GCN

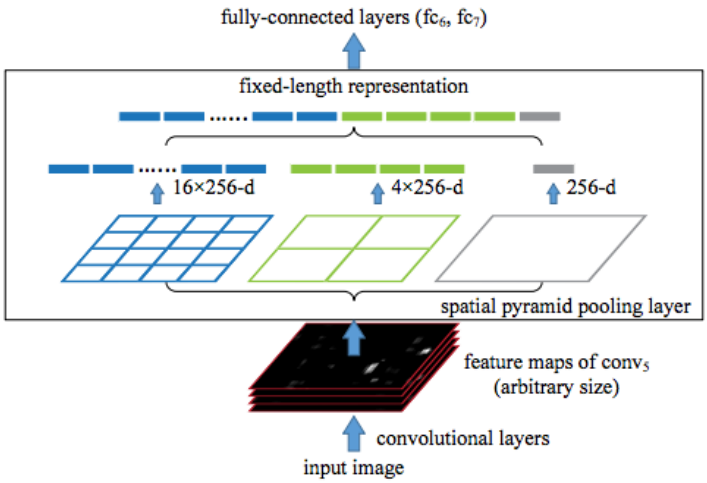
10. Februar 2017

1 CNNs auf variierenden Größen

Meistens werden Bilder für die Eingabe in ein CNN gecroppt oder gewarped, damit sie einer fixen Bildgröße (z.B. 244×244) entsprechen. Dabei gehen aber Bildteile verloren oder werden geometrisch gestreckt, so dass sie nicht mehr naturgetreu sind. Desweiteren helfen Bilder mit verschiedenen Größen dabei, skalierungsinvariant zu trainieren und reduzieren Overfitting. Warum brauchen CNNs eine fixe Größe? Ein CNN besteht aus zwei Teilen, Convolutional Layern und Fully-Connected Layern. Die Convolutional Layer opeieren über einem beweglichen Fenster. Convolutional Layer brauchen eigentlich keine fixe Größe und können Feature Maps beliebiger Größe generieren. Das ist der Ursache geschuldet, dass wir immer nur einen Bildausschnitt betrachten und shared Weights and Biase haben. Die Fully-Connected Layer allerdings haben diesen Vorteil nicht, sie brauchen stets eine feste Anzahl an Neuronen. Beim Übergang zwischen Convolution und Fully-Connected kriegen wir deshalb Probleme.

Ein Layer zwischen diesen beiden Teilen mit der Aufgabe, eine beliebige Größe von Feature Maps auf eine feste Anzahl an Neuronen zu mappen, könnte dieses Problem beseitigen. Dieser Layer wird auch SPP genannt (*Spatial Pyramid Pooling*).

1.1 SPP Layer



Aus der letzten Convolution (z.B. 256 Feature Maps, d.h. Shape `[-1, -1, 256]`) werden mehrere *Spatial Bins* verschiedener Größen über *Max-Pooling* berechnet. In jedem Bin werden die Features Maps gepoolt mit Größe kM, wobei M die Anzahl an Bins ist und k die Anzahl an Feature Maps.

Angenommen wir haben eine Bildgröße von 224×224 und ein CNN, dass uns nach der Convolution Shapes der Form `[13, 13, 256]` liefert mit $a \times a = 13 \times 13$. Sagen wir wir wollen 3 3 Bins aufbauen mit $l = 3$: Dann berechnen sich die Pooling Windows mit $\lceil \frac{a}{n} \rceil$ und Strides mit $\lfloor \frac{a}{n} \rfloor$ mit $n = 1, 2, 3$.

Dann können wir z.B. 3 mal Max Pooling auf diesen Daten anwenden:

1. $3 \times 3 \Rightarrow$ Size 5, Stride 4
2. $2 \times 2 \Rightarrow$ Size 7, Stride 6
3. $1 \times 1 \Rightarrow$ Size 13 Stride 13

Dann ergibt dies:

$$3 \times 3 \times 256 + 2 \times 2 \times 256 + 1 \times 1 \times 256 = 3584 \tag{1}$$

Diese Neuronen können wir nun mit unseren Fully Connected Neuronen verbinden.

Jetzt testen wir noch eine unterschiedliche Bildgröße von 180×180. Dann hat das CNN nach der Convolution die Shape `[10, 10, 256]` mit $a \times a = 10 \times 10$.

1. $3 \times 3 \Rightarrow$ Size 4, Stride 3
2. $2 \times 2 \Rightarrow$ Size 5, Stride 5
3. $1 \times 1 \Rightarrow$ Size 10 Stride 10

Auch das ergibt:

$$3 \times 3 \times 256 + 2 \times 2 \times 256 + 1 \times 1 \times 256 = 3584 \tag{2}$$

2 Graph Convolutional Networks

- großes wissenschaftliches Problem, willkürliche Graphen in ein neuronales Netz zu füttern
- Bereich bisher dominiert von kernelbasierten Methoden, graphbasierter Regularisierung oder ähnlichem
- Es gibt eine Ansätze, der neuste ist ein *spektraler*
- spektraler Ansatz gilt als langsam
- das ganze zählt zu dem Bereich des *Semi-unsupervised Learning*, das bedeutet, dass wir einen Teil des Graphen gelabelt haben und basierend auf seiner Graphstruktur, den Rest labeln wollen.

2.1 Definitionen

- Wir wollen eine Funktion von Merkmalen auf einem Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ lernen
- **Eingaben:**
  - Eine Merkmalsbeschreibung x_i für jeden Knoten i dargestellt als eine Merkmalsmatrix $X = N \times D$, wobei N Anzahl der Knoten und D Anzahl der Merkmale
  - Eine repräsentative Beschreibung einer Graphstruktur in Matrixform, normalerweise eine Adjazenzmatrix A
- **Ausgabe**:
  - eine Merkmalsbeschreibung $Z = N \times F$ für jeden Knoten, wobei F Anzahl der Ausgabefeatures für einen Knoten

Jede Schicht des neuronalen Netzes kann über eine nicht-lineare Funktion

$$H^{(l+1)} = f(H^{(l)}, A) \tag{3}$$

beschrieben werden, wobei $H^{(0)} = X$ und $H^{(L)} = Z$ mit L Anzahl der Schichten.
Die Propagationsregel ist dann zum Beispiel:

$$f(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)}) \tag{4}$$

wobei, σ eine nicht lineare Aktivierungsfunktion ist (z.B. `ReLU`) und $W^{(l)}$ eine Gewichtsmatrix für den $l$ten Layer.

Sagen wir A ist eine $n \times n$ Matrix, die über mehrere Graphen hinweg variabel ist. Graph 1 hat zum Beispiel $n = 100$ Knoten, wohingegen Graph 2 nur $n = 80$. $H^{(l)}$ ist eine $n \times f_j$. Die Anzahl an Features pro Layer sind unabhängig vom Graphen und immer gleich. Dann ist $AH^{(l)}$ eine $n \times f_j$ Matrix. $W^{(l)}$ ist eine $f_j \times f^{j+1}$ Matrix. Damit ist die Gewichtsmatrix unabhängig der Knotenanzahl und kann auch für verschiedene große Graphen genutzt werden. Die Gleichheit von $W^{(l)}$ liefert uns den Convolution Gedanken (Shared Weights). Dann ist $H^{(l+1)}$ eine $n \times f_{j+1}$ Matrix.

Am Ende der Convolution hin zum Fully-Connected muss dann $n \times f_j$ gepoolt werden, damit es immer gleich viele Eingabeneuronen gibt (siehe SPP).

A muss jedoch leicht modifziert werden, denn eine Multiplikation mit A summiert alle Feature Vectors der lokalen Nachbarschaftsknoten auf, jedoch ohne den betrachteten Knoten (falls Knoten keine Kante zu sich selbst). Wir können das fixen, in dem wir für jeden Knoten eine Kante sich zu selbst hinzufügen in dem wir A mit der Identitätsmatrix addieren.

Desweiteren ist A nicht normalisiert, das bedeutet, dass eine Multiplikation mit A die Featurevectors komplett anders skaliert. A kann zum Beispiel normalisiert werden, in dem alle Reihen zu Eins aufsummiert werden.

$$D_{ii} = \sum_j A_{ij} \tag{5}$$

beziehungsweise

$$D_{ii}^{-1} = \frac{1}{\sum_j A_{ij}} \tag{6}$$

Matrixnormalisierung dann zum Beispiel $D^{-1}A$ oder $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.

Der erste Layer würde dann die Features eines Knotens mit denen der direkten Nachbarschaft kombinieren. Ein weiterer Layer würde, diess für alle Wege mit Länge 2 tun und so weiter.

Dies entspricht in etwa einer generaliserten Version des Weisfeiler-Lehman Algorithmus auf Graphen:

Für alle Knoten $v_i \in \mathcal{G}$:

1. Sammle Merkmale $\{h_{v_j}\}$ für alle Nachbarschaftsknoten v_j
2. Update Knotenmerkmal $h_{v_i} \leftarrow \text{hash}(\sum_j h_{v_j})$ Wiederhole k-mal oder bis Konvergenz.

Jedem Knoten wird also ein Merkmal zugeordnet, dass seine Rolle im Graphen beschreibt. Dies funktioniert jedoch nicht gut für z.B. reguläre Graphen, bei dem jeder Knoten gleich viele Kanten besitzt. Der Weisfeiler-Lehman Algorithmus wird oft benutzt, um Graphisomorphismen zu bestimmen.

2.2 Klassifierung eines Graphen

Wie kann dieses Model genutzt werden, um einen Graphen zu klassifizieren? Wir erhalten $N \times F$ Features für einen Graphen nach x vielen Convolutions. F kann natürlich dann auch die Anzahl der Klassen sein.

Weiteres Vorgehen: Wir wollen nicht einzelne Knoten labeln, sondern das gesamte Bild bzw. den gesamten Graphen. Diese Features beschreiben den Knoten sowie seine direkte und indirekte Nachbarschaft. Die Knoten müssten jetzt eigentlich noch geeignet gelabelt werden und können dann in ein Netz gefüttert werden.

Wir können Spatial Informationen ausnutzen, wir wissen wie der Graph in der Lage aussieht. Das sollte man sich ruhig zu nutze machen.

3 Graph Pooling / Graph Coarsening / Graph Clustering

Pooling Operationen erfordern es, Graphen basierend auf lokalen Nachbarschaften zu clustern. Wenn dies pro Layer geschieht, dann entspricht dies einem Multi-Scale Clustering auf Graphen, der lokale geometrische Strukturen wahrt. Graph Clustering ist jedoch NP-schwer, d.h. das insbesondere Approximationen benötigt werden, um effiziente Pooling Layer zu generieren. Ein Graph Clustering Algorithmus, der die Knotenanzahl pro Layer um einen Faktor von 2 reduziert, scheint für die Anwendung in einem CNN ideal, wie z.B. die *Coarsening Phase* des *Graclus Multilevel Clustering*-Algorithmus.

Der Algorithmus kriegt als Eingabe einen Graphen $\mathcal{G}_0 = (V_0, E_0, A_0)$ und die Anzahl an gewünschten Partitionen. Die Coarsening Phase des Algorithmus reduziert den initialen Graphen $\mathcal{G}_0$ in immer kleinere Graphen $\mathcal{G}_1$, $\mathcal{G}_2$, ..., $\mathcal{G}_m$, sodass $|V_0| > |V_1| > |V_1| > \ldots > |V_m|$. Um den Graphen zu vergröbern, werden Knoten aus G_i zu einem Superknoten für den Graphen G_{i+1} zusammengefasst. Wenn eine Knotenauswahl kombiniert wird, dann werden die Kantengewichte als die Summe der Kantengewichte der originalen Knoten dargestellt. Die Knotenmerkmale werden über Max Pooling berechnet.

Die Frage, die noch offen ist, ist wie eine Knotenauswahl zu einem Superpixel getroffen wird. Der Ansatz, der hier verfolgt wird, heißt *Heavy Edge Coarsening*. Er funktioniert wie folgt:

1. Sei ein Graph G gegeben, dann setze alle seine Knoten auf den initialen unmarkierten Status.
2. Durchlaufe alle Knoten in einer zufälligen Reihenfolge. Für jeden Knoten x:
   a) `continue`, falls x bereits markiert
   b) Verbinde x mit dem unmarkierten Knoten y, sodass y ein Nachbar von x ist und das Kriterium zwischen x und y maximal für alle unmarkierten Nachbarn.
   c) Markiere x und y.
   d) Falls es keinen Knoten y gibt, d.h. alle Nachbarn von x sind bereits markiert, markiere x als alleinstehenden Knoten.

Als Verbindungskriterium sind einige Freiheitsgrade gegeben:

- das höchste Kantengewicht (vgl. Metis)
- **Max-cut coarsening**: maximiere $\frac{e(x,y)}{w(x)} + \frac{e(x,y)}{w(y)}$, wobei $w(x), w(y)$ die Gewichte der Knoten sind (z.B. Grad des Knoten (**Normalized Cut**))

Dieser Algorithmus reduziert die Knotenanzahl approximiert auf 2. Es kann natürlich ein paar nicht verbundene Knoten geben, sodass die Knotenanzahl ein bisschen weniger reduziert wird.

4 Graph Labeling

Jeder Ansatz von Graph hat zur Folge, dass wir irgendeine *eindeutige* Sortierung von Knoten auf dem Graphen brauchen. Wenn wir das nicht haben, dann haben zwei absolut gleiche Graphen mit unterschiedlicher Sortierung einen unterschiedlichen Output im Netz. Das geht natürlich nicht.

Deswegen gibt es den *Isomorphismus* und *Automorphismus*.

In der Graphentheorie ist ein Isomorphismus zwischen zwei Graphen G und H eine Bijektion zwischen den Knoten der beiden Graphen $f : V(G) \to V(H)$, sodass genau dann zwei Knoten u und v in G adjazent sind, wenn $f(u)$ und (v) adjazent in H sind. Ein Automorphismus ist ein Isomorphismus von G zu sich selbst. Ein Automorphismus ist demnach eine Permutation der Knoten, der die Knoten und Kantenverbindungen aufrecht erhält. Ein Automorphismus gibt demach eine Ordnung der Knoten an, die gleich ist für gleiche Graphen, die unterschiedlich gelabeled sind.

4.1 Canonicalization

Canonical Labeling beschreibt den Prozess des Labelings eines Graphen G, so dass ein Graph der isomorph zu G ist, das gleiche Labeling besitzt.

5 Offene Fragen

Adjazenzmatrix beschreibt den Abstand zu den einzelnen Knoten Dabei sollen diese skalierungsinvariant sein und sich daher im Intervall $[0, 1]$ befinden. Gewichte müssen dabei umgekehrt werden, d.h. eine 1 beschreibt den nächstmöglichsten Abstand zu Knoten v. Als ein

Gewichtsmapping bietet sich dann die Funktion $f : x \to \frac{1}{x}$ an, wobei $\frac{1}{0} = 1$ für Self-Loops. Damit sind wir immer noch skalierungsinvariant (glaube ich). Gewichte 2 und 4 werden zu $0,5$ und $0,25$ gemappt mit Verhältnis 2. Gewichte 4 und 8 werden auf $0,25$ und $0,125$ gemappt mit Verhältnis 2.

Wir definieren unsere Adjazenzmatrix $\tilde{A}$ aus A dann wie folgt:

$$\tilde{A}_{ij} = \begin{cases} 1, & \text{wenn } i = j, \\ 1, & \text{wenn } 0 < a_{ij} \leq 1, \\ \frac{1}{a_{ij}}, & \text{wenn } a_{ij} > 1, \\ 0, & \text{sonst.} \end{cases} \tag{7}$$