Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

Tag 7

Vertiefung Regularisierung
Tiefe neuronale Netze,
Vanishing & Exploding Gradients
Skip-Verbindungen,
Batch-Normalization

Frühere Probleme

Frühere Probleme tiefer neuronaler Netze.
Warum tiefe neuronale Netze erst ab 2010 populär wurden:

`Vanishing Gradients`
`Exploding Gradients`

Vanishing Gradients

Verschwindende Ableitungen

Problem: Während des Backpropagation-Prozesses entstanden durch wiederholtes Multiplizieren kleiner Ableitungen exponentiell abnehmende Werte. Der Gradient wird immer kleiner, je tiefer man (von oben, vom output/Fehler her) **im Netzwerk zurück geht. Dies führte dazu, dass die Gewichte in den frühen Schichten kaum noch angepasst werden und das Netzwerk daher nicht gut lernt.**

- Gradienten werden in tiefen Schichten sehr klein
- Frühere Schichten lernen kaum
- **Lösungen**:
  - **ReLU** Aktivierungsfunktion (keine Sättigung im positiven Bereich, Ableitung **0**/1!)
  - **Weight-Initialization** (Xavier **/ He 0 nicht gut / random() automatisch in pytorch**)
  - **Spectral Initialization/Normalization** (Unitarisierte Matrizen Spektralnorm nahe 1)

  - **Skip**-Verbindungen (**Residual** Architektur)
  - **Layer Normalization**
  - **Batch Normalization**
  - **Gated Architectures:** LSTMs, GRUs im RNN "Gedächtnis" (nächste Woche)

Exploding Gradients

Problem: Im Gegensatz zu den vanishing gradients werden bei explodierenden Gradienten die Gradienten in tiefen Netzwerken extrem groß. Wiederholtes **Multiplizieren von Werten größer als eins** ergibt exponentiell wachsende Werte. Dies führt zu Instabilitäten im Lernprozess und kann dazu führen, dass die Gewichte extrem hohe Werte annehmen oder das Training komplett fehlschlägt (**Inf / NaN**).

- Gradienten werden extrem groß
- Führt zu Instabilitäten
- Lösungen:
  - Gradienten-**Clipping** "zu groß => einfach abschneiden;)"
  - **generelle Regularisierung-Techniken**

Regularisierungen II

Verschwindende Ableitungen

- Gradienten werden in tiefen Schichten sehr klein
- Frühere Schichten lernen kaum

Explodierende Ableitungen

- Gradienten werden in tiefen Schichten sehr groß
- Netz wird unstabil

- **Lösungen**:
- Siehe Tag 4
  - Leaky **ReLU** Aktivierungsfunktion (keine Sättigung im positiven Bereich wie bei Δsigmoid)
    `torch.nn.LeakyReLU()  IMMER verwenden statt ReLU!`
  - **Weight-Initialization** (Xavier **/ Kaiming-He)**
  - **Skip-Verbindungen**
  - **L1 / L2 loss**
  - **Dropout**
  - **Batch Normalization**

Initialisierungen

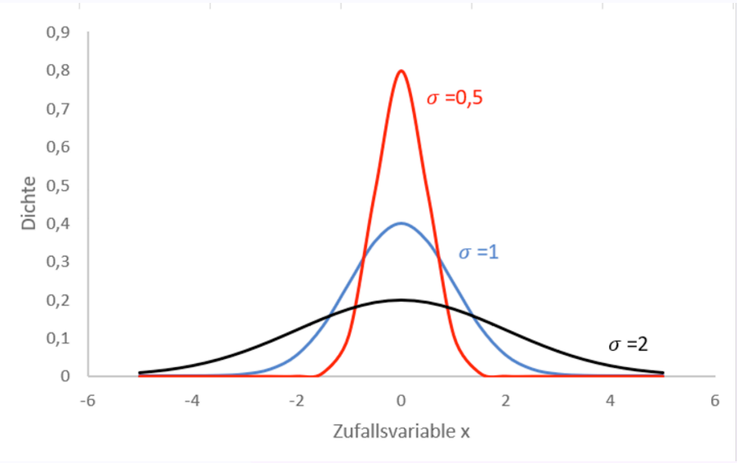
Regularisierung durch **Initialisierung der Gewichte**

Idee:

N = Anzahl der Gewichte Gleichverteilt / N => ∑ w = 1

Gewichte werden zufällig aus einer Normalverteilung um 0 initialisiert.

- 💡 Je **breiter** der **Input** umso **kleiner** werden die **Gewichte**.
- 💡 **Effekt:** Gewichte bleiben nahe 1 und **'schaukeln sich nicht auf'**



Initialisierungen

Manuelle **Initialisierung der Gewichte**

```
# Initialisierungsfunktion
def weights_init(layer):
  if isinstance(layer, nn.Linear):
    nn.init.xavier_uniform_(layer.weight) # alte Initialisierung

# Initialisierung der Gewichte
model.apply(weights_init)
```

Initialisierungen

Regularisierung durch **Initialisierung der Gewichte**

Idee:

Veraltet: Die **Xavier/Glorot**-Initialisierung sorgt dafür, dass die **Varianz** der Ausgabe einer Schicht **weder zu groß noch zu klein** ist. Dies erreicht man, indem die **Gewichte so skaliert** werden, dass die Varianz der Ausgabewerte gleich der Varianz der Eingabewerte ist.

$$W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}\right)$$

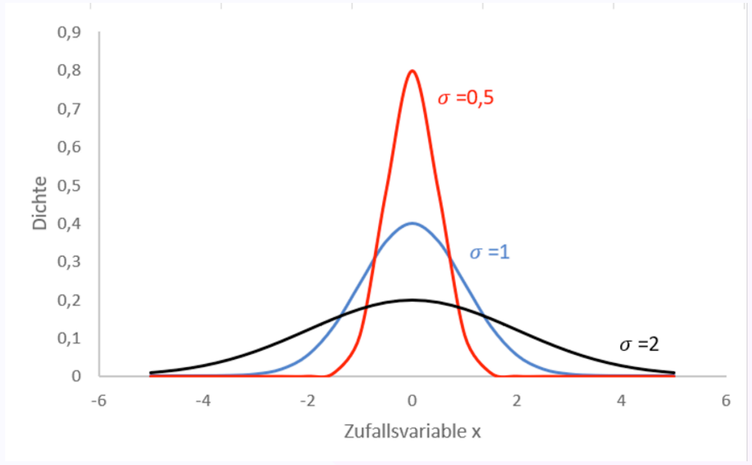
Gewichte werden zufällig aus einer Gleichverteilung **um 0** Intervall initialisiert

Kaiming-He Initialisierung
Inplizierter Standard in pytorch
Für Conv2d und ReLu Aktivierungen optimiert

bias = 0 $W \sim \mathcal{N}(0, \sqrt{\frac{2}{n_{\text{in}}}})$

Gewichte werden zufällig aus einer Normalverteilung um 0 initialisiert.

- 💡 Je **breiter** der **Input** umso **kleiner** werden die **Gewichte**.
- 💡 **Effekt:** Gewichte bleiben nahe 1 und '**schaukeln sich nicht auf**'



Exploding Gradients

- **Gradienten-Clipping**

Ad Hoc Ansatz: Wenn Gradient zu groß wird, schneide ihn einfach ab ;)

"grad = max(grad,1)"

...

```
loss.backward()  # Backward Pass

# Gradient Clipping
nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

# Clipping auf speziellen loss term, z.B.
torch.clip(loss_1,0.0,0.999)
loss = loss + l1_loss

optimizer.step()  # Optimierungsschritt
…
```

Aufgabe

Beispiel vanishing gradient:
mnist_vanishing_gradient.py

vereinfachen

Aufgabe

Eine der Initialisierungen fürs eigene Perceptron übernehmen.

Idealerweise vergleichen ob das die Schrittanzahl reduziert.

Skip-Verbindungen

Warum? Zusätzliche Aspekte können mit original Input "vermengt" werden.

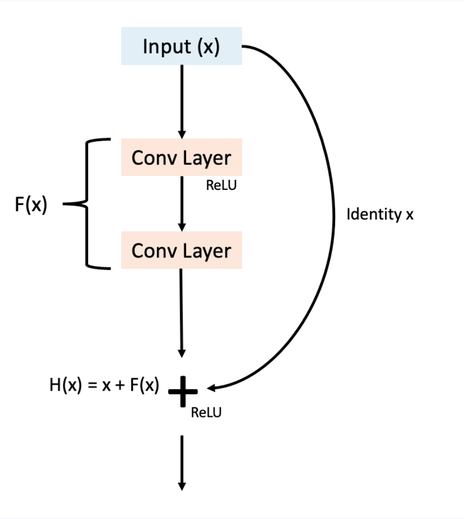
Analogie: Wir erkennen einen Baum aber können trotzdem auf jedes Blatt 'zoomen'

⚠️ Dimension von x und F(x) müssen bei **Residuals** passen!

Wann benutzen?
"Möglichst überall"
ausser Densenets, weil sie Aktivierungen explodieren lassen
bei Resnets **Elementweise addition** kaum overhead.

Wie benutzen?

```
>>> [1,2]+[3,4]
[1, 2, 3, 4]  # Densenet
>>> import numpy as np
>>> np.array([1,2])+np.array([3,4])
array([4, 6]) # Resnet !
```



Skip-Verbindungen

Aufgabe: **Residual Block** sinnvoll in eigenes Netz einbauen

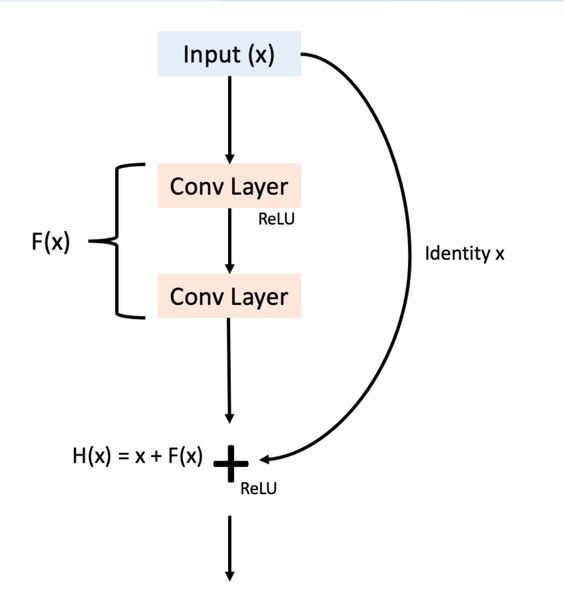
a) erst mal über SubNet bei dem Dimension erhalten bleibt
ConvNet(k=3,p=1,s=1)

b) über SubNet bei dem Dimension reduziert wird (schwer?)

c) Ähnlich aber anders: (Indentity) **Dense Blocks**
(hier wird append'ed/concat'eniert statt addiert)

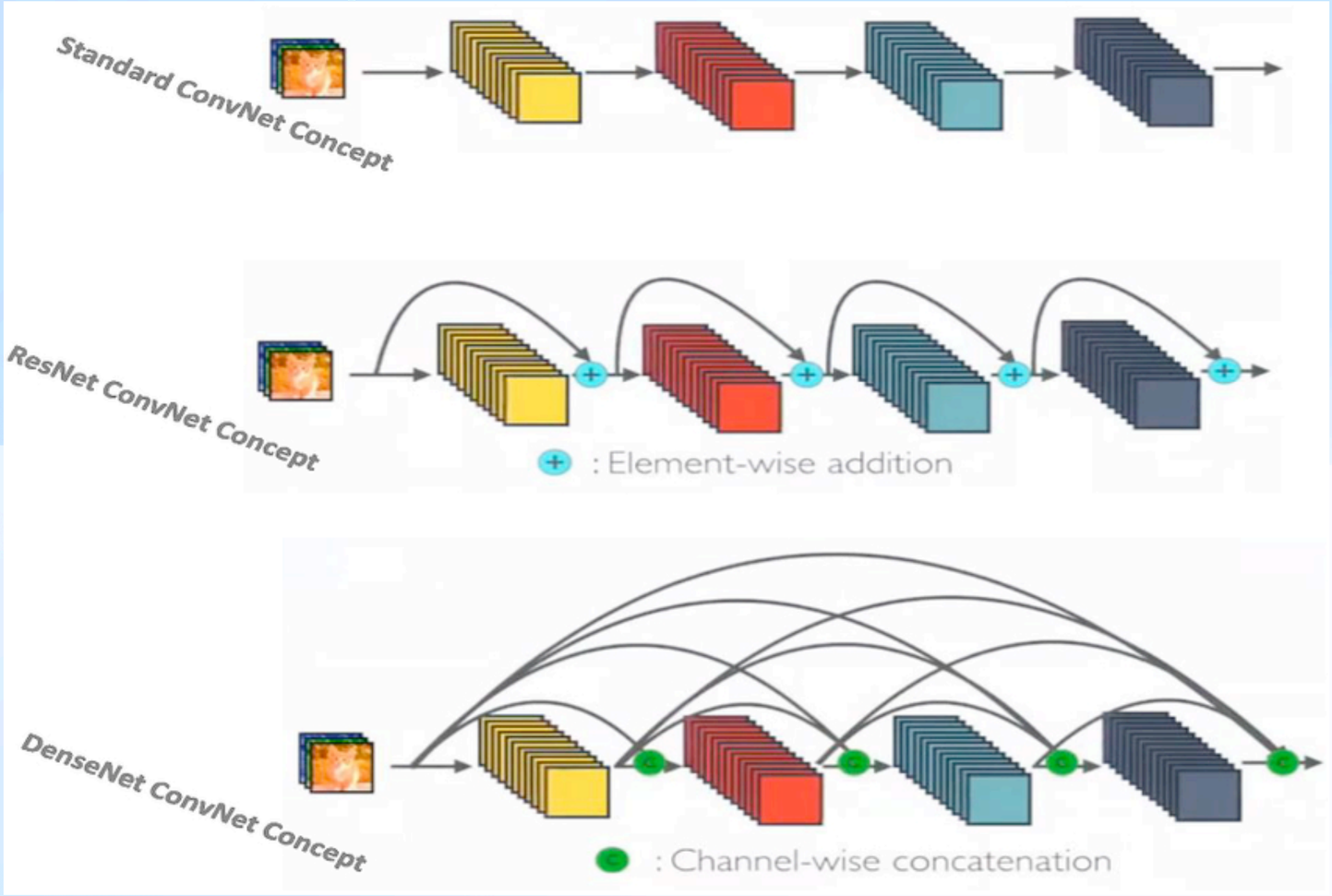
Warum "Concat Blocks" nicht weiter verbreitet?
"Dimensions-Explosion + zu aufwändig" nicht immer
https://paperswithcode.com/method/dense-block

Erkenntnis:
Beides erhöht Lern-Geschwindigkeit signifikant!



Skip-Verbindungen

Skip Verbindungen gehen nicht nur mit Input, sondern für beliebige Aktivierungs-Vektoren, welche als Identität weitergereicht werden.

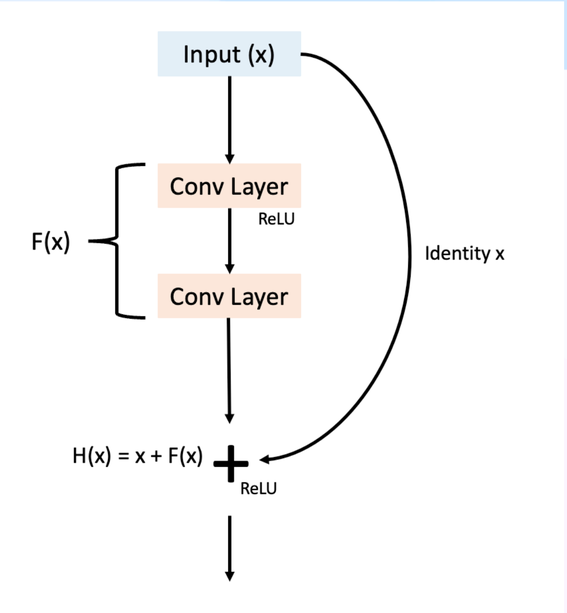


bei DenseNet werden Stapel immer größer! bei ResNet bleiben sie.

Skip-Verbindungen

Idee: wenn wir beobachten dass in Skip Verbindung am Ende Identität überwiegt kann man komplexe Berechnung einfach wegoptimieren!?

Zu überprüfen: ist F(x)=main_branch(x) am Ende ≈ 0? Dann entfernen!



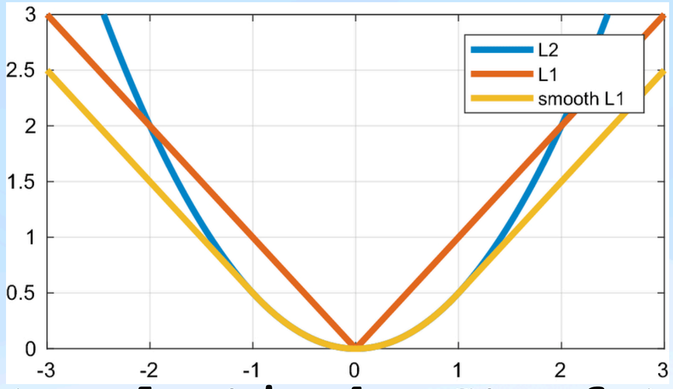
L1 / L2 loss

Wie angedeutet können wir in unsere Verlustfunktion beliebige Custom Terme mitzunehmen um das Verhalten beim lernen fein zu steuern.

Insbesondere können wir
zu große Gewichte bestrafen:

L1 : loss = original loss + λ ∑|wᵢ|

L2 : loss = original loss + λ ∑wᵢ²



(quadratische Strafe)

Das **λ** ist nur ein **Hyperparameter** mit dem man das **Ausmaß der Strafe** angemessen werden kann. Z.B. λ = 0.1*ursprungs_verlust/(Anzahl Gewichte)

💡 Um ein Gefühl für ein gutes Lambda zu bekommen kann man das Netzwerk erst mal durchlaufen lassen und den L1/L2 **λ** dann in etwa der **Größenordnung vom normalen Verlust** skalieren.

L1 / L2 loss

Aufgabe: l2 loss term zum loss eines Netzwerkes hinzu addieren

L2 : loss = original loss + $\lambda \sum w_i^2$

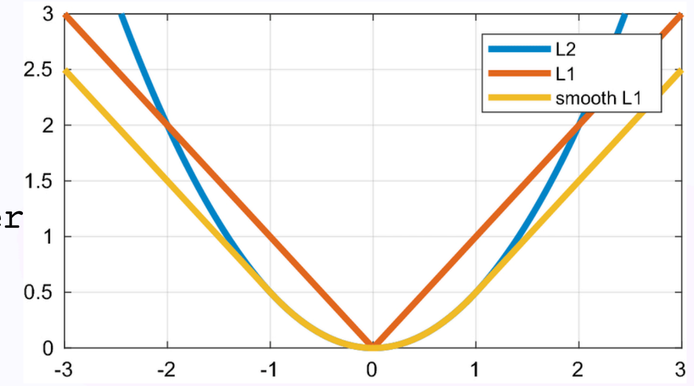
💡 lambda für L2 Regularisierung kann ***kleiner*** angesetzt und muss nicht so genau geschätzt werden, weil Gewichte quadratisch bestraft werden, irgendwann wird der Term schon groß genug, also Gewichte bleiben im überschaubaren Bereich.

z.B. $\lambda < 0.00000001$

```
l1_term = l1_lambda*sum(abs(layer).sum() for layer in model.parameters())  oder
l2_term = l2_lambda*sum((w*w).sum() for w in model.parameters())

cost = cost + l2_term
```

mnist_l1.py



Regularisierung II

A) Im **Model (Architektur)**

- **Dropout:** Zufälliges “Ausschalten” von Neuronen während des Trainings, um das Modell robuster zu machen.
  statt das alle Information in einem Neuron konzentrier werden kann, forciert dieser Ansatz 'holographisches' lernen ✓
- **Skip-Verbindungen**
  Residual Connections: Weiterschleifen von unverfälschten Signalen in tiefere Schichten (Tag 7) ✓
- **Batch Normalization**
- Ausblick: Abziehen von Schwerpunkt und teilen durch Varianz, pro batch(!) auch in tieferen Schichten

B) Im Fehler / In Gewichten

- **Xavier / He Initialisierung** ✓
- **L1- und L2-Regularisierung:** Hinzufügen eines Penalisierungsterms zu der Verlustfunktion, um große Gewichte zu verhindern.
  allgemeiner kann man die Fehlerfunktion mit eigenen Kriterien erweitern (auch als Schicht!) ✓

C) Daten:

- **Datenbereiningung und Vorbereitung (✓)**
- **Synthetische Daten &**
- **Datenaugmentation:** Künstliche Erhöhung der Trainingsdatenmenge durch Transformationen (z.B. Rotationen, Skalierungen).
  sehr anschaulich bei OCR (Texterkennung) : man kann die Texte auf verschiedenste Weisen '**stören**' und das Netz lernen lassen

Regularisierung II

Ad Hoc Techniken auf der Suche nach dem perfekten Rechenschema

Schritt zurück, **warum Regularisierung**?

Vermeidung von Überanpassung (Overfitting)

Vermeidung von Vanishing/Exploding Gradient

Bessere, schnellere Konvergenz

Allgemein: Robusteres Netzwerk, "**Balance**", stabile Numerik "Zahnräder passen gut ineinander"

Regularisierungsmaßnahmen beeinflussen sich gegenseitig. Beispielsweise kann L2-Regularisierung das Überanpassungsproblem mildern, indem es Gewichte verkleinert, was jedoch gleichzeitig das Risiko für das Verschwinden von Gradienten erhöht. Umgekehrt können Methoden zur Bekämpfung des Vanishing-Gradient-Problems, wie etwa die Verwendung von optimierten Aktivierungsfunktionen oder Normalisierungstechniken, indirekt die Effektivität von Regularisierungen wie Dropout oder L2 beeinflussen, da sie die Dynamik der Gewichtsanpassung verändern. Es erfordert eine sorgfältige Balance, um diese Effekte optimal zu steuern.

Schön: Batteries Included in den Bibliotheken (Initialisierung / L2 als Parameter vom Optimizer)

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=0.0001) #L2
```

$$\theta \leftarrow \theta - \eta \cdot (g + \lambda\theta)$$

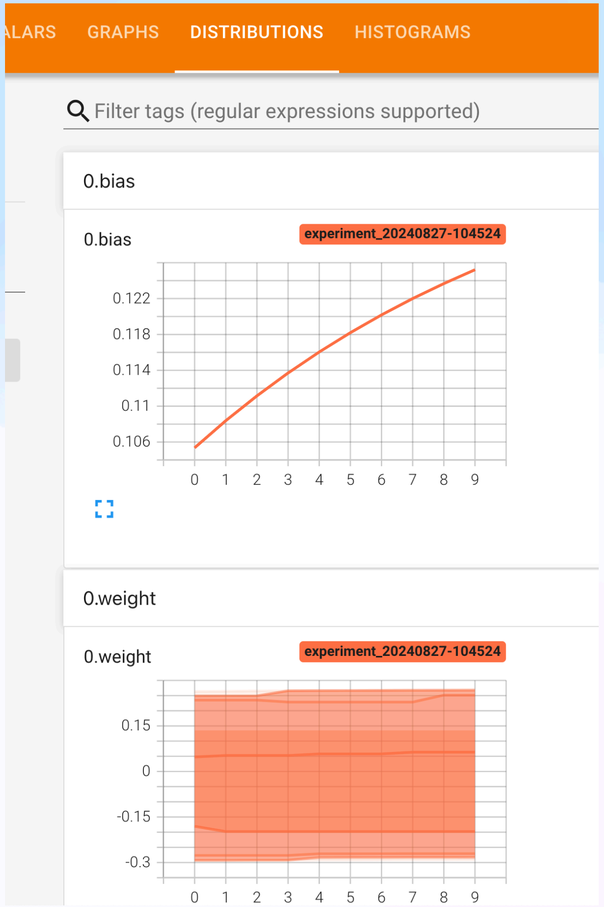
Bei Adam empfiehlt sich oft **AdamW** (torch.optim.AdamW), da dieser die Decay korrekt *entkoppelt* und besser funktioniert als das klassische „L2 im Gradienten“.

Regularisierung II

Nachverfolgung von **aus den Ruder laufende**n **Gewichte**n mittels **tensorboard**:

```
# After each epoch, log weight histograms for each layer
for name, param in model.named_parameters():
    writer.add_histogram(name, param, epoch)
```

dann unter Tab "Distributions"



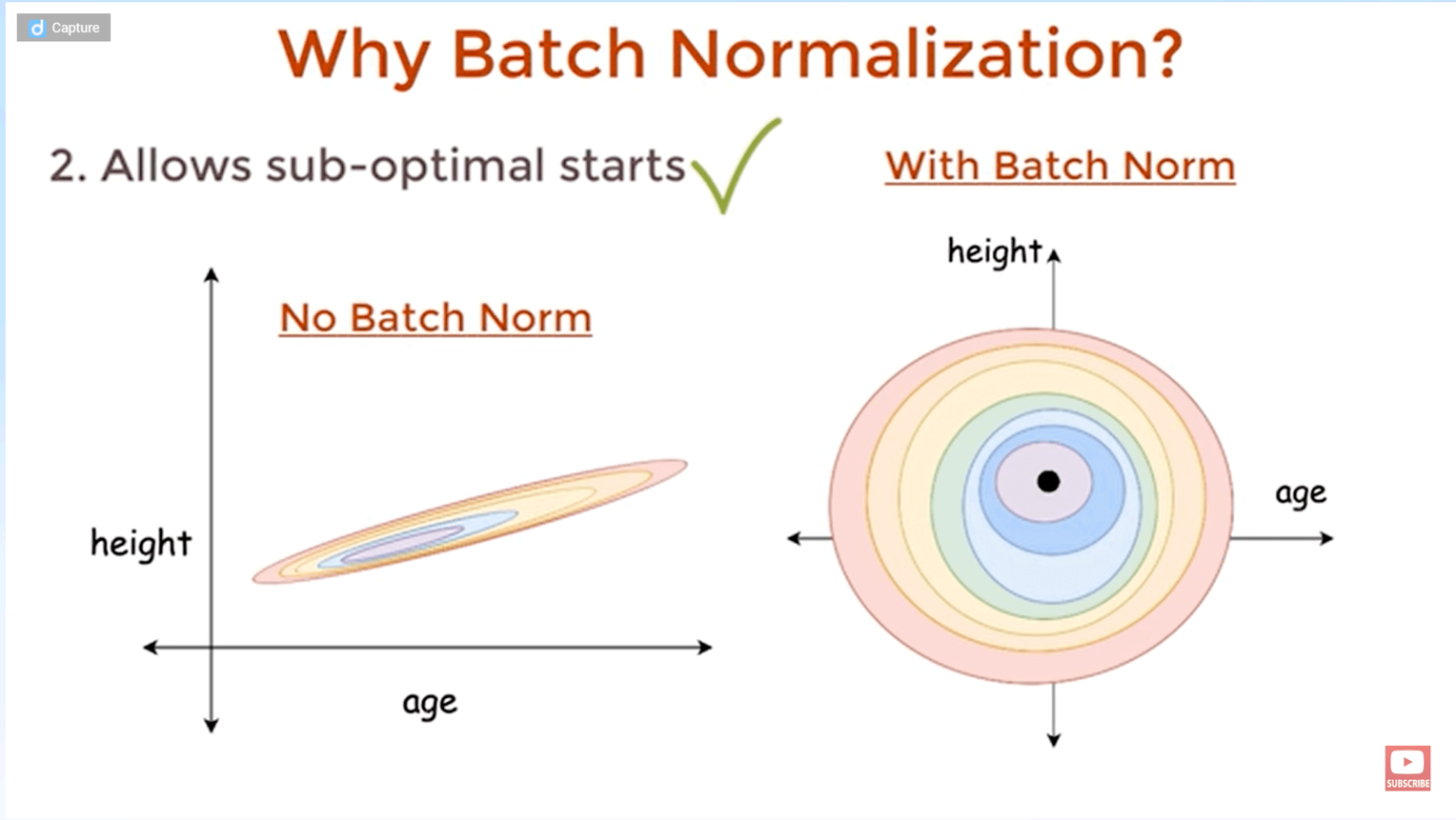
Aufgabe

Visualisiere Gewichte in einem unserer Netzwerke (möglichst tief! LeNet)
Visualisiere Gewichte mit / ohne Regularisierung.

Batch-Normalization

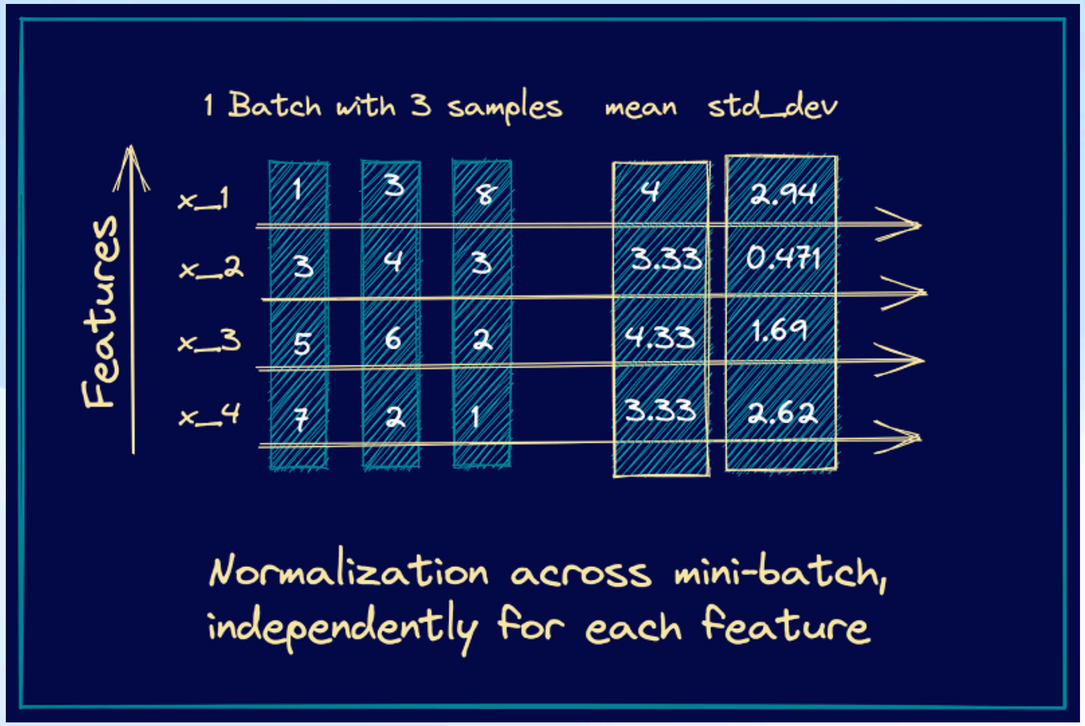
Batch-Normalization

Die Input Daten eines Layers werden **normalisiert** (zentriert und gestaucht), sodass sie einen **Mittelwert von 0 und eine Varianz von 1** haben**.**



Batch-Normalization

Normalisiere die Daten **pro Batch**:
ziehe Mittelwert ab und teile durch Standardabweichung = √**Varianz**

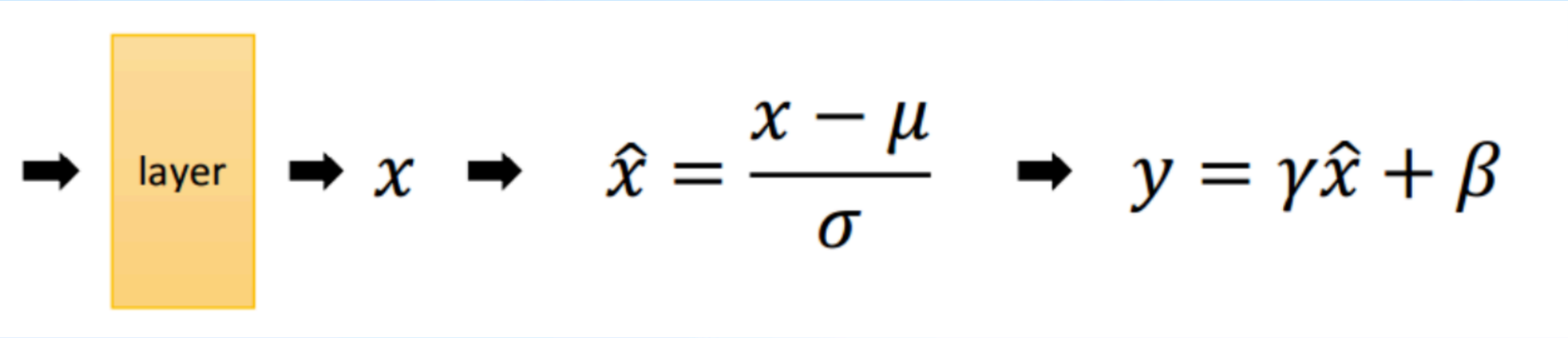


$$x \rightarrow \hat{x} = \frac{x - \mu}{\sigma}$$

μ = Mittelwert
σ = Standardabweichung = √Varianz = √ E(x - μ)

Batch-Normalization

Reskalierung: nach der Normalisierung
können die Daten zusätzlich mit
zwei **lernbaren Parametern** wieder
gestreckt (γ) und verschoben (β) werden



Model kann lernen die Verzerrung rückgängig zu machen
=> Identität wie Residuals

Batch-Normalization

Wenn Daten ohnehin normalisiert kommen,
kann man Batch-Norm weglassen.

Batch-Normalization

Vorteile von Batch-Normalization

- **Schnelleres Training**: Ermöglicht **größere Lernraten** und eine **schnellere Konvergenz**.
- **Stabilität**: Reduziert Probleme durch das Verschwinden von Gradienten bei **Sigmoid oder Tanh**
- **Regularisierung**: Wirkt als eine Art Regularisierung, was die Notwendigkeit von Dropout in einigen Fällen reduzieren kann.
- **Reduzierte Abhängigkeit von der Initialisierung**: Ermöglicht robustere Netzwerke auch bei nicht optimaler Gewichtsinitalisierung.

Nachteile

- **Batch-Abhängigkeit**: Die Berechnungen hängen von der **Batch-Größe** ab.
- Bei sehr **kleinen Batche**s kann dies **instabil** werden.
- erhöhte **Komplexität des Modells** (durch *Libraries* entschärft)
- **Rechenaufwand**: sowohl im Training als auch im Inferencing.

💡 `Komplizierte Ad Hoc Optimisierung welche wahrscheinlich in Zukunft keinen Bestand haben wird?`

Batch-Normalization

⚠️ model.train() ≠ model.eval()

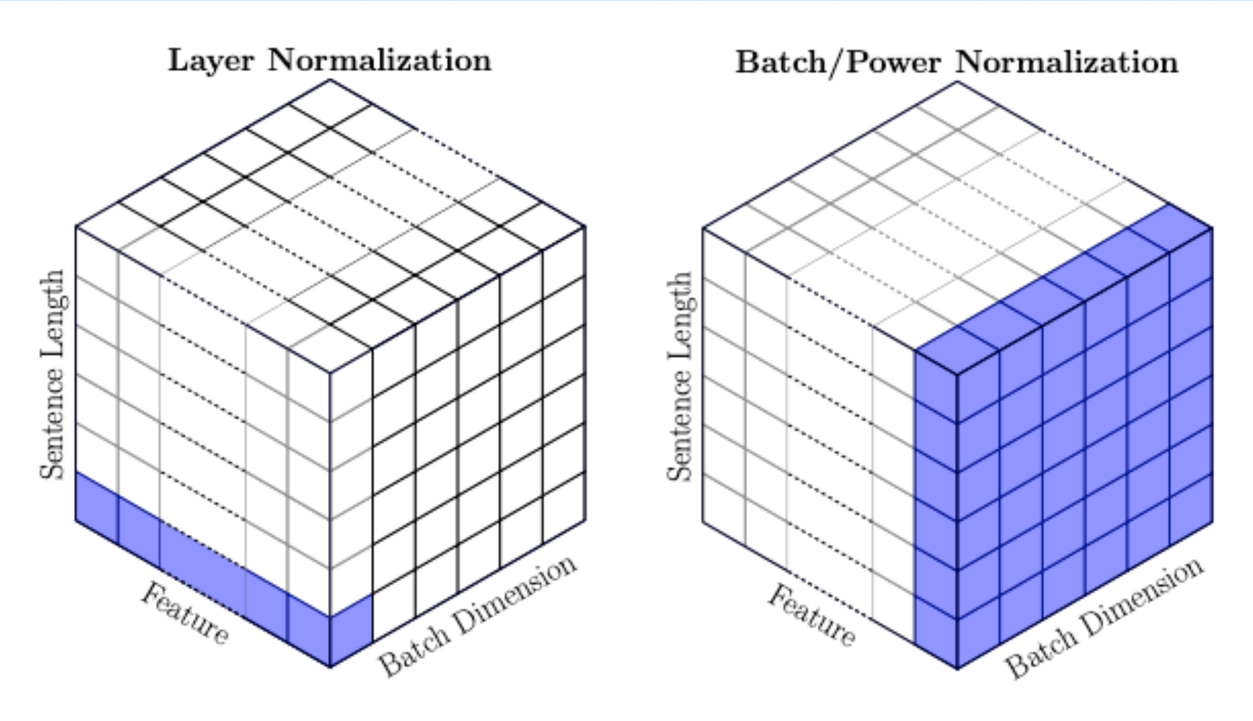
Training vs. Inferencing: Im Training werden Mittelwert und √Varianz pro Batch berechnet. Beim Inferencing verwendet das Netz die globalen Schätzungen, die während des Trainings akkumuliert wurden.

Inferenz = Anwendung des trainierten Models

PS: **Inference** neues **Synonym** für **Vorhersage, Berechnung, Evaluierung, Prediction, forward activation auf echten Daten**

Batch-Normalization

Alternativen: **Layer**-Normalization, **Group**-Normalization und Instance-Normalization sind verwandte Techniken, die für spezielle Anwendungsfälle (z.B. bei kleinen Batches oder RNNs) vorteilhaft sein können.

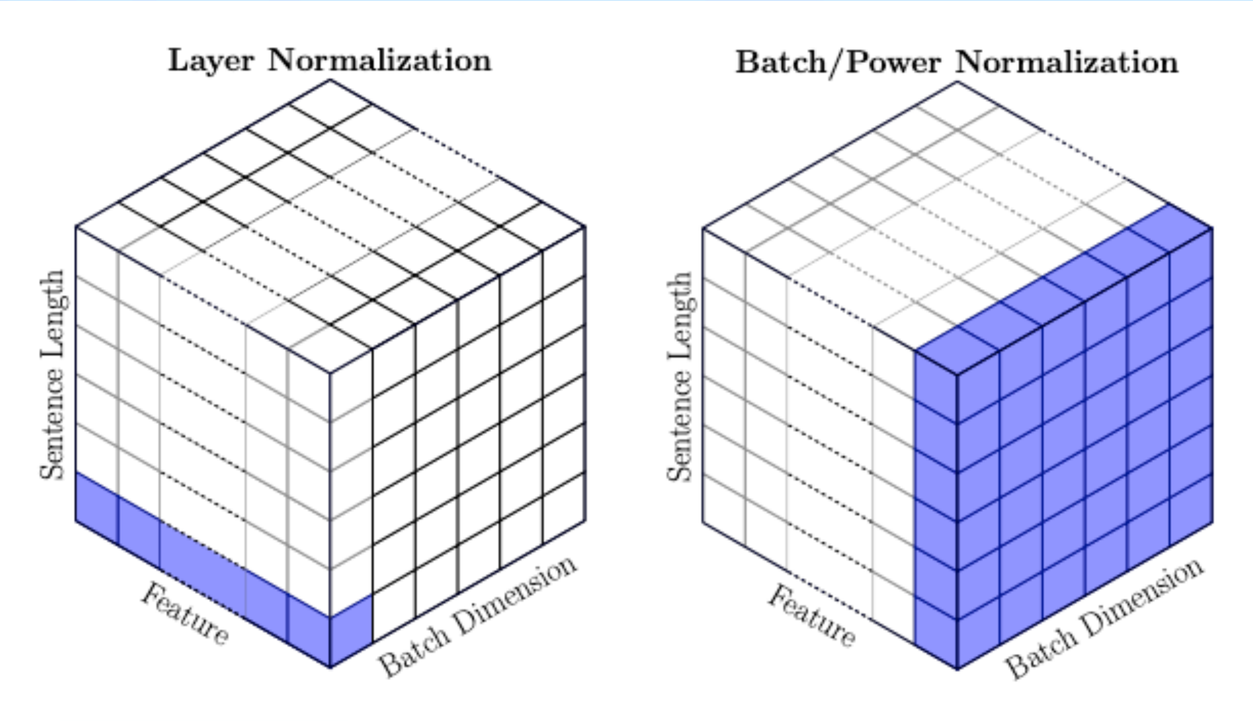


`nn.BatchNorm2d(dim)` einfach als Block verwendbar mit Dimension

Layer-Normalization

Alternativen: **Layer**-Normalization

selber Ansatz wie Batchnorm, nur in der Feature Dimension



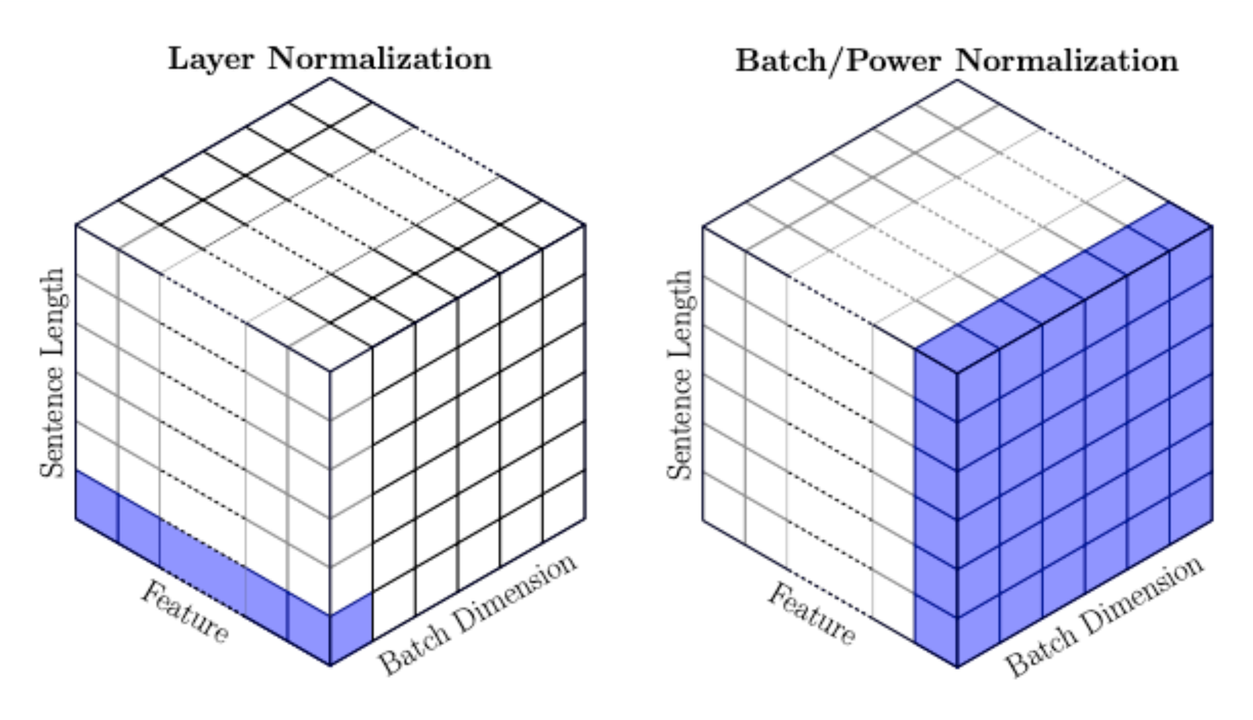
sorgt am Ende dafür, dass alle Feature "*Skalen*" recht einheitlich verteilt sind.

Neuverteilung der Aktivierungen als ≈ Normalverteilung

```
nn.LayerNorm(dim)   einfach als Block verwendbar mit Dimension
```

Layer-Normalization

Alternativen: **Layer**-Normalization
Intuition: alle Aktivierungen ≈ **-1,0,1** => wie dropout
Binäre Feature (**latent** vektor) wie **20-Questions**
Jedes Konzept (Wort) kann mit 20-binär Informationen gespeichert werden 2^20



sorgt am Ende dafür, dass alle Feature "*Skalen*" recht einheitlich verteilt sind.
Neuverteilung der Aktivierungen als ≈ Normalverteilung

```
nn.LayerNorm(dim)   einfach als Block verwendbar mit Dimension
```

Layer-Normalization

Alternativen: **Layer**-Normalization
Intuition: alle Aktivierungen ≈ **-1,0,1** => wie dropout
Binäre Feature (latent Vektor) wie **20-Questions (Rosa/Black Stories)**
Jedes Konzept (Wort) kann mit 20-binär Informationen gespeichert werden
2^20
Lebend: N
Anfassen: J
Raum: J
Menschlich: J
Modern: J
Plastik: J
Leuchtet: ?
Tastatur: J

Batch-Normalization

Alternativen zur Batch-Normalization

1. **Layer-Normalization (LN)**:
   - LN wird vor allem in **Transformer**-Modellen und RNNs eingesetzt. Im Gegensatz zu BN **normalisiert** es die **Aktivierungen über die Merkmale** hinweg, anstatt über die Batch-Dimension. Das macht LN unabhängig von der Batch-Größe und besser geeignet für architekturen mit variabler Eingabelänge, wie bei Text oder Sequenzdaten.
2. **Group-Normalization (GN)**:
   - GN ist besonders nützlich in Szenarien mit kleinen Batches. Es unterteilt die Kanäle in kleinere Gruppen und normalisiert innerhalb dieser Gruppen. GN ist in vielen Computer-Vision-Aufgaben erfolgreich und bietet eine gute Balance zwischen den Vorteilen von BN und LN.
3. **Instance-Normalization (IN)**:
   - IN wird häufig in Stiltransfer-Anwendungen eingesetzt. Es normalisiert die Aktivierungen für jedes einzelne Beispiel unabhängig von der Batch-Größe, was es ideal für Anwendungen macht, in denen das Batch-Normalisierungsverfahren zu ungenauen Statistiken führen würde.
4. **Weight Normalization**:
   - Diese Methode normalisiert die Gewichte anstelle der Aktivierungen und beseitigt somit die Abhängigkeit von der Batch-Größe vollständig. Es wird jedoch seltener verwendet als LN oder GN.
5. **Adaptive Normalization (AdaNorm)**:
   - Neuere Ansätze, wie AdaNorm, passen sich dynamisch an die Normalisierungsstrategie an und kombinieren die Vorteile verschiedener Normalisierungsarten. Diese Methoden sind jedoch noch relativ neu und werden weiterhin erforscht.

Batch-Normalization

Warum?

Internal Covariate Shift:

- Beim Training eines tiefen neuronalen Netzes ändern sich die Verteilungen der Eingaben für jede Schicht kontinuierlich, da die Gewichte des Netzes bei jedem Schritt aktualisiert werden. Dieses Phänomen nennt man *Internal Covariate Shift*.

BN ist besonders effektiv in Convolutional Neural Networks (CNNs). Bei anderen Architekturen, wie Recurrent Neural Networks (RNNs) oder Transformer-Modellen, ist BN weniger effektiv. Hier werden oft andere Normalisierungstechniken wie Layer-Normalization oder Group-Normalization verwendet.

Wohl weil Farbbereiche etc immer ähnlich sind.

Aufgabe

Aufgabe:
Verwende neue Techniken um Cifar-100 auf > 70% zu bringen

in lenet_cifar100_regulation.py