Deep Learning und Neuronale Netzwerke

1 Definition Neuronales Netz

Definition 1.0.1. Ein **Neuron** ist eine Funktion $f(x) = \sigma(w^T x + b)$. Hierbei sind:

- $x \in \mathbb{R}^n$ die Inputs (Eingaben),
- $w \in \mathbb{R}^n$ die Gewichte,
- $b \in \mathbb{R}$ der Bias (Verschiebung),
- $\sigma : \mathbb{R} \to \mathbb{R}$ die Aktivierungsfunktion.

Je nach gewünschtem Modell wählt man eine passende Aktivierungsfunktion σ:

- Für lineare Regression: $\sigma(z) = z$.
- Für binäre Klassifikation: $\sigma(z) = \text{sign}(z)$.

Ein Neuronales Netzwerk ist eine Kombination mehrerer solcher Neuronen.

Beispiel 1.0.2. Ein einschichtiges Neuronales Netzwerk ist von der Form

$$f : \mathbb{R}^d \to \mathbb{R}^{d_1}, \quad f(x) = \sigma_1(W_1 x + b_1)$$

mit $x \in \mathbb{R}^d$, $W_1 \in \mathbb{R}^{d_1 \times d}$ und $b_1 \in \mathbb{R}^{d_1}$.

Es nimmt also einen Eingabevektor x, wendet eine lineare Transformation darauf an und anschließend komponentenweise die Aktivierung.

Das Prinzip lässt sich beliebig erweitern. Ein zweischichtiges Neuronales Netzwerk nimmt den Output der ersten Schicht als Input für die nächste Schicht und wendet das gleiche Schema (lineare Transformation & Aktivierung) erneut an.

Dementsprechend können wir das Ganze für L Schichten definieren:

Definition 1.0.3. Ein Neuronales Netzwerk ist ein **Fully Connected Feed Forward Neural Network**, wenn es durch folgende Hyperparameter charakterisiert wird:

- Die Anzahl der Schichten $L \in \mathbb{N}$,
- die Aktivierungsfunktionen $\sigma_l : \mathbb{R} \to \mathbb{R}$ für $l = 1, \dots, L$,
- die Anzahl der Neuronen pro Schicht d_l für $l = 1, \dots, L$.

Der Output ist gegeben durch $f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ mit

$$f(x) = \sigma_L\big(W_L(\dots \sigma_1(W_1 x + b_1) \dots) + b_L\big),$$

wobei $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$, $b_l \in \mathbb{R}^{d_l}$ sind und die σ_l komponentenweise angewandt werden.

Alternativ kann man rekursiv definieren: Sei $\alpha^{(0)}(x) = x$ und für $l = 1, \dots, L$

$$\tilde{\alpha}^{(l)}(x) = W_l \alpha^{(l-1)}(x) + b_l \quad \text{(Pre-Activation der } l\text{-ten Schicht)},$$
$$\alpha^{(l)}(x) = \sigma_l\big(\tilde{\alpha}^{(l)}(x)\big) \quad \text{(Post-Activation der } l\text{-ten Schicht)}.$$

Der Output ist dann $\alpha^{(L)}(x)$.

Diese Definition hat folgende Eigenschaften:

- Wir können sowohl Klassifikations- als auch Regressionsaufgaben bearbeiten; dafür wählen wir lediglich passende Aktivierungsfunktionen.
- Häufig ist σ_L die Identität. Die ersten $L-1$ Layer extrahieren Features, und in der letzten Schicht wird durch W_L entschieden, welche Features wie stark genutzt werden.
- Die Anzahl der Freiheitsgrade (trainierbaren Parameter) ist gegeben durch $\sum_l (d_{l-1} \cdot d_l + d_l)$.

2 Training und Loss-Funktionen

Nun wollen wir klären, wie wir ein Neuronales Netzwerk trainieren. Das geschieht ähnlich wie bei früheren Modellen über das ERM-Prinzip.

2.1 Regressionsprobleme

Eine typische Loss-Funktion ist hier der mittlere quadratische Fehler (MSE). Für ein Neuronales Netzwerk f_w erhalten wir:

$$C(w) = L(f_w) = \frac{1}{2m} \sum_{i=1}^{m} \|f_w(x_i) - y_i\|_2^2.$$

Wir sehen: Sobald mindestens eine Aktivierungsfunktion nicht linear ist, ist $w \mapsto f_w$ nicht linear. Deshalb ist die Kostenfunktion, selbst wenn die Loss-Funktion konvex ist, nicht notwendigerweise konvex. Damit gibt es keine Garantie, dass Gradient Descent stets ein globales Minimum findet. In der Praxis reichen jedoch oft lokale Minima, um ein zufriedenstellendes Ergebnis zu erzielen.

2.2 Klassifikationsprobleme

Da wir Gradient Descent nicht auf nicht-differenzierbaren Funktionen (wie dem 0-1-Loss $l(y, y') = \mathbf{1}_{y \neq y'}$) anwenden können, betrachten wir eine Ersatz-Loss-Funktion.

Beispiel 2.2.1. Betrachte ein Neuronales Netzwerk $f_w(x) = W_L a^{(L-1)}(x) + b_L$. Um darauf binäre Klassifikation durchzuführen, können wir den Predictor $\operatorname{sign}(f_w(x))$ wählen und als Ersatz-Loss

$$l(f_w(x), y) = \log(1 + e^{-y f_w(x)}).$$

Betrachten wir den Fall $y = 1$ und $f_w(x) \geq 0$ (oder $y = -1$ und $f_w(x) < 0$), so sehen wir, dass der Loss ≈ 0 ist. Andernfalls gilt $l(f_w(x), y) \geq \log 2$.

Beispiel 2.2.2. Ein weiterer häufiger Weg für binäre Klassifikation ist, $d_{L-1} = 1$ und $\sigma_L(z) = \frac{1}{1+e^{-z}}$ zu wählen. Das Neuronales Netzwerk liefert dann eine Zahl in $[0, 1]$, die als Wahrscheinlichkeit interpretiert werden kann, dass das wahre Label 1 ist. Unser Predictor soll Label 0 vorhersagen, falls $f_w(x) \leq 0.5$, und 1 andernfalls. Als Loss wählen wir den Cross-Entropy-Loss:

$$L(f_w) = \frac{1}{m} \sum_{i=1}^{m} \left(y_i \log(f_w(x_i)) + (1 - y_i) \log(1 - f_w(x_i)) \right).$$

Für $y = 0$ geht der Loss gegen 0, je näher $f_w(x)$ an 0 ist (und ebenso für $y = 1$ und $f_w(x)$ nahe 1). Definieren wir $0 \log(0) = 0$, so sehen wir, dass ein perfektes Neuronales Netz einen Loss von 0 erreicht.

Beispiel 2.2.3. Wenn wir uns nicht nur für zwei Klassen interessieren, sondern eine Klassifikation mit n Klassen durchführen wollen, können wir $d_L = n$ und $\sigma_L(z) = \text{softmax}(z)$ wählen. Unser Neuronales Netzwerk ist dann von der Form

$$f_w(x) = \frac{1}{\sum_{k=1}^{n} \exp(a^{(L-1)}(x)_k)} \begin{pmatrix} \exp(a^{(L-1)}(x)_1) \\ \vdots \\ \exp(a^{(L-1)}(x)_n) \end{pmatrix}.$$

Jede Zeile j dieses Vektors steht für die Wahrscheinlichkeit, dass x das Label j hat. Während des Vortrags wurde gefragt, weshalb wir $\exp(\cdot)$ als Aktivierung wählen. Der Grund ist, dass wir positive Werte erhalten wollen. Als Loss wählen wir den Categorical Cross-Entropy-Loss:

$$L(f_w) = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} \left(y_{i,j} \log(f_w(x_i)_j) + (1 - y_{i,j}) \log(1 - f_w(x_i)_j) \right),$$

wobei $y_i = (y_{i,1}, \ldots, y_{i,n})$ nur Nullen enthält bis auf eine 1 in der Zeile, die der zu x_i zugehörigen Klasse entspricht. Erneut erreicht ein perfektes Neuronales Netzwerk einen Loss von 0.

3 Backpropagation

Nachdem wir die Kostenfunktionen definiert haben, wollen wir die Parameter W, b so anpassen, dass der Fehler minimal wird. Hierfür nutzen wir Gradient Descent. Das Problem dabei: Ein Neuronales Netzwerk hat oft Tausende Parameter. Würden wir für jeden Parameter einzeln die Steigung berechnen, wäre das rechenaufwendig. Die Lösung ist der **Backpropagation-Algorithmus**. Die Idee ist, sich von hinten nach vorne zu arbeiten und die gewonnenen Informationen abzuspeichern. Aufgrund der Kettenregel benötigen wir für die Gradienten der vorderen Schichten auch die Gradienten der späteren Schichten, die wir dann bereits berechnet haben. Das illustrieren wir an einem Beispiel.

Beispiel 3.0.1. Sei $f_w : \mathbb{R}^2 \to \mathbb{R}$ ein zweischichtiges Neuronales Netzwerk der Form

$$f_w(x) = \sigma_1\big(W_1 \sigma_0(Wx + b_0) + b_1\big)$$

mit $W \in \mathbb{R}^{2\times 2}$, $W_1 \in \mathbb{R}^{1\times 2}$ und entsprechenden Bias-Vektoren. Wir wählen eine quadratische Loss-Funktion, weshalb die Kostenfunktion die Form hat

$$C(w) = \sum_{i=1}^{m} (y_i - f_w(x_i))^2 = \sum_{i=1}^{m} \big(y_i - \sigma_1(W_1 \sigma_0(Wx_i + b_0) + b_1)\big)^2.$$

Um die Gewichte w_{ij}^1 zu finden, müssen wir $\frac{\partial C(w)}{\partial w_{ij}^1}$ berechnen. Dies erhalten wir mit der Kettenregel:

$$\frac{\partial C}{\partial w_{ij}^1} = \frac{\partial z_i^1}{\partial w_{ij}^1} \frac{\partial \sigma_1}{\partial z_i^1} \frac{\partial C}{\partial \sigma_1},$$

wobei $z^1 = W_1 \sigma_0(Wx + b_0) + b_1$. Um nun w_{ij} zu finden, berechnen wir

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial z_i}{\partial w_{ij}} \frac{\partial \sigma_0}{\partial z_i} \frac{\partial z^1}{\partial \sigma_0} \frac{\partial \sigma_1}{\partial z^1} \frac{\partial C}{\partial \sigma_1}$$

für $z = Wx + b_0$. Man sieht dabei gut, dass die hinteren Faktoren bereits berechnet wurden.

3.1 Herleitung Backpropagation

Der Übersicht halber fassen wir Gewichte und Bias in einer erweiterten Gewichtsmatrix $W_n = (W_n \mid b_n)$ zusammen. Wählen wir zudem den erweiterten Vektor

$$\tilde{X}_{n-1} = \begin{pmatrix} X_{n-1} \\ 1 \end{pmatrix},$$

so können wir unser Neuronales Netzwerk rekursiv darstellen als

$$Z_n = W_n \tilde{X}_{n-1}, \qquad X_n = \sigma_n(Z_n).$$

Nun nehmen wir an, $\frac{\partial C}{\partial X_{n,i}}$ für $i = 1, \dots, d_n$ sei bereits berechnet. Dann lassen sich die folgenden Gradienten rekursiv berechnen:

$$\begin{aligned} \frac{\partial C}{\partial z_{n,i}} &= \sigma_n'(z_{n,i}) \frac{\partial C}{\partial X_{n,i}}, \\ \frac{\partial C}{\partial w_{ij}^n} &= x_{n-1,j} \frac{\partial C}{\partial z_{n,i}}, \\ \frac{\partial C}{\partial X_{n-1,j}} &= \sum_k w_{kj}^n \frac{\partial C}{\partial z_{n,k}}, \end{aligned}$$

mit $z_{n,i} = \sum_j w_{ij}^n X_{n-1,j}$.

Schreiben wir die Gradienten als Vektoren bzw. für die Gewichte als Matrix, so erhalten wir

$$\frac{\partial C}{\partial Z_n} = \sigma_n'(Z_n) \circ \frac{\partial C}{\partial X_n}, \tag{1}$$

$$\frac{\partial C}{\partial W_n} = \left(\frac{\partial C}{\partial Z_n} \right) (X_{n-1})^T, \tag{2}$$

$$\frac{\partial C}{\partial X_{n-1}} = W_n^T \frac{\partial C}{\partial Z_n}, \tag{3}$$

wobei $\circ$ dem Hadamard-Produkt, also der komponentenweisen Multiplikation, entspricht.

Diese drei Gleichungen nutzen wir nun, um $\frac{\partial C}{\partial W_n}$ umzuschreiben:

$$\begin{aligned} \frac{\partial C}{\partial W_n} &= \left(\frac{\partial C}{\partial Z_n} \right) (X_{n-1})^T \\ &= \left(\sigma_n'(Z_n) \circ \frac{\partial C}{\partial X_n} \right) (X_{n-1})^T \\ &= \left(\sigma_n'(Z_n) \circ W_{n+1}^T \frac{\partial C}{\partial Z_{n+1}} \right) (X_{n-1})^T \\ &= \left(\sigma_n'(Z_n) \circ W_{n+1}^T \sigma_{n+1}'(Z_{n+1}) \circ \cdots \circ \sigma_L'(Z_L) \circ \frac{\partial C}{\partial X_L} \right) (X_{n-1})^T. \end{aligned}$$

Wir nennen den Term in der großen Klammer $\delta^n = \frac{\partial C}{\partial Z_n}$. Dann gilt $\frac{\partial C}{\partial W_n} = \delta^n X_{n-1}^T$. Außerdem erhalten wir δ^{n-1} rekursiv durch

$$\delta^{n-1} = \sigma_{n-1}'(Z_{n-1}) \circ W_n^T \delta^n \quad \text{mit} \quad \delta^L = \sigma_L'(Z_L) \circ \frac{\partial C}{\partial X_L}.$$

Anstatt jedes Gewicht komplett neu zu berechnen, nutzen wir die Gradienten der hinteren Schichten weiter und erhalten so die Gradienten aller Parameter in einem einzigen Rückwärtsdurchlauf. Dieses Prinzip macht das Training tiefer Netze überhaupt erst möglich.

4 Vanishing & Exploding Gradients

Wir haben gesehen, wie wichtig die Effizienz von Backpropagation ist. Das Verfahren hat aber auch eine Kehrseite: Bei besonders tiefen Netzen besteht der Gradient aus einer langen Kette von Multiplikationen. Nehmen die Faktoren sehr große bzw. sehr kleine Werte an, führt das exponentiell dazu, dass Gradienten der ersten Schichten explodieren bzw. verschwinden können.

Beispiel 4.0.1. Betrachten wir die Ableitung einer Abbildung $f_L(w)$, die wir rekursiv über Abbildungen g_l für $l = 1, \dots, L$ definieren durch $f_0(w) = w$ und $f_{l+1}(w) = g_{l+1}(f_l(w))$ für $l = 1, \dots, L-1$. Dann folgt

$$f_L'(w) = g_L'(f_{L-1}(w))\, f_{L-1}'(w) = \prod_{i=0}^{L-1} g_{L-i}'(f_{L-i-1}(w)).$$

Angenommen, jede dieser Ableitungen sei $\approx a$. Dann wird $f_L'(w) \approx a^L$ sein.

- Für $a < 1$ erhalten wir einen sehr kleinen Gradienten (Training bleibt stecken).
- Für $a > 1$ wächst er exponentiell (Minima werden übersprungen).

Betrachten wir δ^n, so sehen wir, dass viele Faktoren Ableitungen von Aktivierungsfunktionen sind. Das Problem ist daher eng mit der Wahl der Aktivierungen verbunden.

Beispiel 4.0.2.

- Die **Sigmoid-Funktion** $\sigma(z) = \frac{1}{1+e^{-z}}$ hat eine Ableitung, die gegen 0 geht, sobald man sich etwas vom Ursprung entfernt (*Vanishing Gradients*).
- Eine **Polynom-Funktion** vom Grad ≥ 2 hingegen hat eine Ableitung, die für große Werte explodiert (*Exploding Gradients*).

5 Initialisierungen

5.1 Problem naiver Initialisierungen

Wir haben nun eine Idee, wie ein Neuronales Netzwerk aussieht und wie wir es effizient trainieren können. Um mit dem Training zu beginnen, müssen wir anfangs unsere Parameter initialisieren. Angenommen, wir wählen naive Ansätze:

Beispiel 5.1.1. Ein erster Ansatz ist, alle Gewichte auf 0 zu setzen. Da einige Faktoren von δ^n jedoch Gewichtsmatrizen sind, ergibt die Berechnung des Gradienten-Updates

$$\frac{\partial C}{\partial W_n} = \delta^n X_{n-1}^T = 0.$$

Da der Gradient 0 ist, verändern sich die Gewichte während des Trainings nicht. Das Netzwerk lernt also nichts und bleibt im initialen Zustand stecken.

Beispiel 5.1.2. Der nächste Ansatz ist, alle Gewichte auf einen konstanten Wert $c \in \mathbb{R}$ zu setzen. In diesem Fall gilt

$$\frac{\partial C}{\partial W_n} = \delta^n X_{n-1}^T = \delta^n \sigma_{n-1}\big(c\, \mathbb{1}_{d_{n-1} \times d_{n-2}} X_{n-2}\big)^T = \beta \gamma^T = c_n\, \mathbb{1}_{d_n \times d_{n-1}},$$

mit konstanten Vektoren γ und β sowie $c_n \in \mathbb{R}$. Alle Gewichte der n-ten Schicht erhalten also nach einem Trainingsdurchgang das gleiche Update. Alle Neuronen lernen damit exakt das Gleiche, was zur Folge hat, dass das gesamte Netzwerk sich effektiv wie ein Modell mit nur

einem einzigen Neuron pro Schicht verhält. An dieser Stelle wurde im Vortrag gefragt, wie überhaupt die Gradienten genutzt werden, um die alten Gewichte zu verbessern. Die Formel für das Gewichtsupdate ist von der Form

$$w_{k+1} = w_k - \eta \frac{\partial C}{\partial w_k}$$

für $\eta > 0$ (Schrittweite).

Die eben genannten Probleme lassen sich am einfachsten umgehen, indem wir die Gewichte zufällig initialisieren. Die Frage ist nur: *Wie zufällig?*

5.2 Initialisierungsmethoden und deren Herleitung

Um das Problem der explodierenden bzw. verschwindenden Gradienten zu umgehen, versuchen wir, die Varianz unserer Aktivierungen X_n durch alle Schichten hindurch konstant zu halten. Da die Varianz einer nicht-linearen Funktion schwer analytisch zu berechnen ist, nutzen wir eine Taylor-Entwicklung erster Ordnung um 0. Wir nehmen an $\sigma_n(0) = 0$ (also auch $b_n = 0$ für $n = 1, \dots, L$) und approximieren:

$$X_{n,k} = \sigma_n(z_{n,k}) \approx \sigma_n(0) + \sigma_n'(0) z_{n,k} = \sigma_n'(0) \sum_{j=1}^{d_{n-1}} w_{kj}^n X_{n-1,j}.$$

Wir nehmen zusätzlich an, dass Gewichte und Inputs unabhängig, zentriert (Erwartungswert 0) und jeweils identisch verteilt sind. Dann gilt für die Varianz der Aktivierung in Schicht n:

$$\begin{aligned}
\operatorname{Var}(X_{n,k}) &\approx \operatorname{Var}\left(\sigma_n'(0) \sum_{j=1}^{d_{n-1}} w_{kj}^n X_{n-1,j} \right) \\
&= \sigma_n'(0)^2 \sum_{j=1}^{d_{n-1}} \operatorname{Var}(w_{kj}^n X_{n-1,j}) \\
&= \sigma_n'(0)^2 \sum_{j=1}^{d_{n-1}} \operatorname{Var}(w_{kj}^n) \operatorname{Var}(X_{n-1,j}) \\
&= \sigma_n'(0)^2 \cdot d_{n-1} \cdot \operatorname{Var}(w_{kj}^n) \cdot \operatorname{Var}(X_{n-1,j}).
\end{aligned}$$

Wählen wir $\operatorname{Var}(w_{kj}^n) = \frac{1}{\sigma_n'(0)^2 \cdot d_{n-1}}$, so sehen wir, dass die Varianz von X durch alle Schichten ungefähr gleich bleibt ($\operatorname{Var}(X_n) \approx \operatorname{Var}(X_{n-1})$).

Bemerkung 5.2.1. Im Vortrag wurde gefragt, warum wir in der Rechnung $\operatorname{Var}(w_{kj}^n X_{n-1,j}) = \operatorname{Var}(w_{kj}^n) \operatorname{Var}(X_{n-1,j})$ verwenden dürfen, obwohl X_{n-1} durch Vorwärtspropagation aus den Inputs entsteht und damit von Gewichten abhängt. Entscheidend ist: X_{n-1} hängt nur von den Gewichten der vorherigen Schichten $(W_1, \dots, W_{n-1})$ und den Inputs ab, aber nicht von den aktuellen Gewichten W_n der betrachteten Schicht, welche unabhängig von den vorherigen Gewichten gewählt wurden. Daher kann man w_{kj}^n als unabhängig von $X_{n-1,j}$ annehmen.

Diese eben hergeleitete Formel ist die Basis für die folgenden berühmten Initialisierungsstrategien.

Beispiel 5.2.2. Für die Aktivierungsfunktion $\sigma_n(x) = \tanh(x)$ mit $\sigma_n'(0) = 1$ haben sich folgende Strategien durchgesetzt:

- **Xavier:** $w_{ij}^n \sim U\left(-\frac{\sqrt{3}}{\sqrt{d_{n-1}}}, \frac{\sqrt{3}}{\sqrt{d_{n-1}}} \right)$,
- **LeCun:** $w_{ij}^n \sim N\left(0, \frac{1}{d_{n-1}}\right)$.

Beispiel 5.2.3. Abschließend wollen wir eine geeignete Initialisierung für die Aktivierung $\sigma_n(z) = \text{relu}(z) = \max(0, z)$ finden. Das Problem hierbei: ReLU ist im Nullpunkt nicht differenzierbar. Wir können die obige Herleitung daher nicht direkt nutzen und müssen die Varianz anders herleiten. Diesmal betrachten wir die Pre-Activation $z_n = W_n X_{n-1} + b$ und verfolgen das Ziel $\text{Var}(z_n) \approx \text{Var}(z_{n-1})$, was einen ähnlichen Effekt hat wie der Ansatz zuvor. Erneut müssen wir einige Annahmen treffen.

Annahmen:

- w_{ij}^n unabhängig, identisch verteilt und zentriert,
- X_{n-1} unabhängig und identisch verteilt,
- Bias $b = 0$,
- Gewichte und Inputs sind unabhängig voneinander.

Herleitung:

$$\begin{aligned}\text{Var}(z_{n,i}) &= \text{Var}\left(\sum_{j=1}^{d_{n-1}} w_{ij}^n X_{n-1,j}\right) = \sum_{j=1}^{d_{n-1}} \text{Var}(w_{ij}^n X_{n-1,j}) \\ &= \sum_j \left(\text{Var}(w_{ij}^n)\,\text{Var}(X_{n-1,j}) + (E[w_{ij}^n])^2\,\text{Var}(X_{n-1,j}) + \text{Var}(w_{ij}^n)(E[X_{n-1,j}])^2\right).\end{aligned}$$

Da $E[w_{ij}^n] = 0$, fällt der mittlere Term weg. Nun klammern wir $\text{Var}(w_{ij}^n)$ aus:

$$\text{Var}(z_{n,i}) = d_{n-1}\,\text{Var}(w_{ij}^n)\underbrace{\left(\text{Var}(X_{n-1,j}) + (E[X_{n-1,j}])^2\right)}_{E[(X_{n-1,j})^2]}.$$

Nun berechnen wir $E[(X_{n-1,j})^2]$. Zur Erinnerung: X_{n-1} ist der Output einer ReLU-Schicht ($X_n = \max(0, z_n)$), und z_n ist symmetrisch um 0 verteilt, da W_n es ist.

$$E[(X_{n-1,i})^2] = \int_{-\infty}^{\infty} (X_{n-1,i})^2 p(X_{n-1,i})\,dx = \int_0^{\infty} z_{n-1,i}^2 p(z_{n-1,i})\,dz.$$

Da die Wahrscheinlichkeitsdichte symmetrisch ist, entspricht das Integral von 0 bis ∞ genau der Hälfte der Varianz der vorherigen Pre-Activation:

$$E[(X_{n-1,i})^2] = \frac{1}{2}E[z_{n-1,i}^2] = \frac{1}{2}\,\text{Var}(z_{n-1,i}) = \frac{1}{2}\,\text{Var}(z_{n-1,j}).$$

Setzen wir dies oben ein, erhalten wir:

$$\text{Var}(z_{n,i}) = d_{n-1} \cdot \text{Var}(w_{ij}^n) \cdot \frac{1}{2}\,\text{Var}(z_{n-1,i}).$$

Damit die Varianz konstant bleibt ($\text{Var}(z_n) \approx \text{Var}(z_{n-1})$), muss gelten:

$$\text{Var}(w_{ij}^n) = \frac{2}{d_{n-1}}.$$

Daraus folgt die Initialisierung:

- **Kaiming He:** $w_{ij}^n \sim N\left(0, \frac{2}{d_{n-1}}\right)$.

Abschließend sei erwähnt, dass wir hier nur versucht haben, die Varianz der Aktivierungen im Vorwärtslauf zu kontrollieren; ein verwandter Ansatz (Glorot Initialisierung) zielt zusätzlich darauf ab, auch den Rückwärtslauf der Backpropagation zu stabilisieren, indem die Varianz der Gewichte typischerweise proportional zu $\frac{1}{d_{n-1}+d_n}$ skaliert wird.