1 #2: Mehrschichtiges feedforward-Netz

Bei dieser Programmierübung wird ein mehrschichtiges feedforward-Netz implementiert, das mithilfe von Backpropagation trainiert wird. Die Implementierung erfolgt zuerst nur unter Zuhilfenahme einer Tensor-Bibliothek und unter Anwendung der Kettenregel. Das mehrschichtige feedforward-Netz unterstützt dabei eine Variablen Anzahl an Layern und Neuronen, sowie verschiedene Aktivierungsfunktionen. Mit diesen Parametern wird dann experimentiert, um verschiedene mathematische Funktionen zu lernen und ein Gefühl für die Optimierung von Netzwerken zu bekommen. Danach erfolgt eine zweite Implementierung des mehrschichtigen feedforward-Netz mithilfe einer Autograd (Automatic Differentiation)-Bibliothek, die auch in State-of-the-art-Anwendungen eingesetzt wird. Die Implementierung wird ebenfalls für die Optimierung der verschiedenen mathematischen Funktionen getestet. Abschließend sollen die Unterschiede der beiden Implementierungen dargestellt werden.

1.1 Implementierung Backpropagation

In dieser Aufgabenstellung soll ein mehrschichtiges feedforward-Netz implementiert werden, das mithilfe von Backpropagation und der Methode des steilsten Abstiegs optimiert wird. Details zu dem Verfahren sind im Skript in Abschnitt „Tiefe Neuronale Netze“ und insbesondere in Abschnitt „Backpropagation“ zu finden. Die Implementierung soll innerhalb der Klasse „NeuralNetwork“stattfinden. Die Methode „feedForward“ berechnet die Ausgaben des Netzwerkes indem die Ausgaben eines Layers als Eingaben des nächsten Layers dienen. Die Methode „backPropagation“ verteilt die Deltas der produzierten Ausgaben und der erwarteten Ausgaben anhand der Kettenregel zurück durch das Netzwerk und aktualisiert die Gewichte und Biases. Um die Implementierung zu testen, können die Methoden „checkFeedForward“ und „checkBackPropagation“ verwendet werden.

Für die Umsetzung dürfen keine weiteren Bibliotheken außer der bereits importierten Torch-Bibliothek verwendet werden. Funktionen die potentiell Verwendung finden sollten: torch.zeros, torch.cat, torch.ones, <Tensor>.shape[n], torch.norm, <Tensor>.mm, <Tensor>.sub, <Tensor>.T, (+, -, *, / für Skalar-Operationen)

In der Vorlesung wird die Implementierung dann vorgestellt, wobei die Verwendung der Kettenregel deutlich gemacht werden sollte.

1.2 Optimierung der Hyperparameter

In dieser Aufgabenstellung sollen für die zu lernenden mathematischen Funktionen Netzwerk-Architekturen aufgestellt und Hyperparameter manuell optimiert werden um ein „optimales“ Ergebnis zu erzielen. (Da die Daten künstlich generiert werden, wird auf die Unterscheidung zwischen Trainings, Validierungs und Testdaten verzichtet. D.h. der Loss sollte so gering wie möglich sein.

Neben dem Endergebnis zählt sollte in dieser Aufgabenstellung das Vorgehen beim Prozess der manuellen Optimierung beschrieben werden. Welche Aktivierungsfunktionen eignen sich besonders für die Abbildung welcher mathematischen Funktion? Welchen Einfluss hat die Learnigrate? Welchen Einfluss haben die Neuronen und die Layer-Anzahl? Welche Wirkung hat die Batch-Größe? Welche Aktivierungsfunktionen sind für „tiefe"-Netze geeignet und warum?

Hinweis: Um einen optimalen Lernprozess zu unterstützen, werden die Gewichte mit Werten aus Zufallsverteilungen gefüllt, die sich in der Praxis für die jeweilige Aktivierungsfunktion bewährt haben. Durch die Zufallswerte ist der Lernprozess nicht 1-zu-1 reproduzierbar. Daher kann es zu Abweichungen kommen. Um den Seed für die Zufallsvariablen festzulegen, kann in Pytorch „torch.manual_seed('Wert')" verwendet werden. Generell bietet es sich bei der Aufgabe an, die Ausgaben der einzelnen Optimierungen im Notebook zu speichern und nicht Live während der Vorstellung zu berechnen.

1.3 Implementierung Backpropagation mit Autograd und Layern

In dieser Aufgabenstellung soll ein mehrschichtiges feedforward-Netz mit den State-of-the-art-Mitteln von Pytorch implementiert werden. Die Implementierung erfolgt in der Klasse „NeuralNetworkTorch" mit den Pytorch-Bibliotheken „torch.nn", „torch.autograd" und „torch.optim". Die Implementierung kann anhand der Beispiele aus der vorherigen Aufgabe überprüft werden. Gibt es Abweichungen bei den Ergebnissen, wenn mit denselben Netz-Architekturen und Hyperparametern optimiert wird?

In der Vorlesung wird die Implementierung vorgestellt und auf die verwendeten Bibliotheken und Konzepte aus Pytorch eingegangen.

1.4 Alternative Optimierungsalgorithmen

In den bisherigen Aufgaben wurde mithilfe des stochastischen Gradientverfahrens optimiert. In der Praxis werden weiterführende Algorithmen oder Optimierer verwendet, die das Gradientenverfahren verbessern. Z.B. kann das Gradientenverfahren um ein „Momentum" erweitert werden, das für die Berechnung des Delta-Updates nicht nur auf den Gradienten sondern auch auf die Delta-Updates der vorherigen Iterationen aufsetzt. Die Idee stammt aus der Physik, um Geschwindigkeit in eine Richtung aufzubauen und Oszillationen zu vermeiden. Generell ist die Suche nach den besten Optimierungsalgorithmen immer noch ein Forschungsschwerpunkt bei der Erforschung und Optimierung neuronaler Netzwerke.

In dieser Aufgabe sollen die mathematischen Funktionen der vorherigen Aufgaben nochmal erneut mit denselben Netz-Architekturen und verschiedenen Optimierern von https://pytorch.org/docs/stable/optim.html optimiert werden. Der Hyperparameter LearningRate muss ggf. angepasst werden da er passend zum verwendeten Algorithmus eingestellt werden muss. Lassen sich bessere Er-

gebnisse mit dem SGD-Optimierer erzielen, wenn „Momentum“ aktiviert wird? Wie verhält sich der aktuelle State-of-the-art-Optimierer „Adam“ im Vergleich?

In der Vorlesung sollen die Ergebnisse im Vergleich dargestellt werden.