckHalt() && TestHaltEnabled ==
                            true)
126                         break;
127                     if (testStrategy.CheckRecord() && RecordSaveEnabled
                            == true)
128                         SaveNetworkToFile(@"record_weights" + "_" +
                                testStrategy.GetType().Name.ToString() + "_"
                                + Math.Round(testStrategy.CurrentRecord, 2).
                                ToString() + ".txt");
129                 }
130             }
131         }
132 
133         public void RandomizeWeights()
134         {
135             for (int i = 1; i < Layers.Count; i++)
136             {
137                 Layers[i].RandomizeWeights();
138             }
139         }
140 
141         private void CalculateErrorFunctionChanges(List<double> outputs,
                int row)
142         {
143             for (int i = 0; i < Layers[Layers.Count - 1].Neurons.Count;
                    i++)
144                 ErrorFunctionChanges[Layers.Count - 1][i] = (
                        ExpectedResult[row][i] - outputs[i])
145                     * Functions.BipolarDifferential(Layers[Layers.Count
                            - 1].Neurons[i].InputValue);
146             for (int k = Layers.Count - 2; k > 0; k--)
147                 for (int i = 0; i < Layers[k].Neurons.Count; i++)
148                 {
```

```
149                     ErrorFunctionChanges[k][i] = 0;
150                     for (int j = 0; j < Layers[k + 1].Neurons.Count; j
                            ++)
151                         ErrorFunctionChanges[k][i] +=
                                ErrorFunctionChanges[k + 1][j] * Layers[k +
                                1].Neurons[j].Inputs[i].Weight;
152                     ErrorFunctionChanges[k][i] *= Functions.
                            BipolarDifferential(Layers[k].Neurons[i].
                            InputValue);
153                 }
154             }
155
156             private void ChangeWeights(List<double> outputs, int row)
157             {
158                 CalculateErrorFunctionChanges(outputs, row);
159                 for (int k = Layers.Count - 1; k > 0; k--)
160                     for (int i = 0; i < Layers[k].Neurons.Count; i++)
161                         for (int j = 0; j < Layers[k - 1].Neurons.Count; j
                                ++)
162                             Layers[k].Neurons[i].Inputs[j].Weight +=
163                                 LearningRate * 2 * ErrorFunctionChanges[k][i
                                    ] * Layers[k - 1].Neurons[j].OutputValue;
164             }
165
166             public void SaveNetworkToFile(string path)
167             {
168                 List<string> tmp = new List<string>();
169                 for (int i = 1; i < Layers.Count; i++)
170                     foreach (Neuron neuron in Layers[i].Neurons)
171                         foreach (Synapse synapse in neuron.Inputs)
172                             tmp.Add(synapse.Weight.ToString(CultureInfo.
                                    InvariantCulture));
173
174                 string build = $"{LearningRate.ToString()} {Functions.Alpha.
                        ToString()} {Synapse.MinInitWeight} {Synapse.
                        MaxInitWeight}";
175                 foreach (Layer layer in Layers) build += " " + layer.Neurons
                        .Count.ToString();
176                 tmp.Insert(0, build);
177                 File.WriteAllLines(path, tmp);
178             }
179
180
181             // loading from .txt file where in first line there are:
                    learning rate, alpha, minimum init weight,
182             // maximum init weight and sizes of all layers - all separated
                    by spaces; other lines are synapse weights
183             // (one per line)
184             public static Network LoadNetworkFromFile(string path)
185             {
186                 string[] lines = File.ReadAllLines(path);
187                 string[] firstLine = lines[0].Split();
188                 List<int> hiddenLayerSizes = new List<int>();
189                 for (int i = 5; i < firstLine.Length - 1; i++)
190                     hiddenLayerSizes.Add(Convert.ToInt32(firstLine[i]));
```

```
191
192             Network net = new Network(ConvertUtil.ConvertArg(firstLine
                    [0]), ConvertUtil.ConvertArg(firstLine[1]),
193                 ConvertUtil.ConvertArg(firstLine[2]), ConvertUtil.
                        ConvertArg(firstLine[3]), Convert.ToInt32(firstLine
                        [4]),
194                 hiddenLayerSizes.ToArray(), Convert.ToInt32(firstLine[
                        firstLine.Length - 1]));
195
196             Console.WriteLine(" Loading weights...");
197             if (lines.Length - 1 != SynapsesCount)
198                 Console.WriteLine(" Incorrect input file.");
199             else
200             {
201                 try
202                 {
203                     int i = 1;
204                     for (int j = 1; j < net.Layers.Count; j++)
205                         foreach (Neuron neuron in net.Layers[j].Neurons)
206                             foreach (Synapse synapse in neuron.Inputs)
207                                 synapse.Weight = ConvertUtil.ConvertArg(
                                        lines[i++]);
208                 }
209                 catch (Exception) { Console.WriteLine(" Incorrect input
                        file."); }
210             }
211             return net;
212         }
213
214
215         public int GetLayerSize(int layerIndex)
216         {
217             return Layers[layerIndex].Neurons.Count;
218         }
219     }
220 }
```

ConvertUtil.cs

```
1 using System;
2 using System.Globalization;
3
4 namespace NeuralNetwork
5 {
6     public static class ConvertUtil
7     {
8         public static double ConvertArg(string d)
9         {
10            return Double.Parse(d.Replace(',', '.'), CultureInfo.
                  InvariantCulture);
11        }
12    }
13
14 }
```

Neuron.cs

```
using System.Collections.Generic;

namespace NeuralNetwork
{
    class Neuron
    {
        public List<Synapse> Inputs { get; set; }
        public List<Synapse> Outputs { get; set; }
        public double InputValue { get; set; }
        public double OutputValue { get; set; }

        public Neuron()
        {
            Inputs = new List<Synapse>();
            Outputs = new List<Synapse>();
        }

        public void AddOutputNeuron(Neuron outputneuron)
        {
            Synapse synapse = new Synapse(this, outputneuron);
            Outputs.Add(synapse); outputneuron.Inputs.Add(synapse);
        }

        public void AddInputSynapse(double input)
        {
            Synapse syn = new Synapse(this, input);
            Inputs.Add(syn);
        }

        public void CalculateOutput()
        {
            InputValue = Functions.InputSumFunction(Inputs);
            OutputValue = Functions.BipolarLinearFunction(InputValue);
        }

        public void PushValueOnInput(double input)
        {
            Inputs[0].PushedData = input;
        }
    }
}
```

ITestStrategy.cs

```
namespace NeuralNetwork
{
    public interface ITestStrategy
    {
        double CurrentRecord {get;}
        double Test(double[][] input, double[][] expectedOutput);
        bool CheckHalt();
        bool CheckRecord();
    }

}
```

Synapse.cs

```
using System;

namespace NeuralNetwork
{
    class Synapse
    {
        static Random rnd = new Random();
        internal Neuron FromNeuron, ToNeuron;
        public double Weight { get; set; }
        public double PushedData { get; set; }
        public static int SynapsesCount { get; set; } = 0;
        public static double MaxInitWeight { get; set; }
        public static double MinInitWeight { get; set; }

        public Synapse(Neuron fromneuron, Neuron toneuron) // standard
            synapse
        {
            FromNeuron = fromneuron; ToNeuron = toneuron;
            Weight = rnd.NextDouble() * (MaxInitWeight - MinInitWeight)
                + MinInitWeight;
            SynapsesCount += 1;
        }

        public Synapse(Neuron toneuron, double data) // input synapse
            for first layer
        {
            ToNeuron = toneuron; PushedData = data;
            Weight = 1;
        }

        public double GetOutput()
        {
            if (FromNeuron == null) return PushedData; // if it is first
                layer
            return FromNeuron.OutputValue * Weight;
        }
    }
}
```

Functions.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace NeuralNetwork
5  {
6      class Functions
7      {
8          public static double Alpha { get; set; }
9
10         public static double CalculateError(List<double> outputs, int
               row, double[][] expectedresults) // objective function
11         {
12             double error = 0;
13             for (int i = 0; i < outputs.Count; i++)
14                 error += Math.Pow(outputs[i] - expectedresults[row][i],
                       2);
15             return error;
16         }
17
18         public static double InputSumFunction(List<Synapse> Inputs)
19             // input function: sum of products of synapses' weights and
                   neurons' outputs
20         {
21             double input = 0;
22             foreach (Synapse syn in Inputs)
23                 input += syn.GetOutput();
24             return input;
25         }
26
27         public static double BipolarLinearFunction(double input) //
               activation function...
28             => (1 - Math.Pow(Math.E, -Alpha * input)) / (1 + Math.Pow(
                   Math.E, -Alpha * input));
29
30         public static double BipolarDifferential(double input) // ...
               and her differential
31             => (2 * Alpha * Math.Pow(Math.E, -Alpha * input)) / (Math.
                   Pow(1 + Math.Pow(Math.E, -Alpha * input), 2));
32     }
33 }
```

ListExtensions.cs

```
using System;
using System.Collections.Generic;

namespace NeuralNetwork
{
    public static class ListExtensions
    {

        public static int MaxAt<T>(this IList<T> set) where T :
            IComparable<T>
        {
            T maxValue = set[0];
            int index = 0;
            for (int i = 0; i < set.Count; i++)
            {
                if (maxValue.CompareTo(set[i]) < 0)
                {
                    index = i;
                    maxValue = set[i];
                }
            }
            return index;
        }

    }

}
```

ArrayExtensions.cs

```
using System;

namespace ML.Lib
{
    public static class ArrayExtensions
    {

        //Gets "columnIndex" column from given set
        //This function expects that the set is an rectangular matrix
        public static T[] GetColumn<T>(this T[][] set, int columnIndex)
        {
            T[] result = new T[set.Length];
            for (int i = 0; i < set.Length; i++)
            {
                result[i] = set[i][columnIndex];
            }
            return result;
        }


        //Sets "columnIndex"-column in set with given "column"
        //This method expects that the set is an rectangular matrix
        //And that the given data does not exceed any
        public static void SetColumn<T>(this T[][] set, T[] column, int
            columnIndex)
        {
            for (int i = 0; i < set.Length; i++)
            {
                set[i][columnIndex] = column[i];
            }
        }

        //returns index of maximum element
        public static int MaxAt<T>(this T[] set) where T : IComparable<T
            >
        {
            T maxValue = set[0];
            int index = 0;
            for (int i = 0; i < set.Length; i++)
            {
                if (maxValue.CompareTo(set[i]) < 0)
                {
                    index = i;
                    maxValue = set[i];
                }
            }
            return index;
        }
    }
}
```

Layer.cs

```
using System;
using System.Collections.Generic;

namespace NeuralNetwork
{
    class Layer
    {

        private static Random r = new Random();
        public List<Neuron> Neurons;

        public Layer(int numberofneurons)
        {
            Neurons = new List<Neuron>();
            for (int i = 0; i < numberofneurons; i++)
                Neurons.Add(new Neuron());
        }

        public void ConnectLayers(Layer outputlayer)
        {
            foreach (Neuron thisneuron in Neurons)
                foreach (Neuron thatneuron in outputlayer.Neurons)
                    thisneuron.AddOutputNeuron(thatneuron);
        }

        public void CalculateOutputOnLayer()
        {
            foreach (Neuron neuron in Neurons)
                neuron.CalculateOutput();
        }


        /// <summary>
        /// Randomizes weights using Box Muller distribution algorithm
        /// Based on: http://neuralnetworksanddeeplearning.com/chap3.
            html#weight_initialization
        /// </summary>
        public void RandomizeWeights()
        {
            foreach (Neuron n in Neurons)
            {
                foreach (Synapse s in n.Inputs)
                {
                    s.Weight = r.NextDouble() * Math.Sqrt(2.0 / (Neurons
                        .Count + n.Inputs.Count));
                }
            }
        }
    }
}
```

screen_maker.py

```
import os

moviename = ""
def mkdir(name):
    os.system("mkdir "+name)
    global moviename
    moviename = name

def screenshot(name):
    os.system("scrot -z "+os.path.dirname(os.path.abspath(__file__))+"/"
        +moviename+"/"+name+".png") # z argument prevents beeping
```

---

netflix_controls.py

```
from pymouse import PyMouse
import time
playX = 50
playY = 1020

forwardX = 260
forwardY = 1020

centerX = 200
centerY = 200


m = PyMouse()



def wait_for_hide():
    m.move(x=centerX,y=centerY)
    time.sleep(4)

def play_pause():
    m.click(button=1,x=playX,y=playY)
    time.sleep(0.52)

def forward(n=1):
    for i in range(n):
        m.click(button=1,x=forwardX,y=forwardY)
        time.sleep(0.52)
```

script.py

```
import netflix_controls as nc
import screen_maker as sm
import sys
import time

begin_offset = 0 #seconds

name = ""
try:
    name = sys.argv[1]
except:
    print("no movie name given!")

if name is not "":
    print("starting taking screenshots for movie: "+name)
    print("leave stopped movie on fulscreen at 0:0")
    for i in range(10,-1,-1):
        time.sleep(1)
        print("starting in "+str(i)+" !")

    print("here we go!!!")

    sm.mkdir(name)

    nc.play_pause()
    nc.play_pause()
    nc.forward(int(begin_offset/10))

    i = 0
    while True:
        nc.wait_for_hide()
        sm.screenshot(name+"_"+str(i))
        nc.forward()
        nc.forward()
        i+=1
```

randomize.py

```
import sys
from random import uniform
lines = open(sys.argv[1], "r").readlines()
h = lines[0]
lines = lines[1:]
r = float(sys.argv[2])
print(h,end = "")
for i in lines:
    print(float(i)+uniform(-r,r))
```

extend.py

```
import sys
from random import randint
lines = open(sys.argv[1], "r").readlines()
h = lines[0]
lines = lines[1:]
num = int(sys.argv[2])

print(h,end = "")
s = len(lines)

for i in range(num):
    lines.append(lines[randint(0,s)])

for i in lines:
        print(float(i))
```
