Opis i porównanie wybranych algorytmów uczenia maszynowego na przykładzie klasyfikacji cyfr pisanych odręcznie

Piotr Sieński 184297, Sebastian Leśniewski 184711

May 30, 2022

1 Wstęp

Problemem, który staraliśmy się rozwiązać przy pomocy Sztucznej Inteligencji było rozpoznawanie ręcznie pisanych cyfr. W tym celu skorzystaliśmy z data setu MINST, który oferował 60,000 przykładów treningowych oraz 10,000 przykładów testowych.Pliki zawarte w data secie to monochromatyczne cyfry o rozmiarach 28x28 pikseli, gdzie na każdy piksel przypada jeden bajt informacji o kolorze (0-255) co daje około 55MB danych nie wliczając etykiet. Wartości pikseli są w dalszym etapie jednak normalizowane i przyjmują wartości -1 dla dolnej połowy zakresu i 1 dla górnej połowy zakresu.Do realizacji zadania zastosowaliśmy trzy metody: SVM,Sieć Neuronowa oraz SoftMax Regression i w dalszej części sprawozdania opisane zostały matematyczne podstawy działania tych metod i wyniki zastosowania ich do problemu klasyfikacji.

2 Opisy i wyniki działania metod

2.1 SVM

2.1.1 Opis algorytmu SVM

Celem algorytmu SVM jest znalezienie takich parametrów w, b, na podstawie których algorytm będzie poprawnie klasyfikował binarnie dane według funkcji decyzyjnej $h(x) = wx + b$ z jak największym marginesem od powierzchni decyzyjnej, margines ten dla i-tego elementu wektora danych można zdefiniować jako

$$\gamma^{(i)} = y^{(i)}(w^T x^{(i)} + b) \tag{1}$$

i margines dla całego zestawu danych jako

$$\gamma = min\ \gamma^{(i)}$$

Cel optymalizacji można sformułować więc jako

$$max\ \gamma$$

pod warunkiem

$$\forall_i \gamma^{(i)} \geq \gamma$$

Należy wziąć również pod uwagę, że w sformułowanym powyżej problemie zwiększenie w i b o stałą zwiększałoby również margines, czemu zaradzić można poprzez maksymalizowanie marginesu podzielonego przez normę wekotra w, dzięki temu możemy dowolnie zmieniać w, b (więc również

ustalić dowolną poszukiwana wartość γ), co prowadzi do kolejnego uproszczenia problemu poprzez ustawienie $\gamma = 1$. Otrzymujemy więc:

$$max\ \frac{1}{||w||} \Leftrightarrow min\ \frac{1}{2}||w||^2 \tag{2}$$

pod warunkiem

$$\forall_i \gamma^{(i)} \geq 1 \tag{3}$$

Na podstawie [1] w można przedstawić jako kombinację liniową $w = \sum^i \alpha_i y_i x_i$ pod warunkiem $\sum^i \alpha_i y_i = 0$, więc cel optymalizacji można przepisać jako:

$$min\ \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j (x_i \cdot x_j)\alpha_i \alpha_j - \sum_{i=1}^{N}\alpha_i \tag{4}$$

pod warunkiem

$$\forall_i \alpha_i \geq 0\ ,\quad \sum_{i=1}^{N} y_i \alpha_i = 0 \tag{5}$$

W celu umożliwienia algorytmowi radzenie sobie z danymi których nie da się odseparować wprowadzane jest kolejne ograniczenie na α,

$$\alpha \leq C$$

gdzie C jest parametrem umożliwiającym nie poprawną klasyfikację przypadków w celu osiągnięcia poprawnego marginesu.

Chcąc uzyskać nieliniowe powierzchnie decyzyjne można użyć funkcji jądrowych (kernel functions) mierzących podobieństwo / dystans między dwoma wektorami cech. Funkcje jądrowe efektywnie realizują podniesienie wymiarowości danych wejściowych bez konieczności robienia tego explicite. Finalnie cel optymalizacji wygląda następująco :

$$min\ \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j K(x_i, x_j)\alpha_i \alpha_j - \sum_{i=1}^{N}\alpha_i \tag{6}$$

pod warunkiem

$$\forall_i\ 0 \leq \alpha_i \leq C\ ,\quad \sum_{i=1}^{N} y_i \alpha_i = 0 \tag{7}$$

gdzie K jest funkcją jądrową.

A funkcję decyzyjną można sformułować jako :

$$h(x) = \sum_{i=1}^{N} y_i \alpha_i K(x_i, x) + b \tag{8}$$

W oparciu o [2] można stwierdzić, że problem optymalizacyjny jest rozwiązany wtedy, gdy spełnione są Warunki Karusha–Kuhna–Tuckera (KKT) dla każdego α. Warnuki KKT dla tego problemu to :

$$\alpha_i = 0 \Leftrightarrow y_i h(x_i) \geq 1$$

$$0 < \alpha_i < C \Leftrightarrow y_i h(x_i) = 1$$

$$\alpha_i = C \Leftrightarrow y_i h(x_i) \leq 1$$

2.1.2 Algorytm SMO

Algorytm SMO w każdej iteracji pętli wybiera najmniejszy możliwy do rozwiązania problem optymalizacyjny - wybiera dwa α i znajduje wartość optymalną dla obu.

Obliczanie nowych wartości α i b Z wybranych α_i, α_j jako pierwsza będzie ustalana wartość dla α_j. Najpierw należy obliczyć w jakich granicach powinna mieścić się ta wartość, tak aby warunki (7) były spełnione.

Jeśli $y_j = y_i$:

$$L = max(0, \alpha_2 - \alpha_1), \quad H = min(C, C + \alpha_2 - \alpha_1) \tag{9}$$

W przeciwnym wypadku :

$$L = max(0, \alpha_2 + \alpha_1 - C), \quad H = min(C, C + \alpha_2 + \alpha_1) \tag{10}$$

Następnie minimalizujemy funkcję (6) po α_i, α_j:

$$min_{\alpha_j,\alpha_i} \frac{1}{2}K(x_j,x_j)\alpha_j + \frac{1}{2}K(x_i,x_i)\alpha_i + \frac{1}{2}y_j\alpha_j\sum_{k\neq j} y_k\alpha_k K(x_j,x_k) + \frac{1}{2}y_i\alpha_i\sum_{k\neq i} y_k\alpha_k K(x_i,x_k) - \alpha_j - \alpha_j \tag{11}$$

Otrzymaliśmy równanie kwadratowe dwóch zmiennych postaci

$$Ax^2 + By^2 + Cx + Dy + Exy + F$$

które ma minimum

$$x_m = \frac{DE - 2BC}{\eta} \tag{12}$$

wtedy i tylko wtedy gdy

$$4AB - E^2 = \eta > 0 \tag{13}$$

Jeśli

$$\eta = 2K(x_i,x_j) - K(x_j,x_j) - K(x_i,x_i) > 0 \tag{14}$$

α_j ustawiane jest według (12) na

$$\alpha_j += \frac{y_j(E_i - E_j)}{\eta} \tag{15}$$

, gdzie

$$E_i = h(x_i) - y_i \tag{16}$$

i ewentualnie przycinane do granic ustaloych w (9) lub (10).

Jeśli zaś $\eta \leq 0$ funkcja celu jest ewaluowana w górnym i dolnym ograniczeniu i α_j jest ustawiane na ten koniec przedziału w którym ma ona mniejszą wartosć

Po ustawieniu α_j sprawdzane jest czy poczyniona została jakakolwiek znacząca zmiana (większa od pewnej ustalonej tolerancji), jeśli tak, to α_i ustawiane jest na

$$\alpha_1 += y_1y_2(\alpha_{2,old} - \alpha_{2,new}) \tag{17}$$

w celu spełnienia ograniczeń.

Po zmianie wartości α należy wyliczyć nowe b, tak aby warunki KKT były spełnione.

Jeśli $0 \leq \alpha_i \leq C$

$$b = b_1 = b - E_i - y^{(i)}(\alpha_{i,new} - \alpha_{i,old})\langle x^{(i)}, x^{(i)}\rangle - y^{(j)}(\alpha_{j,new} - \alpha_{j,old})K(x^{(i)}, x^{(j)}) \tag{18}$$

Jeśli $0 \leq \alpha_j \leq C$

$$b = b_2 = b - E_j - y^{(i)}(\alpha_{i,new} - \alpha_{i,old})K(x^{(i)}, x^{(j)}) - y^{(j)}(\alpha_{j,new} - \alpha_{j,old})K(x^{(j)}, x^{(j)}) \tag{19}$$

Jeśli $0 \leq \alpha_j \leq C$ i $0 \leq \alpha_i \leq C$, wtedy $b_1 = b_2$

Jeśli zaś oba α_i, α_j są ograniczone wtedy wszystkie b pomiędzy b_1 i b_2 będą spełniały warunki KKT, więc wybieramy

$$b = \frac{b_1 + b_2}{2} \tag{20}$$

Wybór α do optymalizacji Podczas pracy algorytmu w pamięci przechowywane są zbiory α - nieograniczonych ($0 < \alpha_i < C$) i ograniczonych ($\alpha_i = C \vee \alpha_i = 0$).

Wybór pierwszej wartości Algorytm na przemian wykonuje następujące kroki dopóki możliwe jest wykonanie jakiejkolwiek znaczącej zmiany:

1) Iteruje po wszystkich elementach zbioru treningowego i z podejmuje próbę optymalizacji j-tego elementu jeśli α_j narusza warunki KKT.

2) Iteruje po wszystkich nieograniczonych α podejmując próby optymalizacji j-tego elementu jeśli α_j narusza warunki KKT.

Algorytm powtarza krok 2) dopóki rezultatem jego jest znacząca zmiana chociaż jednej pary α, gdyż to właśnie nieograniczone α mają największe prawdopodobieństwo naruszania warunków KKT. Spełnienie warunków KKT jest sprawdzane do pewnej tolerancji.

Wybór drugiej wartości Po wybraniu pierwszego α drugie musi zostać wybrane w taki sposób, aby maksymalizować zmianę w funkcji celu. Najpierw podejmowana jest próba wyboru α_i według następującej heurystyki : chcemy żeby moduł z licznika z równania (15) był jak największy, więc jeśli $E_j < 0$ wybieramy α_i o największym błędzie, a gdy $E_j > 0$ wybieramy α_i o najmniejszym błędzie. Jeśli przy użyciu α_i wybranego na podstawie powyższej heurystyki nie może zostać poczyniony postęp, α_i które umożliwi poczynienie postępu szukane jest pośród nieograniczonych α, a w przypadku porażki pośród ograniczonych α. Oba wspomniane poszukiwania zaczynają się losowym miejscu obu zbiorów.

2.1.3 Szczegóły implementacji

Klasa SMO inicjalizowana jest przy podaniu znormalizowanej do przedziału <-1, 1> macierzy treningowej i wektora etykiet zawierającego wartości {-1, 1}, parametru funkcji jądrowej gamma, parametru C i opcjonalnej, obliczonej wcześniej macierzy kernel_matrix, takiej, że kernel_matrix[i, j] = kernel_function(x[i], x[j], gamma). Przy inicjalizacji klasy ustawiane są następujące pola:

wektor zer alpha, b = 0,

eps - tolerancja numeryczna potrzebna do określenia czy dane α uznać za zmienione w danej iteracji,

zbiory bound_alphas (wszystkie alpha), unbound_alphas (pusty),

cache błędów - mające przyspieszyć wyznaczanie błędów E_i,

unbound_err_cache - zbiór potrzebny do wyznaczania heurystycznie wartości potrzebnych w podczas wyboru drugiego α do optymalizacji,

alpha_metadata - tablica zawierająca dane o każdym z α - czy jest on ograniczony i czy wartość jego błędu jest w cache.

Ze względu na fakt, że zbiór danych na którym ma uczyć się algorytm jest relatywnie duży (macierz danych o wymiarach (12000 x 784)) W celu przyspiesznia obliczeń podczas poszukiwania optymalnych parametrów do algorytmu wprowadzona została możliwość obliczenia wcześniej macierzy kernel_matrix i uniknięcia obliczania relatywnie kosztownej obliczeniowo kernel_function podczas uczenia algorytmu.

[ ]:
```
    def __init__(self, x, y, c, gamma, km):
        self.b = 0
        self.alpha = np.zeros((x.shape[0], 1))
        self.features = x
        self.labels = y.reshape(len(y), 1)
        self.c = c
        self.gamma = gamma
        self.eps = 10**(-3)
        self.unbound_alphas = set()
        self.bound_alphas = set(range(len(self.alpha)))
        self.err_cache = {}
        self.unbound_err_cache = {}
        self.alpha_metadata = [SMO.AlphaMetadata() for _ in self.alpha]

        if km is None:
            self.train_hypothesis = self.no_km_hypothesis
        else:
            self.kernel_matrix = km
            self.train_hypothesis = self.km_hypothesis
```

Wybór pierwszej wartości α do optymalizacji realizowany jest w głównej pętli funkcji **train**

[ ]:
```
    while iters < max_iters and (changed_alphas > 0 or examine_all):
                changed_alphas = 0
                if examine_all:
                    for i in range(self.features.shape[0]):
                        changed_alphas += self.examine_example(i, tol)
                else:
                    set_cpy = copy.copy(self.unbound_alphas)
                    for i in set_cpy:
                        changed_alphas += self.examine_example(i, tol)
```

```
                if examine_all:
                    examine_all = False
                elif changed_alphas == 0:
                    examine_all = True
                iters += 1
```

Sprawdzenie czy dane α_j naursza warunki KKT(z podaną jako argument tolerancją) i wybór α_i realizuje funkcja **examine_example**

[ ]:
```
    def examine_example(self, i, tol):
        E_i = self.get_error(i)
        if (self.labels[i] * E_i < -tol and self.alpha[i] < self.c) or \
                (self.labels[i] * E_i > tol and self.alpha[i] > 0):
            if len(self.unbound_err_cache) != 0:
                idx = self.choice_cheuristic(i)
                if self.take_step(i, idx) == 1:
                    return 1

            tmp_list = list(self.unbound_alphas)
            start = random.randint(0, len(tmp_list))
            for j in range(len(tmp_list)):
                if self.take_step(i, tmp_list[(j + start) % len(tmp_list)]) == 1:
                    return 1

            tmp_list = list(self.bound_alphas)
            start = random.randint(0, len(tmp_list))
            for j in range(len(tmp_list)):
                if self.take_step(i, tmp_list[(j + start) % len(tmp_list)]) == 1:
                    return 1

        return 0
```

Próba optymalizacji obu α podejmowana jest w funkcji **take_step**

[ ]:
```
    def take_step(self, i, j):
        if i == j:
            return 0
        l, h = self.calculate_constrains(i, j)
        if l == h:
            return 0
        old_a_i = copy.deepcopy(self.alpha[i])
        old_a_j = copy.deepcopy(self.alpha[j])

        k_ii = self.kernel_function(self.features[i], self.features[i], self.
↪gamma)
        k_jj = self.kernel_function(self.features[j], self.features[j], self.
↪gamma)
```

```
        k_ij = self.kernel_function(self.features[i], self.features[j], self.
↪gamma)

        eta = 2*k_ij - k_ii - k_jj

        if eta < 0:
            new_a_j = self.update_alpha_j(i, j, eta)
        else:
            Lobj, Hobj = self.objective_function_at_bounds(i, j, k_ii, k_jj,␣
↪k_ij, l, h)
            if Lobj < Hobj - self.eps:
                new_a_j = l
            elif Lobj > Hobj + self.eps:
                new_a_j = h
            else:
                new_a_j = self.alpha[j]

        if abs(self.alpha[j] - new_a_j) < self.eps:
            return 0

        self.alpha[j] = new_a_j

        self.check_idx_bounds(j)

        self.alpha[i] += self.labels[i] * self.labels[j] * (old_a_j - self.
↪alpha[j])

        self.check_idx_bounds(i)

        self.calculate_b(i, j, old_a_i, old_a_j)
        return 1
```

Używana powyżej funkcja **check_idx_bounds** usuwa z cache błędy zmienionych α, pilnuje poprawności zbiorów bound_alphas i unbound_alphas oraz tablicy alpha_metadata. Funkcja **update_alpha_j** realizuje obliczenia (15), **calculate_constrains** obliczenia (9), (1), a **self.calculate_b** obliczenia (18) – (20)

2.1.4 SVM rozpoznający wiele klas

Klasyfikacja wielu klas odbywa się według podejścia one-vs-rest, tworzona jest oddzielna instancja SVM dla każdej z klas, jako dane dla każdej z nich przekazywana jest pewna część zestawu danych zawierająca wszystkie elementy z etykietą oznaczającą klasę jaka ma być rozpoznawana przez dany SVM oraz taką samą liczbę losowo wybranych elementów z etykietą nie zawierającą tej klasy. Podejście takie zostało wybrane poprzez nie korzystne skalowanie się algorytmu do dużych danych wejściowych i pozwala ono na znaczne przyspieszenie uczenia niewielkim kosztem.

2.1.5 Parametry

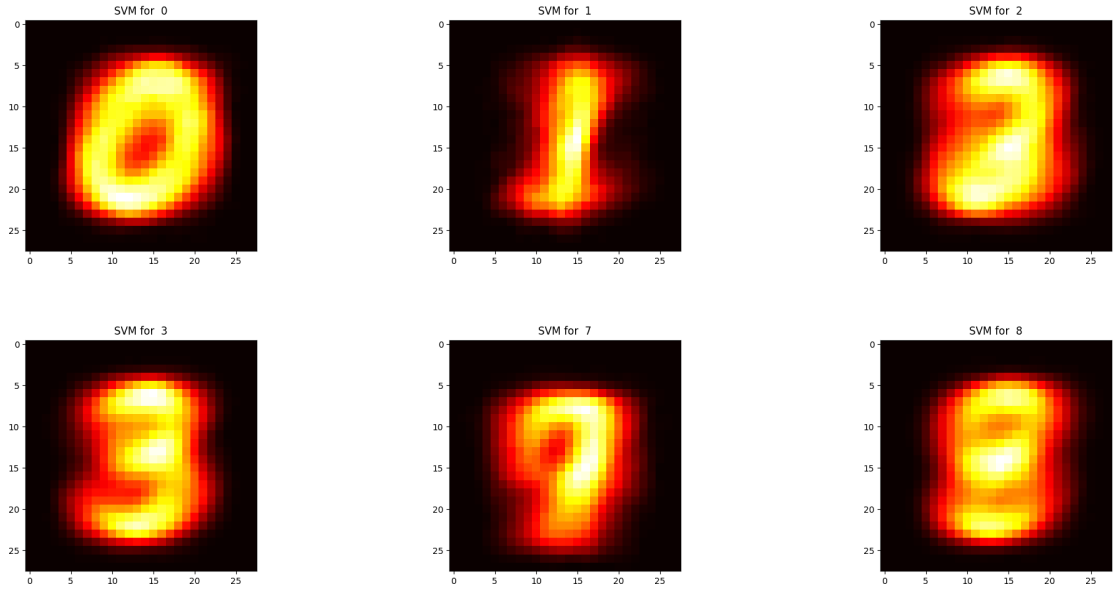
Parametry jakie należy podać do algorytmu są to : parametr gamma funkcji jądrowej, parametr regularyzacji C oraz tolerancja odnośnie spełnienia warunków KKT. Optymalny zestaw parametrów został wybrany w sposób eksperymentalny poprzez testowanie wyników działania algorytmu dla różnych zestawów parametrów. Testowane zestawy parametrów obejmowały $C \in \{10, 5, 1, 0.1, 0.05, 0.01\}$ i $\gamma \in \{10, 100, 1000, 5000, 10000\}$ dla tolerancji $tol \in \{0.01, 0.005, 0.001\}$. Można zaobserowować zależność, że algorytm osiąga największą dokładność dla parametrów γ i C pozostających w stosunku około $\frac{C}{\gamma} = \frac{1}{100}$ i osiąga największą dokładność dla $\gamma = 100$ i $C = 1$. Można również zauważyć, że czas działania (ilość iteracji) jest odwrotnie proporcjonalny do tolerancji, aczkolwiek podczas testów ustawiona była tolerancja $tol = 0.005$ w celu utrzymania akceptowalnego czasu obliczeń.

Dla każdego z SVM (dla każdej z cyfr) zostały użyte te same parametry.

2.1.6 Wyniki Działania

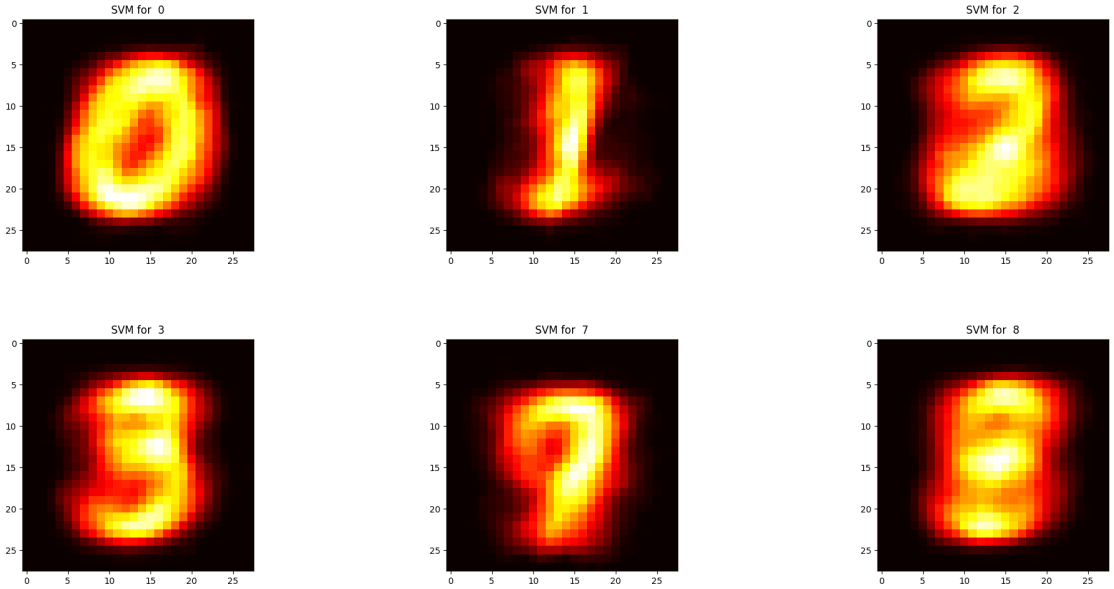
Najlepsze dopasowane parametry Dla parametrów $C = 1.0$, $\gamma = 100.0$ ***algorytm poprawnie sklasyfikował 98.7% obrazów*** z zestawu testowego. Przy czym każdy z SVM poprawnie sklasyfikował 96-99% danych treningowych.

Wizualizacje macierzy $\alpha \cdot X$



Porównanie z wynikami dla parametrów mniej optymalnych Dla porównania wynik dla mniej optymalnych parametrów $C = 5.0$, $\gamma = 100.0$ wynosił 32.22%. Przy czym każdy z SVM poprawnie sklasyfikował 53-91% danych treningowych.

Wizualizacje macierzy $\alpha \cdot X$



2.1.7 Źródła

1. Burges, C. J. C., "A Tutorial on Support Vector Machines for Pattern Recognition" (https://www.di.ens.fr/~mallat/papiers/svmtutorial.pdf)
2. Platt, J.C. "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines" (https://web.iitd.ac.in/~sumeet/tr-98-14.pdf)

2.2 Softmax Regression

2.2.1 Opis algorytmu

Softmax regression jest algorytmem będącym generalizacją regresji logistycznej (może być też interpretowany jako sieć neuronowa bez warstw ukrytych ze znormalizowanymi aktywacjami). Uczenie odbywa się poprzez gradientową minimalizację funkcji kosztu

$$J(\Theta) = -\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)} = k\} log \frac{exp(\Theta^{(k)T} x^{(i)})}{\sum_{j=1}^{K} exp(\Theta^{(j)T} x^{(i)})} + \lambda ||\Theta||^2$$

, gdzie Θ jest wektorem parametrów o wymiarach (n+1, k), gdzie n jest liczbą cech wektora wejściowego, a k liczbą klas i $\Theta^{(k)}$ oznacza wektor Θ dla k-tej klasy, a λ jest parametrem określającym regularyzację.

Funkcja decyzyjna $h(x)$ zwraca wektor, którego i-ty element jest prawdopodobieństwem, że x należy do klasy i

$$h(x) = \frac{1}{\sum_{j=1}^{K} exp(\Theta^{(j)T} x)} \cdot \begin{bmatrix} exp(\Theta^{(1)T} x) \\ \vdots \\ exp(\Theta^{(K)T} x) \end{bmatrix}$$

A predykcja może być dokonywana jako wybranie największego elementu wektora zwróconego przez $h(x)$

2.2.2 Uczenie

Uczenie algorytmu odbywa się metodą gradientową poprzez odejmowanie od Θ wektora pochodnych cząstkowych po funkcji kosztu, przesuwając się w stronę minimum globalnego funkcji kosztu. Odbywa się to partiami (mini batch gradient descent). Krok pętli uczącej można zapisać jako:

$$\Theta -= \frac{\alpha}{B} \cdot \nabla_{\Theta} J(\Theta)$$

gdzie α jest współczynnikiem uczenia, a B jest rozmiarem "partii" danych (batch), $\nabla_{\Theta} J(\Theta)$ jest wektorem pochodnych cząstkowych i dla danego x wynosi :

$$\nabla_{\Theta} J(\Theta) = x^T \cdot (h(x) - y) + \lambda\Theta$$

2.2.3 Szczegóły implementacji

Wszystkie obliczenia wykonywane są na wetkorach, więc implementacja mini batch gradient descent wygląda następująco:

[ ]:
```
 for c in range(iters):
            for i in range(0, y.shape[0]):
                if i + batch_size < y.shape[0]:
                    self.train_batch(x[i:i+batch_size, :], y[i:i+batch_size, :],
                                     learning_rate, reg_param, batch_size)
                else:
                    self.train_batch(x[i:, :].reshape(y.shape[0] - i, x.
 ↪shape[1]),
                                     y[i:, :].reshape(y.shape[0] - i,  self.k),
                                     learning_rate, reg_param, y.shape[0] - i)

def train_batch(self, x, y, learning_rate, reg_param, batch_size):
        self.Theta -= learning_rate * self.gradient(np.c_[np.ones(x.shape[0]),␣
 ↪x], y, reg_param) / y.shape[0]
```

Do każdego wywołania funkcji **gradient** czy funkcji decyzyjnej przekazywana jest macierz x, do której należy dodać wektor jedynek, w celu umożliwienia algorytmowi wprowadzenia bias term do obliczeń. Macierz Θ jest inicjalizowana uwzględniając bias term na rozmiar (n+1, k)

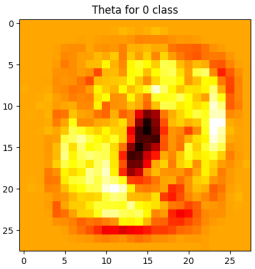
2.2.4 Parametry

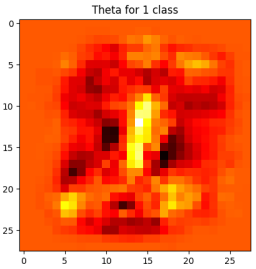
Parametry jakie należy podać do algorytmu są to : parametr określający szybkość uczenia alpha oraz parametr określający poziom regularyzacji. Prędkość zbiegania się algorytmu do optymalnego rozwiązania jest uzależniona od parametru α - jeśli jest on zbyt mały czas działania algorytmu będzie niepotrzebnie wydłużony, zaś jeśli jest on zbyt duży algorytm może nie osiągnąć w ogóle rozwiązania optymalnego. Parametr λ określa jak dobrze algorytm generalizuje w porównaniu do przykładów na których się uczył, dla zbyt dużego λ widoczne będzie zjawisko underfitingu, zaś dla zbyt małego overfitingu.

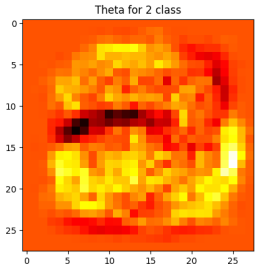
2.2.5 Wyniki Działania

Najlepsze dopasowane parametry Dla parametrów $\alpha = 0.001$, $\lambda = 0.001$ ***algorytm poprawnie sklasyfikował 91.51% obrazów*** z zestawu testowego.

Wizualizacje parametrów θ algorytrmu







2.3 Neural Network

2.3.1 Opis algorytmu

W opisywanej sieci neuronowej wszystkie warstwy są w pełni połączone, a aktywacjami neuronów w poszczególnych warstwach mogą być funkcje

$$sigmoid(x) = \frac{1}{1 + exp(-x)}$$

$$ReLU(x) = max0, x$$

Celem optymalizacji jest minimalizacja funkcji kosztu

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[-y_k^{(i)}log(h(x^{(i)})_k) - (1-y_k^{(i)})log(1-(h(x^{(i)})_k))] + \frac{\lambda}{2}\sum_{d=1}^{D}\sum_{l}^{a_d}\sum_{k}^{b_d}(w_{lk}^{[d]})^2$$

, gdzie $ h(x) $ są aktywacjami ostatniej warstwy neuronów, D jest liczbą warstw, a a_d, b_d są rozmiarami macierzy wag dla warstwy d, λ jest parametrem regularyzacji

Aktywacje ostatniej warstwy ($h(x)$) obliczane są poprzez algorytm propagacji w przód, a predykcja może być dokonywana jako wybranie największego elementu wektora ($h(x)$), uczenie sieci odbywa się przy użyciu algorytmu propagacji w tył.

2.3.2 Propagacja w przód

Algorytm propagacji w przód zaczyna swoją pracę na danych wejściowych do sieci i oblicza wartości aktywacji w kolejnych warstwach. Poszukujemy wartości z, a dla każdej z warstw.

$$z_i = w_i a_{i-1} + b_i, \quad a_i = activation_i(z_i)$$

, gdzie w_i jest macierzą wag w i-tej warstwy, a_i jest wektorem aktywacji w danej warstwie. Zakładamy, że $a_0 = x$

2.3.3 Propagacja w tył

Celem propagacji w tył jest wyznaczanie $\frac{\partial J}{\partial w^{[k]}}$ i $\frac{\partial J}{\partial b^{[k]}}$ dla każdej z warstw, żeby w sposób gradientowy aktualizować wagi. Obliczenia należy zacząć od warstwy ostatniej i korzystając z reguły łańcuchowej pochodnych posuwać się w tył do osiągnięcia pierwszej warstwy.

$$\frac{\partial J}{\partial w^{[k]}} = \frac{\partial J}{\partial z^{[k]}} a^{[k-1]^T}, \quad \frac{\partial J}{\partial b^{[k]}} = \frac{\partial J}{\partial z^{[k]}}$$

Przyjmując oznaczenie $\delta^{[k]} = \frac{\partial J}{\partial z^{[k]}}$ i r jako liczbę warstw

$$\delta^{[r]} = \frac{\partial J}{\partial z^{[r]}} = a^{[r]} - y$$

$$\forall_{k<r}\ \delta^{[k]} = (w^{[k+1]^T} \delta^{[k+1]}) \odot activation_k{}'(z^{[k]})$$

gdzie $activation_k{}'$ jest pochodną funkcji aktywacji w k-tej warstwie, a $ $\odot$oznaczamnoeniepoelementach$

Można więc podać regułę aktualizacji dla w, b jako

$$w_k \mathrel{-}= \frac{\alpha}{B} \delta^{[k]} a^{[k-1]^T} + \lambda w_k, \quad b_k \mathrel{-}= \frac{\alpha}{B} \delta^{[k]}$$

gdzie α jest współczynnikiem uczenia, a B jest rozmiarem "partii" danych (batch)

2.3.4 Szczegóły implementacji

Poczas inicjalizacji klasy NeuralNet należy podać liczbę neuronów wejściowych, wyjściowych, architekturę warstw ukrytych oraz funkcje aktywacji na poszczególnych warstwach, włączając ostatnią. Wagi inicjalizowane są jako wartości losowe z dystrybucji normalnej o wartości średniej równej 0 i wariancji równej $\frac{1}{\sqrt{d}}$, gdzie d jest wielkością warstwy poprzedniej

[ ]:
```
    def initialize_weights(self, input_size, layers_layout):
        prev_layer_size = input_size
        for layer_size in layers_layout:
            self.weights.append(np.random.randn(prev_layer_size, layer_size) *␣
↪np.sqrt(1/prev_layer_size))
            prev_layer_size = layer_size
```

Ze względu na wektoryzację kodu propagacji w tył i przód możliwa jest implementacja uczenia przy użyciu mini batch gradient descent przy użyciu tej samej pętli głównej co SoftmaxRegression, z różnicą w funkcji **train batch**

[ ]:
```
    def train_batch(self, x, y, learning_rate, reg_param, batch_size):
        self.forward_prop(x)

        # backpropagation algorithm
        deltas = [0 for _ in range(self.layers_count)]
        deltas[-1] = self.activations[-1] - y
        for i in range(self.layers_count - 2, -1, -1):
```

```
            deltas[i] = np.transpose(self.weights[i + 1].dot(np.
↪transpose(deltas[i + 1])))
            deltas[i] = np.multiply(deltas[i], self.activation_derivative(self.
↪z[i], i))
        #                                                    l2 regularization
        self.weights[0] -= (np.transpose(x).dot(deltas[0]) + reg_param * self.
↪weights[0] * batch_size)\
                           * learning_rate/batch_size
        self.biases[0] -= np.sum(deltas[0], axis=0).reshape(self.biases[0].
↪shape) * learning_rate/batch_size
        for i in range(1, self.layers_count):
            #                                                          l2␣
↪regularization
            dw = np.transpose(self.activations[i-1]).dot(deltas[i]) + reg_param␣
↪* self.weights[i] * batch_size
            self.weights[i] -= dw * learning_rate/batch_size
            db = np.sum(deltas[i], axis=0).reshape(self.biases[i].shape)
            self.biases[i] -= db * learning_rate/batch_size
```

Funkcja **forward_prop** realizuje propagację w przód macierzy x

2.3.5 Parametry

Parametry jakie należy podać do algorytmu są to : architektura warstw ukrytych oraz parametr regularyzacji lambda. Parametr λ określa jak dobrze algorytm generalizuje w porównaniu do przykładów na których się uczył, dla zbyt dużego λ widoczne będzie zjawisko underfitingu, zaś dla zbyt małego overfitingu.

Dobór architektury sieci dokonywany jest w sposób eksperymentalny, aczkolwiek można zauważyć pewne zależności:

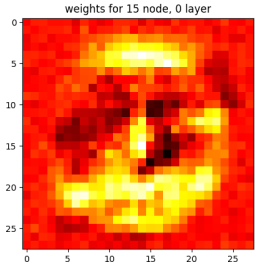
- wraz z liczbą warstw wydłuża się czas uczenia, co nie koniecznie jest skorelowane z lepszymi wynikami,
- rozmiary kolejnych warstw powinny maleć - tj. pierwsza warstwa ukryta powinna mieć najwięcej neuronów (acz maksymalnie w okolicy rozmiaru warstwy wejściowej) a ostania najmniej (acz nie mniej niż warstwa wyjściowa),
- warstwy mogą również mieć ten sam rozmiar, ale użycie wielu takich samych warstw o rozmiarze mniejszym niż dane wejściowe prowadzi do gorszych wyników niż jedna warstwa.
- aktywacja ReLU spisuje się lepiej w sieci wielowarstwowej, zaś sigmoid przy mniejszej liczbie warstw

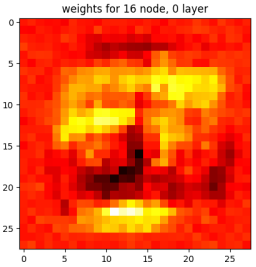
2.3.6 Wyniki Działania

Najlepsze dopasowane parametry Dla $\lambda = 0.001$ i 4 warstw ukrytych po 800, 400, 200, 50 neuronów z aktywacjami ReLU. ***Algorytm poprawnie sklasyfikował 95.4% obrazów*** z zestawu testowego.

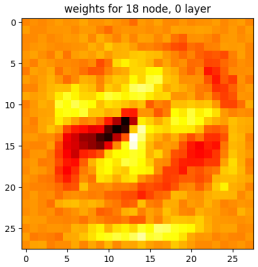
Przykładowa wizualizacja Próbując interpretować wyniki działania sieci neuronowej można spróbować wizualizować jaką wartość zależną od wag mają poszczególne pixele wejściowe w neuronach kolejnych warstw, co może być zrealizowane przez mnożenie kolejnych macierzy wag. Poniższe wizualizacje dotyczą sieci z jedną, 20 neuronową warstwą ukrytą osiągająca 91.2 % poprawnie sklasyfikowanych przypadków spośród danych testowych

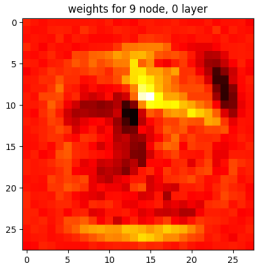
Warstwa ukryta

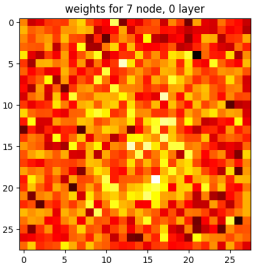




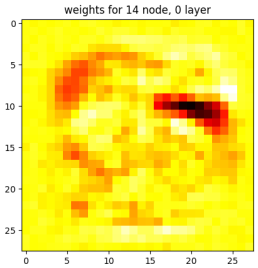
%



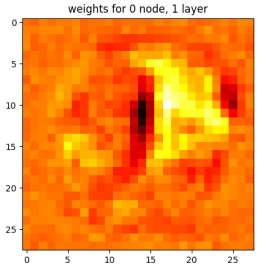


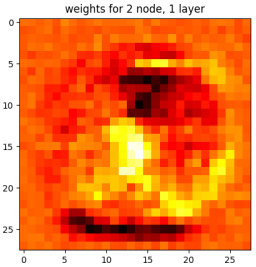


%

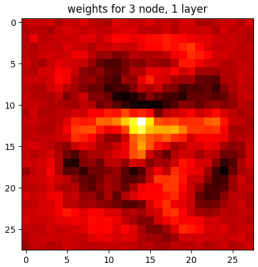


Warstwa wyjściowa





%



3 Podsumowanie

- SVM jest algorytmem relatywnie trudnym w implementacji, źle skalującym się, aczkolwiek jego wyniki są interpretowalne, posiada mało parametrów które należy dostosować i osiąga bardzo dobre wyniki klasyfikacji (przy odpowiednio niskiej tolerancji i przy operowaniu na całym zbiorze danych prawdopodobnie osiągnąłby lepszy wynik niż podany wyżej w opracowaniu)
- Regresja softmax jest prostym w implementacji, szybkim i łatwo interpretowalnym algorytmem nie osiągającym wyników porównywalnych do pozostałych dwóch
- Sieć neuronowa jest algorytmem wymagającym włożenia dużej ilości pracy w zaprojektowanie odpowiedniej architektury, a jego wyniki są ciężkie w interpretacji. Z drugiej strony osiągając bardzo dobre wyniki klasyfikacji, działa w rozsądnym czasie i relatywnie dobrze się skaluje (przy odpowiedniej architekturze prawdopodobnie algorytm osiągnąłby lepsze wyniki niż podane wyżej w opracowaniu)

Algorytm regresji softmax najszybciej znajduje rozwiązanie problemu, sieć neuronowa w różnym, zależnym od architektury, tempie, aczkolwiek zawsze wolniej niż regresja softmax. Czas działania obu tych algorytmów liczony jest w minutach SVM zaś w godzinach ze względu na słabe skalowanie do dużych zbiorów danych.

Podsumowując, można więc powiedzieć, że algorytmy SVM i sieci neuronowej znacznie przewyższają regresję softmax w zadaniu klasyfikacji, aczkolwiek są też od niej znacznie bardziej skomplikowane. Wyniki osiągnięte przez sieć neuronową i SVM są ciężkie do porównania, gdyż prawdopodobnie nie został włożony wystarczający wysiłek w zaprojektowanie architektury sieci. Można jednak stwierdzić, że algorytm sieci neuronowej działa znacznie szybciej od SVM, lepiej się skaluje i jest prostszy w implementacji kosztem skomplikowania w dobrze parametrów.