Klassifizierung von handgeschriebenen Ziffern

Kathrin Parchatka

11. Januar 2019

1 Einleitung

In diesem Paper wird im Rahmen des Zulassungsprojekts für das Modul 'Machine Learning' an der Heinrich Heine Universität ein Klassifizierungsproblem, in welchem handgeschriebene Ziffern richtig klassifiziert werden sollen, mit Hilfe von drei Algorithmen behandelt: mit einer k-Nearest-Neighbors Klassifizierung, einer Support Vector Maschine und mit einem Neuronalen Netz.

1.1 Datensatz

Ich verwende eine Kopie des UCI ML Repository Datensatzes 'Optical Recognition of Handwritten Digits Data Set' von E. Alpaydin aus dem Jahr 1990 [5], welche bei sci-kit learn unter dem Namen 'Digit Dataset' zur Verfügung steht [1]. Der Datensatz enthält Daten über handgeschriebene Ziffern von 0 bis 9, welche als 8×8-Pixel-Bild gespeichert sind. Dementsprechend besitzt der Datensatz 64 Attribute, die jeweils einen ganzzahligen Graustufen-Wert von 0 bis 16 enthalten. 0 steht für ein schwarzes Pixel und 16 für ein weißes Pixel.

Da Klassifizierung eine Form von Supervised Learning ist, gibt es im Datensatz neben den Graustufen-Werten der Pixel zu jeder Instanz ein Label. Dieses Label gibt an, welche Zahl zwischen 0 und 9 erkennbar ist. Der Datensatz enthält insgesamt 1797 Instanzen mit je 64 Attributen und je einem Label. In Abbildung 1 sind vier Beispiele von handgeschriebenen Ziffern dargestellt.

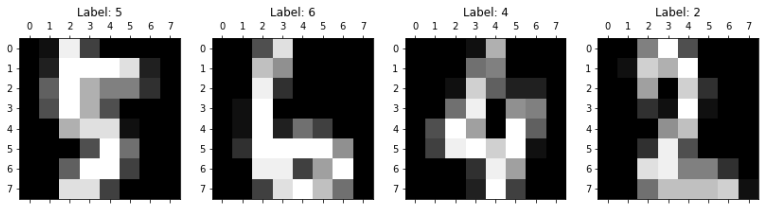

Abbildung 1: Beispiele der 8×8-Pixel-Bilder mit Labeln

Der Datensatz wird für die Klassifizierung in einen Trainings- und einen Testdatensatz aufgeteilt. Rund 85% der Daten werden als Trainingsdaten benutzt. Für die Validierung der Hyperparameter werden die Trainingsdaten jeodch weiterhin in 'Trainings-' und 'Testdaten' aufgeteilt. Um eine Verwechslung mit den tatsächlichen Testdaten zu verhindern, bezeichne ich die Datenmenge zur Validierung der Hyperparameter auf dem Trainingsdatensatz im Folgenden als Validierungsdaten.

1.2 Methodik

Zur Analyse und Klassifizierung des Datensatzes werden folgende Klassifizierungsalgorithmen behandelt: k-Nearest Neighbors Klassifizierung, Support Vector Machine Klassifizierung und Klassifizierung mit Neuronalen Netzen. Auf die Algorithmen und deren Ergebnisse wird im Folgenden genauer eingegangen und am Ende ein Fazit gegeben.

2 k-Nearest-Neighbors Klassifizierung

Die Klassifizierung durch den k-Nearest Neighbors Algorithmus erfolgt durch die Beachtung der Labels der k nächsten Nachbarn eines Datenpunkts. Der zu klassifizierende Datenpunkt erhält das Label mit dem häufigsten Vorkommen innerhalb dieser k Nachbarspunkte. Datenpunkte sind 'nahe' Nachbarn, wenn die Werte ihrer Attribute ähnlich sind. Daher kann optional zusätzlich die Entfernung der Nachbarn zu dem zu klassifizierenden Datenpunkt berücksichtigt werden. Denn die Wahrscheinlichkeit, dass ein Datenpunkt das gleiche Label hat wie das eines Nachbarspunkts, verringert sich mit immer größer werdenden Distanz des Nachbarn zum Datenpunkt.

Der Algorithmus, welcher in diesem Paper modelliert wird, hat daher zwei Hyperparameter: k, die Anzahl der 'nearest neighbors', und die Unterscheidung, ob der Einfluss der Nachbarn mit Distanzwerten gewichtet werden soll. Letzteres wird mit 'uniformly weighted neighbors' und 'distance weighted neighbors' bezeichnet.

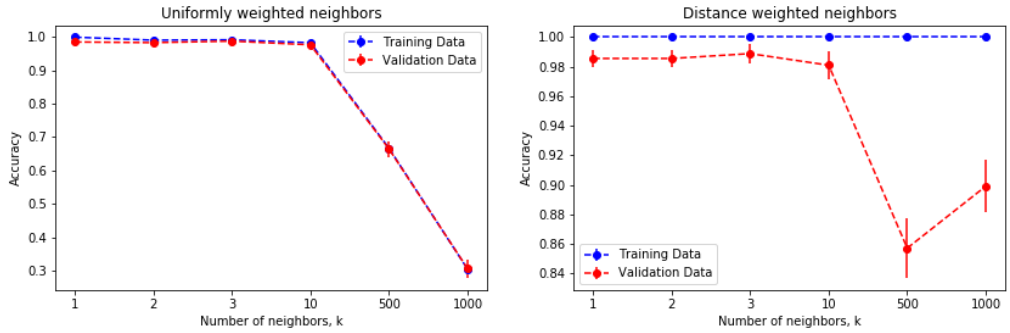

Abbildung 2: Ergebnisse der Genauigkeitsanalyse der k-Nearest Neighbors Klassifizierung ohne und mit Berücksichtigung der Distanz zum Nachbarspunkt

In Abbildung 2 sind die Trainings- und Validierungsergebnisse einer Genauigkeitsanalyse des Algorithmus für $k = 1, 2, 3, 10, 500$ und 1000 Nachbarn für 'uniformly weighted neighbors' und 'distance weighted neighbors' aufgezeichnet. Die blauen Punkte beschreiben die durchschnittliche Genauigkeit der Klassifizierung in den Trainingsdaten und die roten Punkte die Genauigkeit der Klassifizierung in den Validierungsdaten. Die vertikalen Linien an den Punkten kennzeichnen die Standardabweichung, während die gestrichelten Linien den ungefähren Verlauf der Genauigkeiten zwischen den Punkten skizzieren. Dabei ist zu beachten, dass die k-Achse nicht gleichmäßig skaliert ist.

Werden die Ergebnisse für den Algorithmus ohne Berücksichtigung der Distanz zu den Nachbarn betrachtet, so ist eindeutig zu erkennen, dass die Genauigkeit auf den Test- und Validierungsdaten mit steigender Anzahl an Nachbarspunkten abnimmt. Dies lässt sich dadurch erklären, dass mit steigender Anzahl an Nachbarspunkten auch mehr Punkte berücksichtigt werden, deren Werte ihrer 64 Attribute stark von den Werten des betrachteten Datenpunkts abweichen und damit wahrscheinlich ein anderes Label tragen. Wird die Distanz der Punkte jedoch berücksichtigt, so ist dieser Effekt wesentlich geringer erkennbar. Während die Genauigkeit der Klassifizierung ohne Berücksichtigung der Distanz für $k = 1000$ auf unter 30% sinkt, so ist sie bei Klassifizierung mit Berücksichtigung der Distanz bei über 80% in den Validierungsdaten und sogar über 99% in den Trainingsdaten.

2.1 Implementierung und Ergebnisse

Für die Implementierung des k-Nearest Neighbors Algorithmus wurde der `KNeighborsClassifier()` von scikit learn benutzt. Die Hyperparameterauswahl wurde per Cross Validation mit 5 folds mittels `GridSearchCV()` realisiert. Als Hyperparametervorschlag wurde die Menge

$$\{(k, w) \mid k \in \{1, 2, 3, 10, 500, 1000\} \text{ und } w \in \{\text{'uniformly weighted', 'distance weighted'}\}\}$$

vorgegeben. Das beste Ergebnis für die Validierungsdaten wurde für $k = 3$ mit $w =$ 'distance weighted' Nachbarn mit einer Genauigkeit von 98.89% erzielt. Diese Wahl der Hyperparameter führt zu einer Genauigkeit von 98.15% auf den Testdaten.

3 Support Vector Machine Klassifizierung

Die Klassifizierung durch eine Support Vector Machine (SVM) basiert auf Decision Boundaries, welche die Daten so gut wie möglich in die 10 Klassen separieren sollen. Das Aufstellen dieser Decision Boundaries erfolgt durch Lösen eines Optimierungsproblems, welches den Abstand der Support Vektoren zu diesen Decision Boundaries maximiert. Als Support Vektoren werden die Datenpunkte bezeichnet, die am nächsten an den Decision Boundaries liegen und damit auch aktiv in der Berechnung dieser Boundaries eingehen. Falls ein Datensatz nicht linear separierbar ist, so kann die SVM durch Kernel-Funktionen modifiziert werden. Mit Hilfe dieser Funktionen werden die Daten zunächst in einen Feature Space abgebildet, in welchem diese Daten wiederrum linear separierbar sind. Beispiele für Kernel Funktionen sind

1. linearer Kernel: $k(x, x') = x^T x'$
2. polynomieller Kernel: $k(x, x') = \left(\gamma \cdot x^T x' + r\right)^d$ mit $\gamma > 0, r =$ konst und $d \in \mathbb{N}$,
3. Radial-Basis-Function Kernel : $k(x, x') = \exp\left(-\gamma \left\|x - x'\right\|^2\right)$ mit $\gamma > 0$ und
4. Sigmoid Kernel: $k(x, x') = \tanh(\gamma \cdot x^T x' + r)$ mit $\gamma > 0$ und $r =$ konst

(vergleiche [2]). Zusätzlich kann das Optimierungsproblem durch den Regularisierungsparameter $C > 0$ beeinflusst werden, welcher den Einfluss eines Regularisierungsterms im Optimierungsproblem bestimmt. Der Regularisierungsterm hat die Funktion, ein Overfitting des Klassifizierers auf den Trainingsdaten zu verhindern. Demnach ergeben sich für die SVM Klassifizierung die Hyperparameter $C > 0$, $\gamma > 0$, $r \in \mathbb{R}$, $d \in \mathbb{N}$ und die Auswahl einer Kernelfunktion.

In Abbildung 3 sind die Trainings- und Validierungsergebnisse einer Genauigkeitsanalyse der SVM Klassifizierung für $C = 0.01$, 1, 3, 4, 5, 6, 7, 10, 100 und 1000, $\gamma = 0.0007$ und $r = 0$,1 für die obigen Kernelfunktionen dargestellt. Für den polynomiellen Kernel wurde die Betrachtung der Genauigkeit auf Polynome vom Grad 2 und 3 beschränkt. Die oberen Plots beinhalten die Genauigkeiten für $r = 0$ und die unteren Plots die Genauigkeiten für $r = 1$. Da der lineare Kernel und der RBF-Kernel unabhängig von r sind, stimmen die obigen gelben und grünen Kurven mit den unteren Kurven überein. Die Punkte beschreiben die durchschnittliche Trainings- und Validierungsgenauigkeit an den angegebenen C-Werten und die gestrichelten Linien skizzieren den ungefähren Verlauf der Genauigkeit zwischen den Punkten. Dabei ist zu beachten, dass die C-Achse nicht gleichmäßig skaliert ist.

Betrachtet man die Ergebnisse für den Klassifizierungsalgorithmus mit $r = 0$, so ist zu erkennen, dass die Klassifizierung mit Nutzung der Sigmoid-Kernelfunktion mit einer Trainings- und Validierungsgenauigkeit von unter 90% für alle Wahlen von C am schlechtesten ist. Für kleine C sinkt die Genauigkeit sogar bis auf unter 10% ab. Die Klassifizierung mit RBF-Kernelfunktion zeigt für kleine C ein ähnliches Verhalten. Sowohl die Trainings- als auch die Validierungsdaten haben für Werte $C \geq 1$ eine sehr hohe Genauigkeit von über 96%, wohingegen die Genauigkeit für $C = 0.01$ nur noch ca 19% beträgt. Die Genauigkeit der betrachteten polynomiellen Kernelfunktionen und der linearen Kernelfunktion zeigen auch sehr gute, stabile

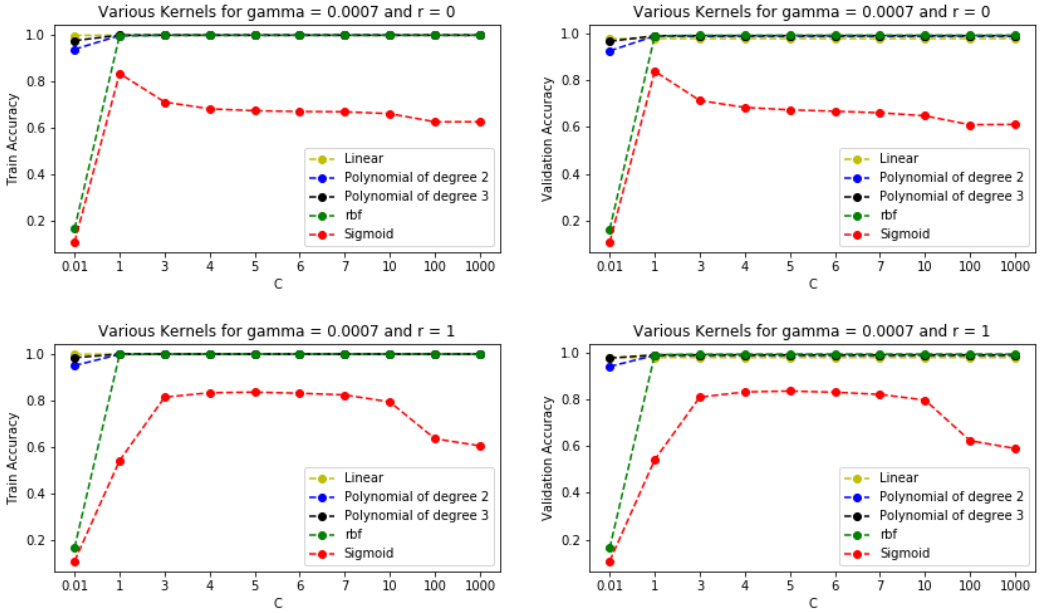

Abbildung 3: Ergebnisse der Genauigkeitsanalyse der SVM-Klassifizierung für verschiedene Wahlen von C

Trainings- und Validierungsgenauigkeiten von über 90%. Ähnliche Beobachtungen lassen sich in den Ergebnissen der Genauigkeitsanalyse für den Algorithmus mit $r = 1$ erkennen, weshalb nicht weiter auf die Plots eingegangen wird.

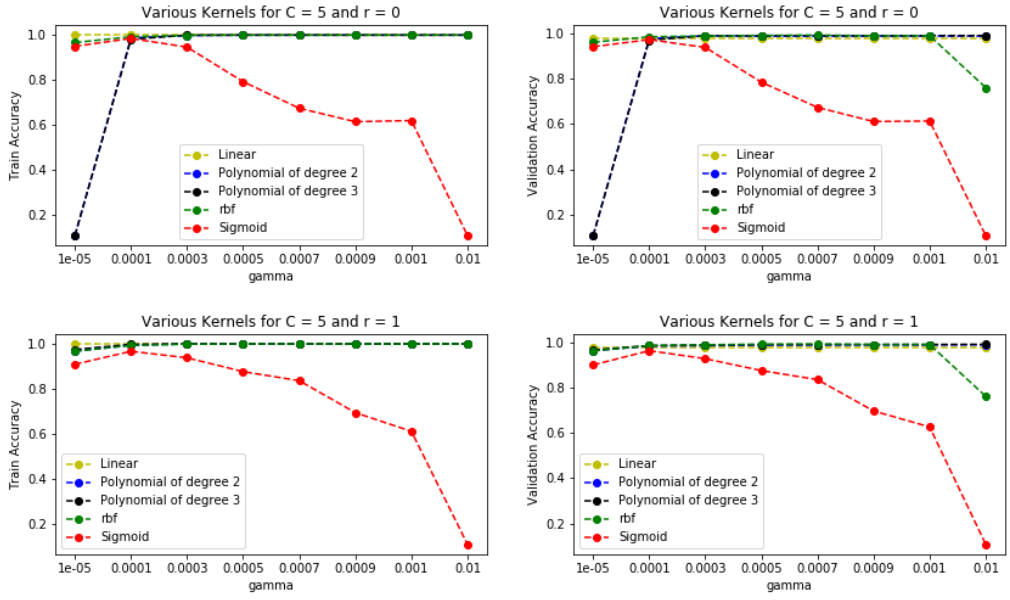

Abbildung 4: Ergebnisse der Genauigkeitsanalyse der SVM-Klassifizierung für verschiedene Wahlen von γ

In Abbildung 4 sind die Trainings- und Validierungsergebnisse einer Genauigkeitsanalyse des SVM Klassifizierers für $\gamma = 0.00001, 0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, 0.01$, $C = 5$ und $r = 0, 1$ für die

verschiedenen Kernelfunktionen dargestellt. Für den polynomiellen Kernel wurde auch hier die Betrachtung der Genauigkeit auf Polynome vom Grad 2 und 3 eingeschränkt. Die Anordnung der Plots und die Notation ist analog zu Abbildung 3. Da die lineare Kernelfunktion unabhängig von γ ist, entspricht die gelbe Kurve jeweils dem konstanten Genauigkeitswert für $C = 5$. Weil die RBF-Kernelfunktion unabhängig von r ist, entsprechen die oberen Kurven jeweils den unteren Kurven. Auch hier ist zu beachten, dass die γ-Achse nicht gleichmäßig skaliert ist.

Bei Betrachtung der Werte zur Sigmoid-Kernelfunktion ist zu erkennen, dass die Genauigkeiten der Trainings- und Validierungsergebnisse sowohl für $r = 0$ als auch für $r = 1$ mit steigendem γ-Wert kleiner werden und für $\gamma = 0.01$ sogar unter 10% betragen. Betrachtet man die Genauigkeit für Trainings- und Validierungsdaten für die polynomiellen Kernelfunktionen, so ist für $r = 0$ ab $\gamma \geq 0.0001$ eine gute Genauigkeit von über 95% erkennbar. Für $\gamma < 0.0001$ fällt diese Genauigkeit für beide polynomielle Kernelfunktionen jedoch stark auf unter 10% ab. Für $r = 1$ ist diese starke Senkung der Genauigkeit jedoch nicht mehr erkennbar. Letztlich ist die Genauigkeit bei Nutzung der RBF-Kernelfunktion für in den Validierungsdaten stabil und mit rund 98% sehr hoch. In den Validierungsdaten ist für großes $\gamma = 0.01$ jedoch ein Verlust an Genauigkeit sichtbar.

3.1 Implementierung und Ergebnisse

Für die Implementierung der SVM Klassifizierung wurde der `SVC()` Klassifizierer von sci-kit learn benutzt. Die Hyperparameterauswahl wurde per Cross Validation mit 5 folds mittels `GridSearchCV()` realisiert. Hyperparametervorschlag wurde die Menge

$$M_{\text{linear}} \cup M_{\text{rbf}} \cup M_{\text{sigmoid}} \cup M_{\text{poly}}$$

mit

$$\begin{aligned}
M_{\text{linear}} =& \{(C, k) \mid k = \text{'linear'}, C \in \{0.01, 1, 3, 4, 5, 6, 7, 10, 100, 1000\}\} \\
M_{\text{rbf}} =& \{(C, k, \gamma) \mid k = \text{'rbf'}, C \in \{0.01, 1, 3, 4, 5, 6, 7, 10, 100, 1000\}, \\
& \gamma \in \{0.00001, 0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, 0.01\}\} \\
M_{\text{sigmoid}} =& \{(C, k, \gamma, r) \mid k = \text{'sigmoid'}, C \in \{0.01, 1, 3, 4, 5, 6, 7, 10, 100, 1000\}, \\
& \gamma \in \{0.00001, 0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, 0.01\}, r \in \{0, 1\}\} \\
M_{\text{poly}} =& \{(C, k, \gamma, r, d) \mid k = \text{'poly'}, C \in \{0.01, 1, 3, 4, 5, 6, 7, 10, 100, 1000\}, \\
& \gamma \in \{0.00001, 0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, 0.01\}, r \in \{0, 1\}, d \in \{2, 3\}\}
\end{aligned}$$

vorgegeben. Das beste Ergebnis wurde mit der RBF-Kernelfunktion für $C = 5$ und $\gamma = 0.0007$ mit einer Validierungsgenauigkeit von 99.21% erzielt. Eine Klassifizierung mit dieser Wahl von Hyperparametern führt zu einer Genauigkeit von 98.89% auf den Testdaten.

4 Klassifizierung mit Neuronalen Netzen

Klassifizierung kann auch mit Hilfe von Neuronalen Netzen implementiert werden. Ein Neuronales Netz ist eine Abbildung, welche Daten auf einen Wert abbildet, der die Zugehörigkeit zu einer Klasse angibt. Diese Abbildung hat typischerweise eine Schichten-Struktur, die unter anderem aus Multiplikationen mit Gewichtsvektoren und einer im Allgemeinen nichtlinearen Abbildung bestehen kann. Diese Abbildung wird 'Aktivierungsfunktion' und die Schichten werden 'Hidden Layers' genannt. In meinem Neuronalen Netz nutze ich ausschließlich lineare Layer, welche durch Aktivierungsfunktionen voneinander getrennt sind. Sind die Daten klassifiziert, so wird draufhin eine 'Loss'-Funktion, die den Fehler zwischen der berechneten Klasse und der tatsächlichen Klasse bestimmt, mit Hilfe eines Gradientenabsteigverfahrens in den Gewichten minimiert. Die Minimierung der Loss-Funktion mit anschließender Anpasung der Gewichtsvektoren entspricht einer Verbesserung des Klassifizierers. Die Berechnung der Gradienten für den Gradientenabstieg

erfolgt durch Backpropagation. Die Gewichtsvektoren werden von dem Modell demnach durch das Gradientenabstiegsverfahren selbst 'gelernt', sodass nur noch folgende Hyperparameter für die Klassifizierung zu bestimmen sind: Anzahl und Größe der linearen Layer (Anzahl der Neuronen pro Layer), Aktivierungsfunktion, Auswahl eines Gradientenabstiegsverfahrens, Lernrate des Gradientenabstiegsverfahrens und ein Regularisierungsparameter α, welcher die Gewichtung eines Strafterms in der Loss-Function darstellt, um ein Overfitting des Modells auf den Trainingsdaten zu vermeiden. Beispiele für Aktivierungsfunktionen sind

1. Identität: $f(x) = x$
2. Logistic-Sigmoid-Funktion: $f(x) = 1/(1 + \exp(-x))$
3. Tangens Hyperbolicus: $f(x) = \tanh(x)$ und
4. ReLU: $f(x) = \max(0, x)$

(vergleiche [3]). Da ich ausschließlich lineare Layer nutze, ist die Identität als Akivierungsfunktion keine sinnvolle Wahl. Ich möchte, dass der Klassifizierer nichtlinear ist, um komplexere Features lernen zu können. Eine Verkettung von linearen Layern mit linearen Aktivierungsfunktionen resultiert dagegen lediglich in einer linearen Abbildung. Aus diesem Grund werden im Folgenden nur die Logistic-Sigmoid-Funktion, der Tangens Hyperbolicus und die ReLU-Aktivierungsfunktion betrachtet.

Für die Neuronalen Netze in diesem Paper wird ausschließlich das Quasi Newton Verfahren L-BFGS zur Minimierung der Loss-Funktion mit einer konstanten Lernrate von $\mu = 0.001$ benutzt, weil dieses Verfahren auf kleinen Datensätzen in der Regel schneller konvergiert und bessere Ergebnisse liefert. [4]

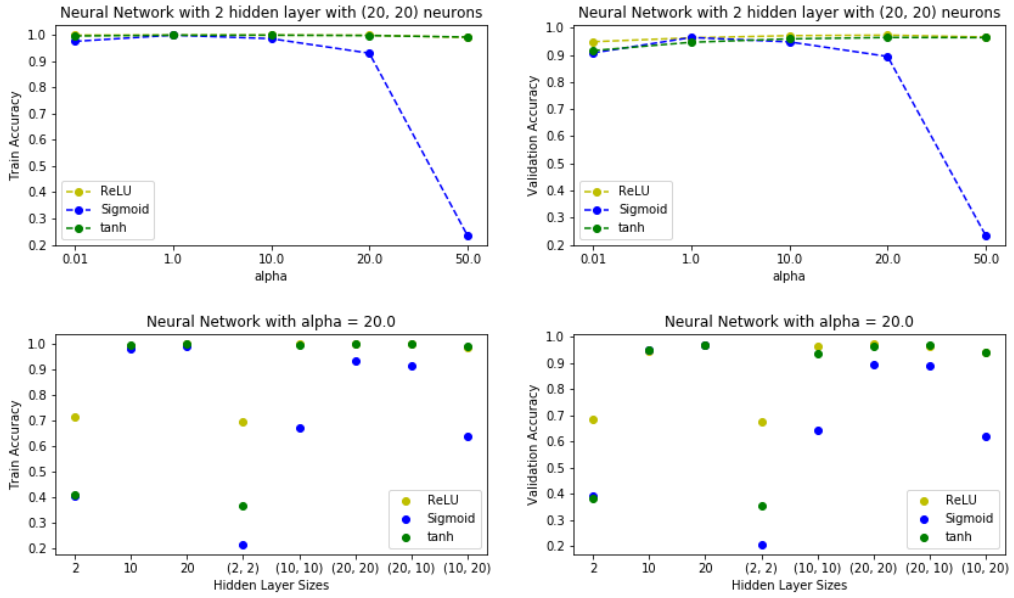

Abbildung 5: Ergebnisse der Genauigkeitsanalyse der Klassifizierung mit einem Neuronalen Netz für verschiedene Wahlen von α und diverse Anzahl und Größen von Hidden Layers

In Abbildung 5 sind auf den oberen Plots die Trainings- und Validierungergebnisse einer Genauigkeitsanalyse der Algorithmen für $\alpha = 0.01$, 1, 10, 20 und 50 aufgeführt. Auf den unteren Plots sind die Ergebnisse für Neuronale Netze mit einem Hidden Layer mit je 2, 10 und 20 Neuronen, und Neuronalen Netzen mit 2 Hidden Layers mit je 2, 10 und 20 Neuronen oder 2 Hidden Layers mit 20 Neuronen im ersten Layer und 10 Neuronen im zweiten Layer oder 10 Neuronen im ersten Layer und 20 Neuronen im zweiten Layer dargestellt.

Auf den oberen Plots beschreiben die Punkte die Genauigkeit der Klassifizierung für verschiedene Wahlen von α und auf den unteren Plots entsprechen die Punkte der Genauigkeit der Klassifizierung zu obigen Konbinationen von Anzahl und Größe von Hidden Layers. Die gestrichelten Linien skizzieren den ungefähren Verlauf der Genauigkeit, wobei bei den unteren Plots keine Ordnung auf der 'Hidden Layer Sizes'-Achse gegeben ist, weshalb eine Verbindung der Punkte keine Auskunft über den Verlauf der Genauigkeit geben würde. Dabei ist zu beachten, dass auch die α-Achse nicht gleichmäßig skaliert ist.

Die Genauigkeit auf den Trainingsdaten in einem Neuronalen Netz mit 2 Hidden Layers mit je 20 Neuronen ist für $0.01 \leq \alpha \leq 50$ für die Sigmoid- und ReLU-Aktivierungsfunktionen mit einem Wert von über 99% sehr gut. Der Klassifizierer mit dem Tangens Hyperbolicus als Aktivierungsfunktion verliert für großes α in den Trainingsdaten schnell an Genauigkeit mit einem Minimalwert von unter 24% für $\alpha = 50$ auf den Trainingsdaten. Die Genauigkeit der Klassifizierung auf den Validierungsdaten zeigt für alle drei Aktivierungsfunktionen ähnliche Effekte.

In den unteren Plots der Abbildung 5 ist erkennbar, dass die Klassifizierung mit einem Neuronalen Netz mit nur einem Hidden Layer Genauigkeit gewinnt, wenn die Anzahl der Neuronen im Hidden Layer vergrößert wird. Dies ist sowohl in den Trainings- als auch in den Validierungsdaten unabhängig von der Wahl der Aktiviterungsfunktion gut erkennbar. Derselbe Effekt tritt auch bei gleichzeitiger Vergrößerung der Anzahl der Neuronen in einem Neuronalen Netz mit zwei Hidden Layers auf. Hierbei ist deutlich zu erkennen, dass die Klassifizierung mit ReLU- und Tangens Hyperbolicus-Aktivierungsfunktionen im Vergleich zur Klassifizierung mit Sigmoid-Aktivierungsfunktionen bessere Genauigkeiten liefert. Werden weiterhin die Ergebnisse der Neuronalen Netze mit 2 Hidden Layers mit unterschiedlicher Anzahl an Neuronen betrachtet, so lässt sich vermuten, dass die Anzahl der Neuronen im ersten Hidden Layer die entscheidende Größe ist, die zu einer optimalen Genauigkeit führt. Während sich die Trainings- und Validierungsgenauigkeiten zwischen dem Neuronalen Netz mit je 20 Neuronen und dem Neuronalen Netz mit 20 Neuronen im ersten Layer und 10 Neuronen im zweiten Layer nur kaum unterscheiden, ist ein deutlicher Genauigkeitsverlust beim Vergleich des Neuronalen Netzes mit je 20 Neuronen mit dem Neuronalen Netz mit 10 Neuronen im ersten Layer und 20 Neuronen im zweiten Layer erkennbar.

4.1 Implementierung und Ergebnisse

Für die Implementierung der Klassifizierung mittels Neuronalen Netzen wurde der Multi-layer Perceptron Klassifizierer `MLPClassifier()` von sci-kit learn benutzt. Die Parameter `solver = 'lbfgs'` und `learning_rate ='constant'` wurden festgelegt, während die restlichen Hyperparameter per Cross Validation mit 5 folds mittels `GridSearchCV()` gewählt wurden. Als Hyperparametervorschlag wurde die Menge

$$\begin{aligned}\{(\text{act}, \text{hls}, \alpha) \mid &\text{act} \in \{\text{ReLU}(), \text{Sigmoid}(), \tanh()\},\\ &\text{hls} \in \{(2), (10), (20), (2,2), (10,10), (20,20), (20,10), (10,20)\},\\ &\alpha \in \{0.01, 1, 10, 20, 50\}\}\end{aligned}$$

vorgegeben. Das beste Ergebnis wurde mit der ReLU-Aktivierungsfunktion für $\alpha = 20$ in einem Neuronalen Netz mit zwei Hidden Layers mit je 20 Neuronen mit einer durchschnittlichen Validierungsgenauigkeit von 97.25% erzielt. Diese Wahl der Hyperparameter führt zu einer Genauigkeit von 96.3% auf den Testdaten.

5 Fazit

Insgesamt sind die Ergebnisse der Genauigkeiten auf den Testdaten für alle Klassifizierungsmethoden mit der richtigen Hyperparameterwahl sehr gut. Das Neuronale Netz hat eine Testgenauigkeit von über 96%, während der k-Nearest Neighbors Klassifizierer und der SVM-Klassifizierer sogar Testgenauigkeiten von über 98% aufzeigen. Bei der geringen Anzahl an 1797 Instanzen ist jedoch der k-Nearest Neighbor Klassifizierer wesentlich schneller als die Support Vector Machine und das Neuronale Netz. Die sehr gute Performance des k-Nearest Neighbors Klassifizierers lässt darauf schließen, dass das Problem der Erkennung von handgeschriebenen Ziffern für den gegebenen Datensatz sehr einfach zu lösen ist. Es scheint, dass keine komplexeren

Features zur Klassifizierung benötigt werden, weshalb der komplexere SVM-Klassifizierer und die Klassifizierung durch Neuronale Netze keine besseren Ergebnisse liefern.

6 Anforderungen

Um eine einwandfreie Ausführbarkeit des Quellcodes zu gewährleisten, wird empfohlen, die benötigten Pakete via pip zu installieren. Alle nötigen Pakete sind in der Datei `requirements.txt` codiert und können mit dem Kommandozeilenbefehl

```
pip install -r requirements.txt
```

installiert werden.

Im Notebook `DigitRecognition.ipynb` ist der zugehörige Python-Quellcode zu finden. Die Ausführung des Quellcodes kann wegen der hohen Anzahl an Kombinationen von Hyperparametern dauern. Zum Ausführen des Quellcodes muss mit ca 15-20 Minuten Ausführzeit gerechnet werden.

Literatur

[1] Digits Dataset. `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html#sklearn.datasets.load_digits`. Accessed: 2019-01-10.

[2] Kernel Functions, Support Vector Machines. `https://scikit-learn.org/stable/modules/svm.html`. Accessed: 2019-01-10.

[3] MLPClassifier. `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`. Accessed: 2019-01-10.

[4] Neural network models (supervised), Tips on Practical Use. `https://scikit-learn.org/stable/modules/neural_networks_supervised.html`. Accessed: 2019-01-10.

[5] Optical Recognition of Handwritten Digits. `http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits`. Accessed: 2019-01-10.