OST
Ostschweizer
Fachhochschule

3 Bias-Variance und Regularisierung

Systemtechnik BSc
FS 2025

Lernziele

Nach dem Bearbeiten dieser Übungsserie ...

- können Sie mit `pytorch-lightning` einfache sequentielle neuronale Netze für die Klassifizierung und Regression aufbauen.
  Aufgaben:
- können Sie beurteilen, ob ein NN-Modell eine hohe Varianz (overfitting) oder eher einen hohen Bias (underfitting) aufweist. Sie sind in der Lage, mit geeigneten Regularisierungsmassnahmen eine Überanpassung eines Modells zu vermeiden (weight regularization [L1,L2], early stopping, dropout, model complexity)
  Aufgaben: 1

Über- und Unteranpassung

Overfitting

Overfitting bedeutet: Unser Modell passt sich den Trainingsdaten *zu sehr* an. Es ist wichtig zu lernen, wie man mit der Überanpassung umgeht. Obwohl es oft möglich ist, eine hohe Genauigkeit im *Trainingsdatensatz* zu erreichen, wollen wir eigentlich Modelle entwickeln, die sich gut auf einen *Testdatensatz* verallgemeinern lassen (oder auf Daten, die sie noch nie gesehen haben).

Underfitting

Das Gegenteil von Overfitting ist *Underfitting*. Underfitting liegt vor, wenn die Performance des Modells auf den Trainingsdaten noch verbesserungswürdig sind. Dies kann aus verschiedenen Gründen geschehen: Wenn das Modell nicht leistungsfähig genug ist, überreguliert ist oder einfach nicht lange genug trainiert wurde. Das bedeutet, dass das Netz die relevanten Muster in den Trainingsdaten nicht gelernt hat.
Wenn Sie jedoch zu lange trainieren, wird das Modell anfangen, sich zu sehr anzupassen und Muster aus den Trainingsdaten zu lernen, die sich nicht auf die Testdaten übertragen lassen. Wir müssen also ein Gleichgewicht finden. Es ist nützlich zu wissen, wie man für eine angemessene Anzahl von Epochen trainiert, wie wir weiter unten erläutern werden.

Regularisierung

Um eine Überanpassung zu vermeiden, ist die beste Lösung die Verwendung vollständigerer Trainingsdaten. Der Datensatz sollte die gesamte Bandbreite der Eingaben abdecken, die das Modell verarbeiten soll. Zusätzliche Daten sind nur dann sinnvoll, wenn sie neue und interessante Fälle abdecken. Der einfachste Weg, eine Überanpassung zu verhindern, ist, mit einem kleinen Modell zu beginnen: Ein Modell mit einer kleinen Anzahl von lernbaren Parametern (die durch die Anzahl der Schichten und die Anzahl der Einheiten pro Schicht bestimmt wird). Beim Deep Learning wird die Anzahl der lernbaren Parameter in einem Modell oft als *Kapazität* des Modells bezeichnet. Deep-Learning-Modelle neigen dazu, sich gut an die Trainingsdaten anzupassen, aber die eigentliche Herausforderung ist die Verallgemeinerung, nicht die Anpassung.
Wenn das Netzwerk andererseits nur über begrenzte Speicherressourcen verfügt, kann es das Mapping nicht so leicht erlernen. Um seinen Verlust zu minimieren, muss es komprimierte Darstellungen lernen, die eine höhere Vorhersagekraft haben. Wenn Sie Ihr Modell jedoch zu klein machen, wird es Schwierigkeiten haben, sich an die Trainingsdaten anzupassen. Es gibt ein Gleichgewicht zwischen „zu viel Kapazität“ und „nicht genug Kapazität“.
Ein Modell, das auf umfangreicheren Daten trainiert wurde, wird natürlich besser verallgemeinern. Wenn dies nicht mehr möglich ist, besteht die nächstbeste Lösung darin, Techniken wie die *Regularisierung* anzuwenden. Diese schränken die Menge und Art

der Informationen ein, die Ihr Modell speichern kann. Wenn sich ein Netzwerk nur eine kleine Anzahl von Mustern merken kann, wird es durch den Optimierungsprozess gezwungen, sich auf die auffälligsten Muster zu konzentrieren, die eine bessere Chance haben, gut zu verallgemeinern. In dieser Übungsaufgabe werden wir verschiedene *gängige Regularisierungstechniken* untersuchen und sie zur Verbesserung eines Klassifizierungsmodells einsetzen.

Aufgabe 1. Benchmark: Über- und Unteranpassung

(a) **Daten laden:** Laden Sie den Datensatz `HIGGS.csv.gz` von mlphysics.ics.uci.edu herunter. Achtung: Es sind rund 2.7 GB an Daten. Legen Sie den Datensatz auf einen Ordner Ihrer Wahl ab. Sie brauchen das komprimierte File nicht zu entzippen.

Die Daten wurden mit Hilfe von Monte-Carlo-Simulationen erstellt. Die ersten 21 Merkmale (Spalten 2-22) sind kinematische Eigenschaften, die von den Teilchendetektoren im Beschleuniger gemessen wurden. Die letzten sieben Merkmale sind Funktionen der ersten 21 Merkmalen; es handelt sich dabei um hochrangige Merkmale, die von Physikern abgeleitet wurden, um zwischen den beiden Klassen zu unterscheiden. Das Ziel dieser Übungsserie ist nicht die Teilchenphysik, daher sollten Sie sich nicht mit den Details des Datensatzes beschäftigen. Er enthält 11'000'000 Beispiele, jedes mit 28 Merkmalen und einem binären Klassenlabel.

(b) **Template:** Öffnen Sie das Jupyter-Template `overfit_and_underfit_TEMPLATE.ipynb` und führen Sie die erste Code-Zelle aus, um die Daten zu laden. Passen Sie den Datenpfad entsprechend an. Beim Setup kann das Package `ipywidgets` installiert werden für schönere Ladebalken während des Trainings. Das Jupyter-Template enthält folgende Hilfsfunktionen:

Funktion	Beschreibung
`download_file_if_not_exists`	Laden der Daten
`config_dict`	für allgemeine Einstellungen
`HiggsDataset`	Klasse für die Daten
`create_model`	für das einfache Erstellen der Architektur
`LightningModel`	Klasse für das Training
`train_model`	allgemeine Einstellungen für das Training für mehr Übersichtlichkeit
`train_with_early_stopping`	Early Stopping, um abzubrechen, falls das Modell nicht mehr verbessert

Tabelle 1: Übersicht der Funktionen und ihrer Beschreibungen

(c) **Winziges Modell**: Um eine geeignete Modellgrösse zu finden, starten Sie am besten mit relativ wenigen Schichten und Parametern. Beginnen dann, die Grösse der Schichten zu erhöhen oder neue Schichten hinzuzufügen, bis Sie eine Verringerung des Validierungsverlustes feststellen. Wir starten mit einem einfachen Modell, das nur `nn.Linear`-Layer als Basis verwendet. Erstellen Sie dann grössere Versionen und vergleichen Sie die Performence dieser Modelle mit einer geeigneten Metrik. Die Definition der Anzahl Lagen und Neuronen wird über ein Dictionary `config_dict` definiert.

```
config_tiny = {
  "model": {
    "input_size": 28,  # Eingangsgroesse des Modells
    "tiny": {
      "hidden_layers": [16],
      "dropout_rate": 0.0,
      "l2_reg": 0.0,
    },  # Konfiguration fuer Tiny-Modell
  },
}
```

Trainieren Sie das kleine Modell mit folgenden Schritten. Der Trainings- und Validierungsloss wird geloggt und anschliessend für die Analyse des Bias-Variance Tradeoffs geplottet.

```
print("Training Basis Tiny Model...")
```

```
# printe die Konfiguration des Modells
print(config_tiny["model"]["tiny"])
tiny_model = create_model(
config_tiny["model"]["input_size"], **config_tiny["model"]["tiny"]
)
tiny_lightning_model = LightningModel(tiny_model)
tiny_trained = train_with_early_stopping(
tiny_lightning_model, train_loader, validate_loader)
```

(d) **Kleines Modell**: Um zu sehen, ob Sie die Leistung des kleinen Modells übertreffen können, trainieren Sie nach und nach einige grössere Modelle. Versuchen Sie es mit zwei versteckten Schichten mit je 16 Einheiten:

```
config_small = {
  "model": {
    "input_size": 28,  # Eingangsgroesse des Modells
    "small": {
      "hidden_layers": [16, 16],
      "dropout_rate": 0.0,
      "l2_reg": 0.0,
    },  # Konfiguration fuer Small-Modell
  },
}
```

Trainiert wird das Modell wie folgt:

```
print("Training Small Model (Basis, [16,16])...")
# printe die Konfiguration des Modells
print(config_small["model"]["small"])
small_model = create_model(
config_small["model"]["input_size"], **config_small["model"]["small"]
)
small_lightning_model = LightningModel(small_model)
small_trained = train_with_early_stopping(
small_lightning_model, train_loader, validate_loader)
```

(e) **Mittleres Modell**: Testen Sie nun ein Modell mit drei versteckten Schichten (hidden layers) mit je 64 Einheiten.

```
config_medium = {
  "model": {
    "input_size": 28,  # Eingangsgroesse des Modells
    "medium": {
      "hidden_layers": [64, 64, 64],
      "dropout_rate": 0.0,
```

```
      "l2_reg": 0.0,
    },  # Konfiguration fuer Medium-Modell
  },
}
```

Auch dieses Modell wird wie folgt trainiert.

```
print("Training Medium Model (Basis, [64,64,64])...")
# printe die Konfiguration des Modells
print(config_medium["model"]["medium"])
medium_model = create_model(
config_medium["model"]["input_size"], **config_medium["model"]["medium"]
)
medium_lightning_model = LightningModel(medium_model)
medium_trained = train_with_early_stopping(
medium_lightning_model, train_loader, validate_loader)
```

(f) **Grosses Modell**: Als nächstes fügen wir zu diesem Benchmark ein Netzwerk hinzu, das viel mehr Kapazität hat, weit mehr als das Problem rechtfertigen würde:

```
config_large = {
  "model": {
    "input_size": 28,  # Eingangsgroesse des Modells
    "large": {
      "hidden_layers": [512, 512, 512, 512],
      "dropout_rate": 0.0,
      "l2_reg": 0.0,
    },  # Konfiguration fuer Large-Modell
  },
}
```

```
print("Training Large Model (Basis, [512,512,512,512])...")
# printe die Konfiguration des Modells
print(config_large["model"]["large"])
large_model = create_model(
config_large["model"]["input_size"], **config_large["model"]["large"]
)
large_lightning_model = LightningModel(large_model)
large_trained = train_with_early_stopping(
large_lightning_model, train_loader, validate_loader)
```

(g) **Interpretation**: Es werden alle Lernkurven zum Modell auch geplottet, beschreiben Sie in wenigen Sätzen, ob eine Über oder Unteranpassung erkennbar ist Anhand des Trainings- und Validierungslosses.

(h) **Weight Regularization**: Vielleicht kennen Sie das Prinzip von OCCAMS Rasiermesser: Wenn es zwei Erklärungen für etwas gibt, ist die Erklärung, die am wahrscheinlichsten richtig ist, die „einfachste“, diejenige, die die wenigsten Annahmen enthält. Dies gilt auch für die Modelle, die von neuronalen Netzen gelernt werden: Bei bestimmten Trainingsdaten und einer Netzarchitektur gibt es mehrere Sätze von Gewichtungswerten (mehrere Modelle), die die Daten erklären könnten, und bei einfacheren Modellen ist die Wahrscheinlichkeit einer Überanpassung geringer als bei komplexen Modellen.

Ein „einfaches Modell“ ist in diesem Zusammenhang ein Modell, bei dem die Verteilung der Parameterwerte eine geringere *Entropie* aufweist (oder ein Modell mit insgesamt weniger Parametern, wie wir im obigen Abschnitt gesehen haben). Eine gängige Methode zur Abschwächung der Überanpassung besteht daher darin, die Komplexität eines Netzes einzuschränken, indem seine Gewichte gezwungen werden, nur kleine Werte anzunehmen, wodurch die Verteilung der Gewichtswerte „regelmässiger“ wird. Dies wird als *Gewichtsregularisierung* bezeichnet und erfolgt durch Hinzufügen von Kosten zur Verlustfunktion des Netzes, die mit grossen Gewichten verbunden sind. Diese Kosten gibt es in zwei Varianten:

- **L1-Regularisierung**, bei der die hinzugefügten Kosten proportional zum absoluten Wert der Gewichtskoeffizienten sind (d. h. zur so genannten L1-Norm der Gewichte).
- **L2-Regularisierung** bei der die zusätzlichen Kosten proportional zum Quadrat des Wertes der Gewichtungskoeffizienten sind (d.h. zu dem, was man die quadrierte L2-Norm der Gewichte nennt). Die L2-Regularisierung wird im Zusammenhang mit neuronalen Netzen auch als Gewichtsabnahme bezeichnet.

Die L1-Regularisierung verschiebt einige Gewichte genau gegen Null, was ein spärliches Modell fördert. Die L2-Regularisierung bestraft die Parameter der Gewichte, ohne sie spärlich zu machen, da die Strafe bei kleinen Gewichten gegen Null geht - ein Grund, warum L2 häufiger verwendet wird.

In PyTorch Lightning wird die L2-Gewichtsregulierung (auch bekannt als Weight Decay) typischerweise über den Optimierer hinzugefügt.

```
def configure_optimizers(self):
  optimizer = optim.Adam(self.parameters(), lr=self.lr, weight_decay=
    self.l2_reg)  # Optimierer

```

(i) **Dropout**: Dropout ist eine der effektivsten und am häufigsten verwendeten Regularisierungstechniken für neuronale Netze, die von Hinton und seinen Studenten an der Universität von Toronto entwickelt wurde. Die intuitive Erklärung für Dropout ist, dass sich einzelne Knoten im Netz nicht auf die Ausgaben der anderen verlassen können, sondern dass jeder Knoten für sich selbst nützliche Merkmale ausgeben muss. Dropout, angewandt auf eine Schicht, besteht darin, dass während des Trainings eine Anzahl von Ausgangsmerkmalen der Schicht zufällig „weggelassen“ (d. h. auf Null gesetzt) wird. Nehmen wir an, eine gegebene Schicht hätte normalerweise einen Vektor `[0.2, 0.5, 1.3, 0.8, 1.1]` für eine gegebene Eingabeprobe während des Trainings geliefert; nach Anwendung von Dropout wird dieser Vektor einige zufällig verteilte Nulleinträge haben, z. B. `[0, 0.5, 1.3, 0, 1.1]`.

Die *Dropout-Rate* ist der Anteil der Merkmale, die mit Nullen versehen werden; sie wird normalerweise zwischen 0.2 und 0.5 festgelegt. Zur Testzeit werden keine Einheiten herausgenommen, stattdessen werden die Ausgabewerte der Schicht um den Faktor der Dropout-Rate herunterskaliert, um die Tatsache auszugleichen, dass mehr Einheiten aktiv sind als zur Trainingszeit.

Über das `config_` Dictionary kann die `dropout_rate` mitgegeben werden. Sie wird dann bei `create_model` hinzugefügt und wie in der Funktion `create_model` definiert erstellt.

```
def create_model(input_size, hidden_layers, dropout_rate=0.0, l2_reg
    =0.0):
  layers = []
  prev_dim = input_size
  for layer_dim in hidden_layers:
    layers.append(nn.Linear(prev_dim, layer_dim))
    layers.append(nn.ELU())
    if dropout_rate > 0:
      layers.append(nn.Dropout(dropout_rate))
    prev_dim = layer_dim
  layers.append(nn.Linear(prev_dim, 1))
  return nn.Sequential(*layers)
```

```
12
```

(j) **Kombinierte Anwendung von L2 und Dropout**: In diesem Schritt werden L2 und Dropout noch zusammen kombiniert.

(k) **Spielen Sie mit verschiedenen Parametern und Kombinationen:** Bewerten Sie die Kurven. Versuchen Sie das beste Modell zu finden, welche Hyperparameter hat dieses Modell?

Lösungen

Lösung 1.
overfit_and_underfit_SOLUTION-PyTorch.ipynb