Lecture 5: Regularisierung

BA-INF 153: Einführung in Deep Learning für Visual Computing

Prof. Dr. Reinhard Klein

Nils Wandel

Informatik II, Universität Bonn

29.05.2024

Recap

Neuronale Netze und Backpropagation

- Formale Einführung von Feed-forward Netzwerken
- verschiedene Aktivierungsfunktionen
- Symmetrien in Neuronalen Netzen
- Universal Function Approximation Theorem (ohne Beweis)
- Backpropagation-Algorithmus (Herleitung mit Kettenregel und Jacobi-Matrizen)

Today's Lecture:

1. Gradient Clipping
2. Parameter-Initialisierung
3. Regularisierungs-Terme in Loss-funktionen
4. Reduzierung der Modell-Kapazität
5. Data-Augmentation
6. Bagging
7. Dropout

Part 1

Gradient Clipping

Chapter 8.4 von **Goodfellow-et-al-2016-Book**.

Exploding Gradients

Neuronale Netze mit vielen Layern (d.h. tiefe Netzwerke) besitzen oft sehr steile Regionen in der Loss-Funktion. Solche "Klippen"-Strukturen können durch die Multiplikation mit vielen grossen Netzwerkgewichten entstehen.

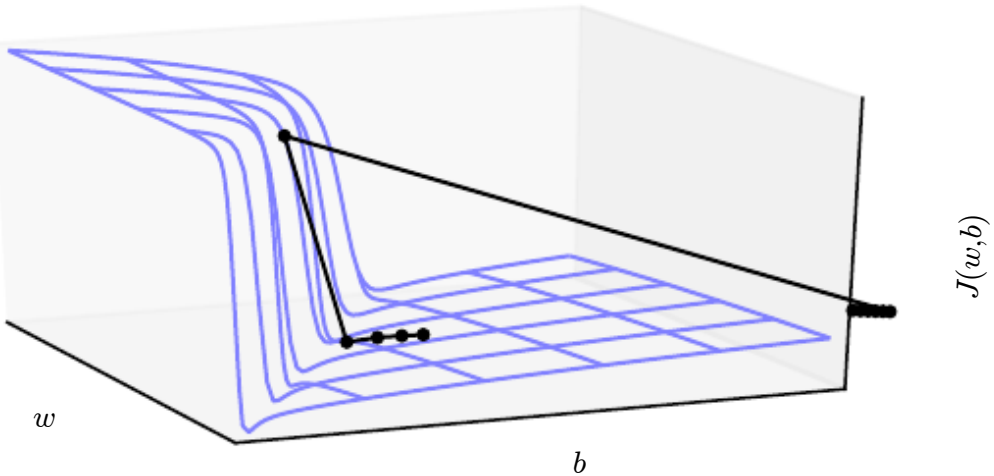


Figure 8.3: The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that had been done. Figure adapted with permission from Pascanu *et al.* (2013).

Figure: 8.3 von Goodfellow.

Exploding Gradients

Wenn die Optimierung auf eine solche Klippe trifft, dann kann der Gradient "explodieren", was zu einem sehr grossen Updateschritt führt und die vorangegangene Optimierung zu Nichte macht.

Gradient Clipping

Explodierende Gradienten können durch *Gradient Clipping* korrigiert werden. Dabei begrenzt man den maximalen Gradienten, um zu grosse Updateschritte zu verhindern.
Hierbei gibt es 2 Möglichkeiten:

- **Norm-Clipping:** Überschreitet die Norm des Gradienten einen Schwellenwert, so wird der Gradient auf den Schwellenwert herunterskaliert.
- **Value-Clipping:** Überschreitet der Betrag eines einzelnen Eintrags des Gradienten einen Schwellenwert, so wird dieser auf den Schwellenwert herunterskaliert.

Part 2

Parameter-Initialisierung

Chapter 8.4 von **Goodfellow-et-al-2016-Book**.

Zu kleine Netzwerk-Gewichte

Wir haben in der letzten Vorlesung gezeigt, dass wir nicht-lineare Aktivierungsfunktionen f anwenden möchten, da sonst das Netzwerk zu einem einzelnen linearen Layer kollabiert werden könnte.
Zum Beispiel kann man dafür die σ-Aktivierungsfunktion verwenden. Wenn die Gewichtsmatrix allerdings nahe 0 ist ($W_{ij} \ll 1$), dann ist auch $W \cdot \boldsymbol{x}$ nahe 0 und die σ-Aktivierungsfunktion nahezu linear. Dadurch kann das Netzwerk in ein nahezu lineares Modell kollabiert werden (was nicht erwünscht ist).

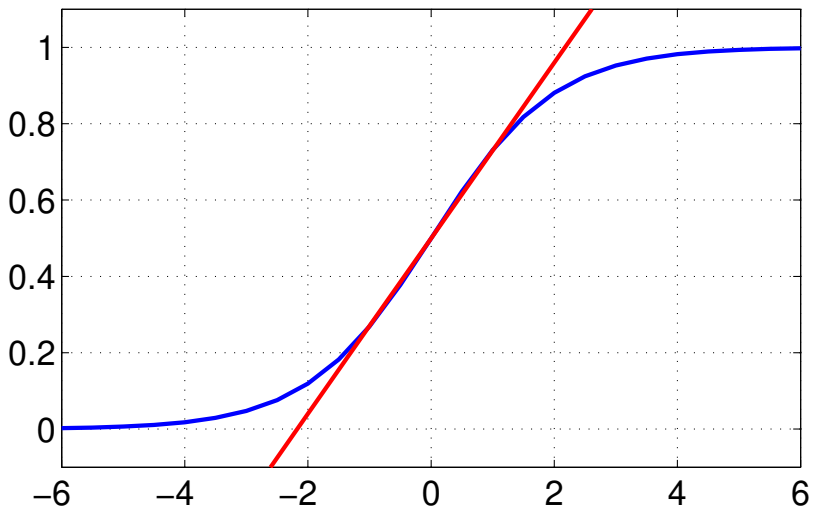


Figure: Die Sigmoid-Aktivierungsfunktion ist nahezu linear für kleine x

Zu grosse Netzwerk-Gewichte

Wenn die Netzwerk-Gewichte zu gross sind, dann werden auch die Beträge der Aktivierungen sehr gross und Sigmoid-Aktivierungsfunktionen nahezu immer gesättigt (liefern Werte nahe 0 oder 1). Dort sind die Gradienten der Sigmoid-Funktion allerdings sehr nahe bei 0 (siehe Abbildung), wodurch das Training erschwert wird. Dieses Sättigungs-Verhalten ist auch ein Grund, warum in hidden Layers üblicherweise ReLU / LReLU etc gegenüber σ- oder $\tanh$-Aktivierungsfunktionen bevorzugt werden.

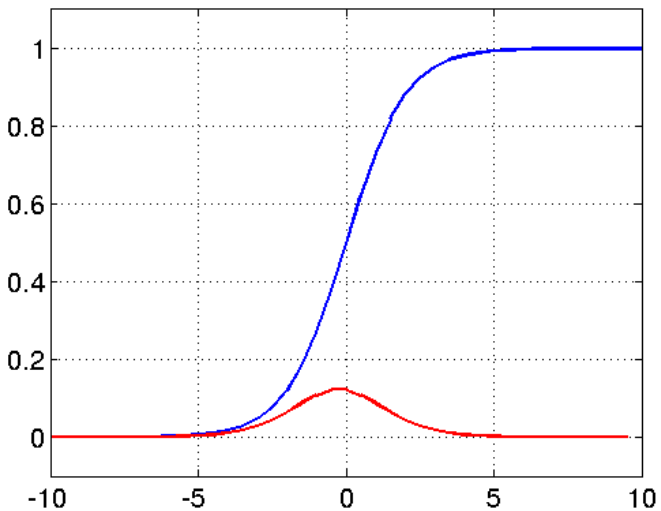


Figure: Sigmoid Funktion mit Ableitung

Skalierung der Eingabedaten

Der Wertebereich der Eingabedaten bestimmt den effektiven Wertebereich der Gewichte in den ersten Layern, was einen grossen Einfluss auf die Qualität des gesamten Neuronalen Netzes hat.
Es wird empfohlen, alle *Eingabedaten* so zu *skalieren*, sodass der *Mittelwert = 0* und die *Standardabweichung = 1* ist. Dies garantiert, dass alle Eingangs-Dimensionen gleich behandelt werden und die Initialisierung der Gewichts-matrizen in einem vernünftigen Intervall liegen.

Initialisierung der Netzwerk-Gewichte

Wenn wir das Training starten, initialisieren wir die Gewichte üblicherweise mit kleinen zufälligen Werten. D.h. das Netzwerk startet nahezu linear und wird, wenn die Gewichte während des Trainings grösser werden nicht-linear. Wenn wir mit zu grossen Gewichten starten, riskieren wir, dass wir die Aktivierungsfunktionen sättigen und somit keine guten Gradienten für das Training erhalten.

Frage: Was wäre, wenn wir die Gewichte alle mit 0 initialisiern?

Parameter Initialisierung

Der Startpunkt spielt eine wichtige Rolle bei gradienten-basierten Verfahren, denn er bestimmt wie schnell und in ein wie gutes lokales Minimum das Verfahren konvergiert.

Breaking Symmetries

Damit dies geschehen kann, darf es keine Symmetrien in den hidden Units eines Layers geben (z.B. indem alle Gewichte auf einen konstanten Wert 0 gesetzt werden). In diesem Fall würden alle Neurone die gleichen Aktivierungen berechnen, gleiche Gradienten bezüglich des Losses erhalten und gleich geupdatet werden. Somit würde sich jedes Layer so verhalten, als würde es nur ein einzelnes Neuron enthalten. Dies wäre natürlich unerwünscht, weshalb man die Gewichte aus einer Zufallsverteilung (z.B. Uniform oder Normalverteilt) zieht.

Parameter Initialisierung

Gradienten bezüglich des Losses

Wenn wir die Start-Parameter aus einer Zufallsverteilung ziehen, müssen wir darauf achten, dass sie in einer Region liegen, in der die Gradienten bezüglich des Losses weder zu klein noch zu gross sind.
Die Gradienten können mit der Kettenregel wie folgt berechnet werden (siehe Backpropagation von letzter Woche):

$$\frac{\partial}{\partial \boldsymbol{x}} L_2 = J_{L_2}(\hat{\boldsymbol{y}}) \cdot J_{f^{\text{out}}}(\boldsymbol{a}^{(3)}) \cdot W^{(3)} \cdot J_{f^{(2)}}(\boldsymbol{a}^{(2)}) \cdot W^{(2)} \cdot J_{f^{(1)}}(\boldsymbol{a}^{(1)}) \cdot W^{(1)}$$

Hierbei gibt es 2 wichtige Komponenten:

1. Jacobi-Matrizen der Aktivierungsfunktionen $J_{f^{(i)}}(\boldsymbol{a}^{(i)})$
2. Gewichtsmatrizen $W^{(i)}$

Damit die Gradienten in einem sinnvollen Bereich liegen, sollten der Betrag der Eigenwerte von $J_{f^{(i)}}(\boldsymbol{a}^{(i)})$ und $W^{(i)}$ ungefähr bei 1 liegen. Wenn die Eigenwerte zu klein wären, würden die Gradienten mit jedem Layer exponentiell kleiner. Wenn die Eigenwerte zu gross wären, würden die Gradienten mit jedem Layer exponentiell grösser und explodieren.

Parameter Initialisierung

Jacobi-Matrizen der Aktivierungsfunktionen

Die Jacobi-Matrizen der Aktivierungsfunktionen $J_{f^{(i)}}(\boldsymbol{a}^{(i)})$ hängen von der Aktivierungsfunktion $f^{(i)}$ und den Aktivierungen $\boldsymbol{a}^{(i)}$ ab. Das bedeutet, dass wir durch die Parameter-Initialisierung von Beginn an gewährleisten sollten, dass die Aktivierungen in einem Bereich liegen, in dem die Ableitung der Aktivierungsfunktion in der Nähe von 1 liegt.
Für σ- oder $\tanh$-Aktivierungsfunktionen bedeutet dies, dass der Betrag der Aktivierungen nicht zu gross sein sollte, da die Ableitung sonst sehr klein wird. Für ReLU-artige Aktivierungsfunktionen sollten die Aktivierungen nicht häufig kleiner 0 sein, da sonst die Ableitung (sehr Nahe) bei 0 liegen würde.

Parameter Initialisierung

Die Wahrscheinlichkeitsverteilung der Aktivierungen eines Neurons hängt von 3 Faktoren ab:

- **Eingangssignale:** Die Wahrscheinlichkeitsverteilung der Eingangssignale sollten ungefähr um 0 zentriert sein und eine Varianz von 1 besitzen.
- **Gewichte:** Die Aktivierung enthält eine gewichtete Summe der Eingangssignale. Unter der Annahme, dass die Eingangssignale unkorreliert sind und Varianz 1 haben, entspricht die Varianz der Summe der Eingangssignale der Anzahl Eingangssignale (n_{in}). Deshalb kann es Sinn machen, die Gewichtsinitialisierung mit $1/\sqrt{n_{\text{in}}}$ zu skalieren.
- **Bias:** Der Bias verschiebt den Mittelwert der Aktivierungsverteilung. Üblicherweise kann er bei 0 belassen werden. Bei ReLU-artigen Aktivierungsfunktionen möchte man jedoch häufig einen leicht positiven Bias, damit Werte kleiner 0 seltener vorkommen.

Skalierung der Gewichtsinitialisierung als ein Hyperparameter

Wenn es die Rechenressourcen erlauben, können wir die anfängliche Skalierung der Gewichte als Hyperparameter behandeln. Eine Möglichkeit, die Skalierung abzuschätzen, ist die Betrachtung des Bereichs bzw. der Standardabweichung der Aktivierungen und Gradienten in einem einzelnen Minibatch von Daten. Wenn die Gewichte zu klein sind, werden die Aktivierungen der hidden Units von Layer zu Layer kleiner werden.

Wir können wiederholt nach den ersten Layern mit zu kleinen Aktivierungen suchen und dessen Gewichtsinitialisierung erhöhen, um ein Netz mit durchgängig vernünftigen Anfangsaktivierungen zu erhalten.

Part 3

Regularisierung

Chapter 7.1 in **Goodfellow-et-al-2016-Book**.
Chapter 5.5.1, 5.5.3 in **bishop2006pattern**

Overfitting

In einem Netzwerk wird die Anzahl der Eingabe und Ausgabe-Units allgemein durch das Datenset bestimmt. Die Anzahl der hidden Units M ist jedoch ein Kapazitätsparameter, der gefunden werden muss.

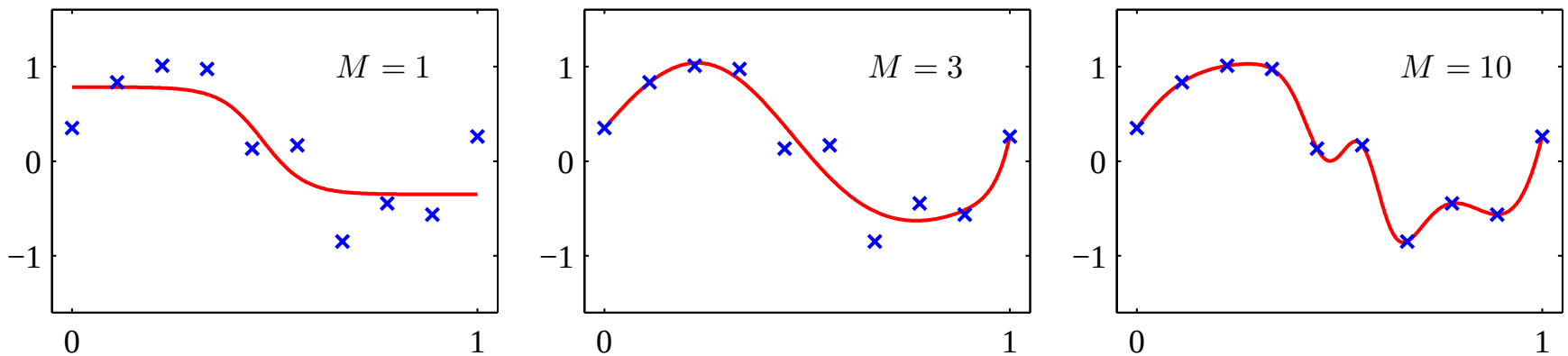


Figure: 5.9 von Bishop: Ein 2-Layer Netzwerk mit $M = 1, 3, 10$ hidden Units.

Umso höher M, desto besser fittet ein Modell die Trainingsdaten. Das Ziel ist jedoch auch auf ungesehene Testdaten zu generalisieren. (Tiefe) neuronale Netze sind durch ihre grosse Modellkapazität besonders gefährdet für Overfitting und um dem vorzubeugen können wir Regularisierungsstrategien verwenden.

Definitionen

Was bedeutet Regularisierung?

Regularisierung kann definiert werden als jede Art der Modifikation eines Lernalgorithmus mit dem Zweck, den *Generalisierungsfehler* zu reduzieren ohne den *Trainingsfehler* signifikant zu erhöhen.

Regularisierungsstrategien

- Extra-Terme in der Lossfunktion
- Extra-Bedingungen für die Modellparameter

Diese Regularisierungsstrategien können Vorwissen enthalten ("Prior" im Satz von Bayes) oder eine Preferenz zu einfacheren Modellen vorgeben um die Generalisierung des Modells zu fördern.

Regularisierung in Deep Learning

Die meisten Regularisierungsstrategien fokussieren auf den Schätzer um einen guten Trade-off zwischen Bias und Varianz zu finden (siehe Vorlesung 2, Slide 54). Dabei sollte die Varianz signifikant gesenkt werden ohne den Bias zu stark zu steigern.

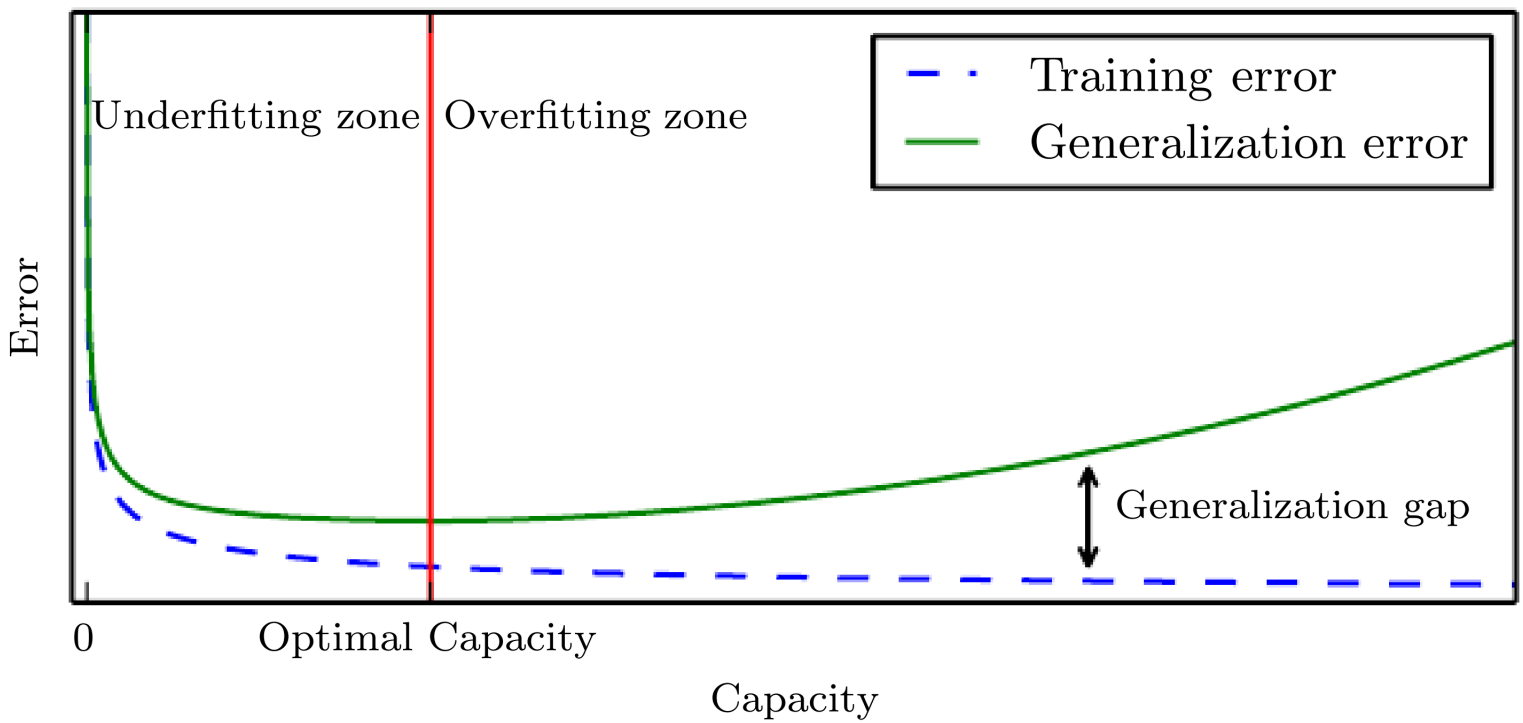


Figure: Bias-Variance Trade-off

Regularisierung in Deep Learning

3 Bias-Variance Situationen

Beim Training kann man den Bias-Varianz Trade-off wie folgt betrachten:

1. Das Modell kann den wahren Daten-generierenden Prozess nicht erfassen und underfittet ($\Rightarrow$ hoher Bias)
2. Das Modell passt zum wahren Daten-generierenden Prozess (das ist unser Ziel)
3. Das Modell enthält den wahren Daten-generierenden Prozess sowie viele weitere mögliche Daten-generierende Prozesse und overfittet ($\Rightarrow$ hohe Varianz)

(Tiefe) neuronale Netze fallen häufig in die dritte Kategorie und das Ziel von Regularisierung ist es, sie in die zweite Kategorie zu bewegen.

Regularisierung in Deep Learning

In der Praxis liegen die Anwendungen von tiefen neuronalen Netzen in sehr komplexen Domänen wie z.B. Bildern, Audio-Sequenzen oder Text, wobei es unmöglich ist, den wahren Daten-generierenden Prozess zu simulieren (oder zu evaluieren).

Dabei geht der Begriff der "Modell-komplexität" im Kontext von tiefen NN nicht nur um die Anzahl der Parameter sondern auch um die Regularisierung bzw. innere Architektur eines Modells.

$\Rightarrow$ wir müssen Strategien finden, um große, tiefe neuronale Netze zu regularisieren.

Parameter Norm Penalties

Eine einfache Methode um die Netzwerk-Kapazität einzuschränken ist, einen Penalty-Term für die Parameter-Norm $\Omega(\theta)$ in der Loss-Funktion L einzuführen.

Die regularisierte Loss-Funktion bezeichnen wir nun als $\tilde{L}$:

$$\tilde{L}(\theta; \boldsymbol{X}, \boldsymbol{y}) = L(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha \cdot \Omega(\theta)$$

wobei α ein Hyperparameter ist, um den relativen Effekt des Penalty-Terms zu gewichten (Umso größer α ist, desto stärker der Regularisierungseffekt).

Während des Training minimieren wir somit die Summe aus der ursprünglichen Loss-funktion, welche u.a. eine Funktion von den Trainingsdaten $\{\boldsymbol{X}, \boldsymbol{y}\}$ ist und einem Zusatzterm, welcher ein Maß für die Größe der Modellparameter ist.

Parameter Norm Penalties

Bevor wir verschiedene Möglichkeiten für Ω betrachten...

Genereller Hinweis zu Parameter-Normen

In neuronalen Netzen werden üblicherweise nur hohe Einträge in den Gewichtsmatrizen W aber nicht in den Bias-Vektoren $\boldsymbol{b}$ durch Ω bestraft. Jedes Gewicht in W spezifiziert, wie 2 Variablen interagieren, sodass das Fitten viele Sample-Daten in verschiedenen Kontexten benötigt. Der Bias b hat jedoch nur Einfluss auf eine einzige Variable und benötigt deshalb üblicherweise deutlich weniger Daten zum Fitten. Eine Regularisierung des Bias kann deshalb zu under-fitting führen.

Manchmal möchte man in tiefen NN verschiedene α für die einzelnen Layers festlegen. Allerdings kann es aufwendig sein, diese Hyperparameter dann alle zu bestimmen sodass es oftmals sinnvoll ist, α einfach konstant für alle Layer zu setzen.

L^2 Parameter Regularisierung

Ein einfacher und häufiger Regularisierungsterm für die Parameter ist die L_2 Norm, welche manchmal auch als *Ridge Regression* or *Tikhonov Regularisierung* bezeichnet wird:

$$\Omega(\theta) = \frac{1}{2}\|\boldsymbol{w}\|^2 \Rightarrow \tilde{L}(\theta; \boldsymbol{X}, \boldsymbol{y}) = L(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha\frac{1}{2}\|\boldsymbol{w}\|^2$$

Dieser Typ der Regularisierung wird auch als *weight decay* bezeichnet, weil er die Gewichte auf 0 zieht, es sei denn der ursprüngliche Loss $L(\theta; \boldsymbol{X}, \boldsymbol{y})$ unterstützt andere Werte mit Daten.

Im Backpropagation-Algorithmus fügt dieser Penalty-term einfach ein $\alpha\boldsymbol{w}$ zu den jeweiligen Gradienten hinzu:

$$\nabla_{\boldsymbol{w}}\tilde{L}(\boldsymbol{w}; \boldsymbol{X}, by) = \nabla_{\boldsymbol{w}}L(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\boldsymbol{w}$$

Weight Decay

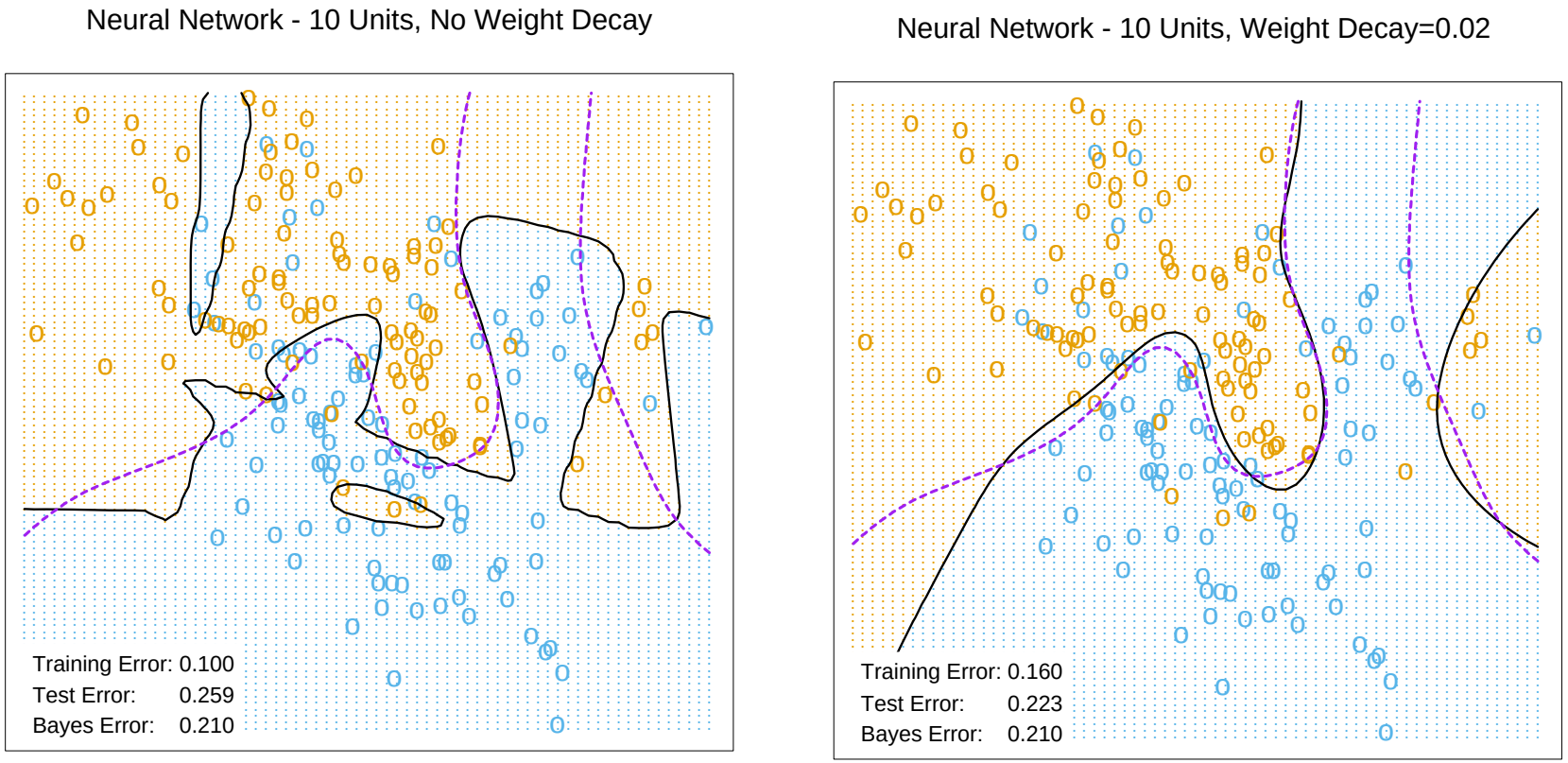


Figure: 11.4 von Hastie. Vergleich des Effekts von weight-decay auf over-fitting. (Links: Training ohne weight decay führt zu overfitting, Rechts: weight decay führt zu deutlich besseren Vorhersagen)

Weight Decay

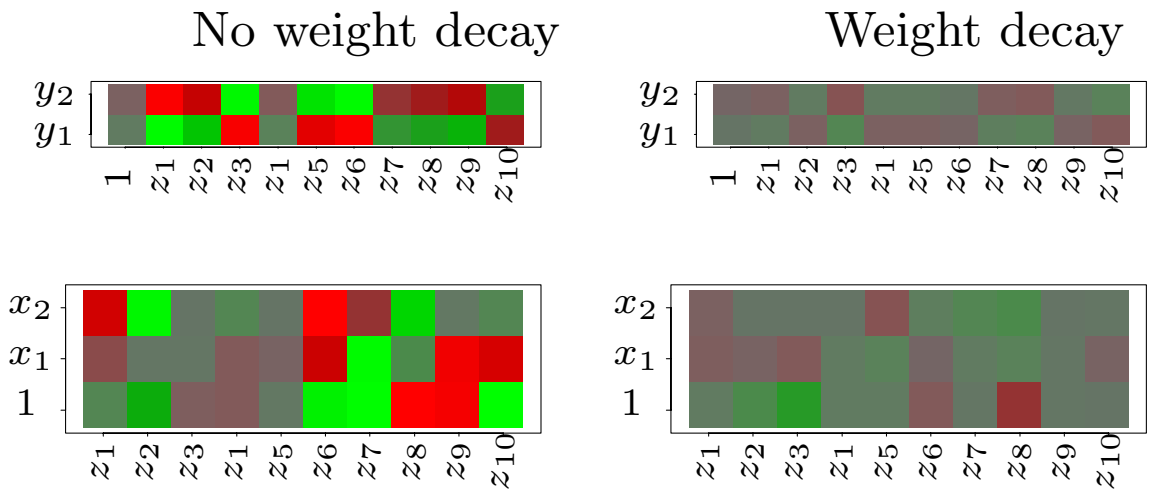


FIGURE 11.5. *Heat maps of the estimated weights from the training of neural networks from Figure 11.4. The display ranges from bright green (negative) to bright red (positive).*

Figure: von Hastie. Weight decay dämpft die Gewichte in beiden Layern und führt zu ausgeglichener verteilten Gewichten über die 10 Hidden units.

Intuition von Weight Decay

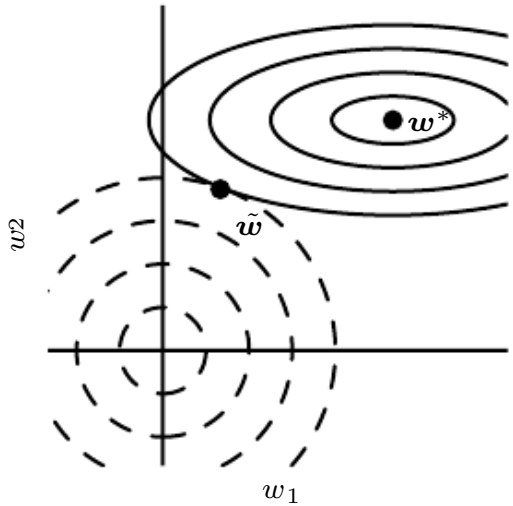


Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to zero. In the second dimension, the objective function is very sensitive to movements away from $\boldsymbol{w}^*$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of w_2 relatively little.

Figure: 7.1 von Goodfellow.

L^1 Regularisierung

Ein weiterer Regularisierungs-Term ist L_1, auch bekannt als *LASSO (least absolute shrinkage and selection operator)*. Die L_1 Regularisierung der Modellparameter W ist definiert als:

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|,$$

was der Summe der Absolutwerte der individuellen Parameter entspricht.

Die Loss-funktion mit Regularisierung ist nun:

$$\tilde{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = L(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \cdot \|\boldsymbol{w}\|_1$$

und der Gradient:

$$\nabla_{\boldsymbol{w}} \tilde{L}(\boldsymbol{w}; \boldsymbol{X}, by) = \nabla_{\boldsymbol{w}} L(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \cdot \mathsf{sign}(\boldsymbol{w})$$

Sparse Networks

Der Gradient der L_1-Regularisierung ($\alpha \cdot \text{sign}(\boldsymbol{w})$) skaliert nicht linear mit w_i wie bei L_2, sondern ist ein konstanter Faktor α mit dem selben Vorzeichen wie w_i. Dies kann dazu führen, dass einige Gewichte ≈ 0 gesetzt werden und die korrespondierenden Synapsen aus dem Netzwerk entfernt werden können ($\rightarrow$ sparse network).

Sparse Networks

In sparsen Netzwerken sind nicht alle hidden Units eines Layers mit allen Units des vorhergehenden Layers verknüpft. Dies spart Rechenarbeit und kann dargestellt werden, indem einzelne synaptische Gewichte gleich 0 gesetzt werden.

Representation Sparsity

Die L_1 Regularisierung induziert spärliche (sparse) Netzwerkgewichte (einige Einträge sind nahezu 0). Auf ähnliche Weise können wir auch Sparsity für die Repräsentationen in den hidden Layers des Netzwerkes erzwingen. Ein Vergleich der beiden Sparsity-Typen werden in folgendem Beispiel für lineare Regression gezeigt:

$$\underbrace{\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}}_{\boldsymbol{y} \in \mathbb{R}^m} = \underbrace{\begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}}_{\boldsymbol{A} \in \mathbb{R}^{m \times n}} \underbrace{\begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}}_{\boldsymbol{x} \in \mathbb{R}^n} \qquad \underbrace{\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}}_{\boldsymbol{y} \in \mathbb{R}^m} = \underbrace{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}}_{\boldsymbol{B} \in \mathbb{R}^{m \times n}} \underbrace{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}}_{\boldsymbol{h} \in \mathbb{R}^n}$$

Figure: Gewichts vs. Repräsentations Spärlichkeit, von Goodfellow

Für Repräsentations-Sparsity können wir einen L_1 Regularisierungs-Term für die Ausgaben h in den hidden Layern zum Standard-Loss hinzufügen:

$$\tilde{L}(\boldsymbol{\theta} = L(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \cdot \Omega(\boldsymbol{h}) \qquad \text{where} \qquad \Omega(\boldsymbol{h}) = \|\boldsymbol{h}\|_1 = \sum_i |h_i|,$$

Regularisierung als Maximum A Posteriori (MAP) - Schätzer

Viele Regularisierungsstrategien können als MAP-Schätzer interpretiert werden. Zum Beispiel ist die L_2-Regularisierung äquivalent zu einem MAP-Schätzer mit einem Normalverteilten Prior über die Gewichte.

Die L_1-Regularisierung ist äquivalent zu einem MAP-Schätzer mit einem Laplaceverteilten Prior über die Gewichte.

Part 4

Reduktion der Modellkapazität

Chapter 7.8, 7.9, 7.10 in **Goodfellow-et-al-2016-Book**.
Chapter 5.5.2 in **bishop2006pattern**

Anzahl von Hidden Units und Layern

Wie viele hidden Units sollten wir in einem Layer verwenden?

Zu viele hidden Units machen das Netz anfällig für Overfitting und zu wenige hidden Units können zu einer unzureichenden Modellkapazität führen. Wie viele hidden Units sollten wir also nehmen? Im Allgemeinen ist es besser, mit Regularisierungen zu trainieren, um Overfitting zu verhindern.

Wie viele Layer sollten wir im Netzwerk verwenden?

Die Anzahl an hidden Layern ist sehr anwendungsspezifisch. Mehrere hidden Layer erlauben uns hierarchische Features mit unterschiedlichen Auflösungen zu konstruieren. Das Verknüpfen vieler Layers ist ein Schlüsselfaktor für den Erfollg von Deep Learning und wird im Laufe der Vorlesung noch weiter besprochen.

Early Stopping

Grosse Netzwerke haben eine grosse Modellierungs-Kapazität (siehe universal Function Approximation Theorem von letzter Vorlesung) und sind deshalb gefährdet overzufitten. Overfitting kann beobachtet werden, wenn wir die Trainings- und Validation-Fehler während des Trainings vergleichen:

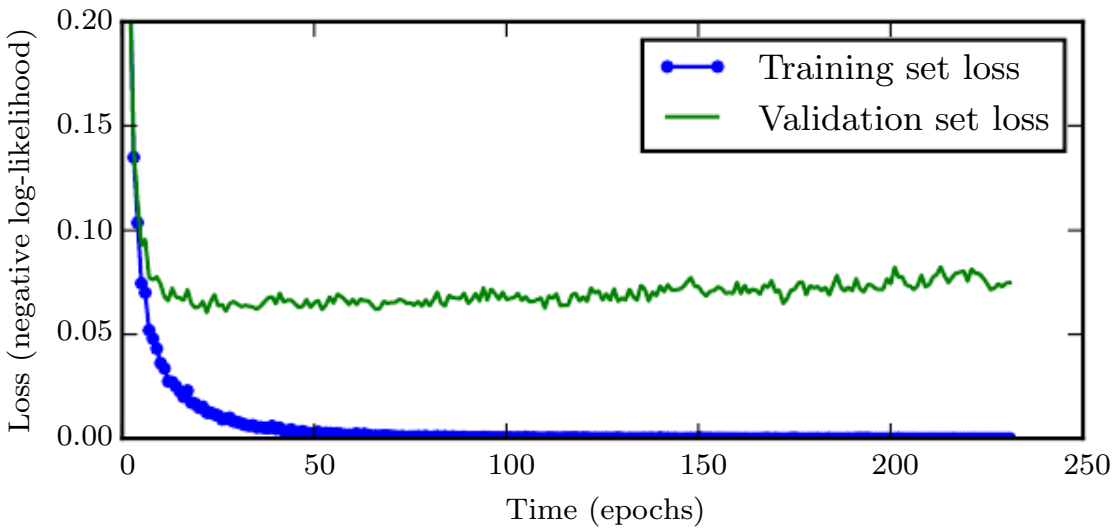


Figure: 7.3 von Goodfellow. Beachte, dass der Trainingsfehler während des Trainings weiter sinkt während der Validierungs-Fehler steigt, sobald der Lernprozess beginnt zu overfitten (hier ca. ab Epoche 30). Wenn wir annehmen, dass der Validierungs-Fehler dem Test-Fehler entspricht, dann sollten wir also die Modell-Parameter von Epoche 30 verwenden.

Early Stopping

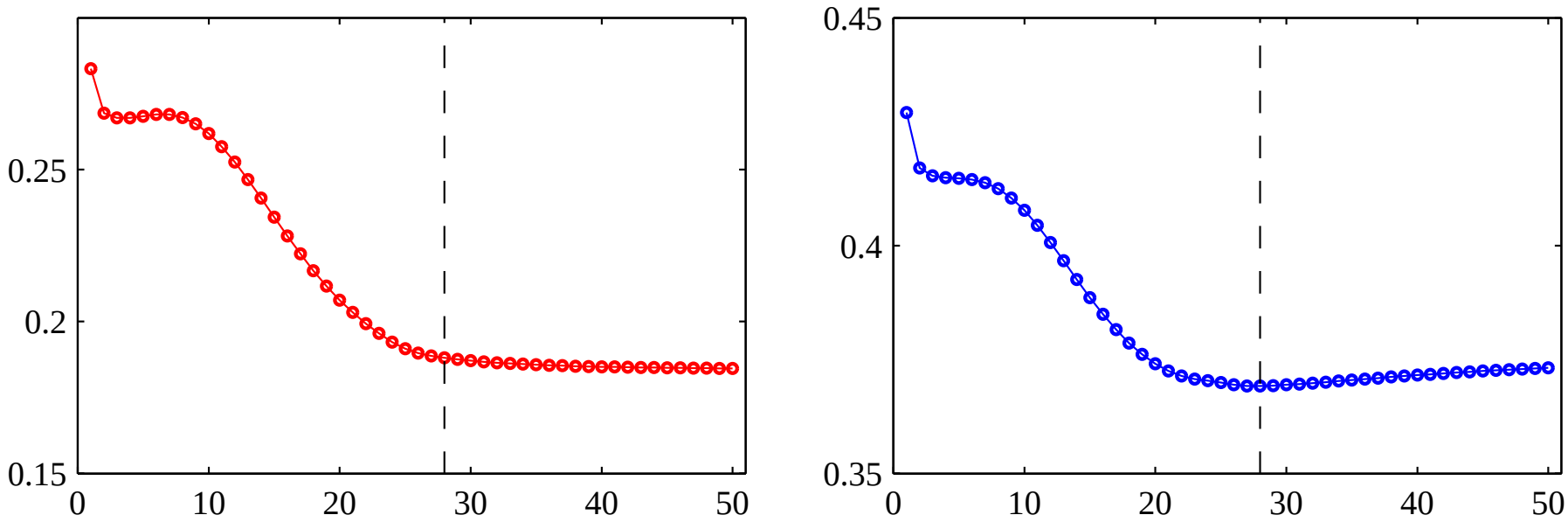


Figure: 5.12 von Bishop. Anschauliches Beispiel des Trainings-Fehlers (links) und des Validierungs-Fehlers (rechts) als Funktion der Trainingsiterationen. Wenn wir eine gute Generalisierungsperformance erzielen möchten sollten wir hier nach etwa 28 Epochen stoppen (siehe vertikale Linie).

Early Stopping

Die Early-Stopping Strategie kann als effiziente Hyperparameter-Auswahl aufgefasst werden um den Hyperparameter der Anzahl Trainingsschritte zu finden. Die meisten Hyperparameter, welche die Modellkapazität kontrollieren (inklusive der Anzahl Trainingsschritte) haben eine U-förmige Validierungs-Performance ähnlich zur vorherigen Abbildung.

Der Rechenaufwand dieser Early-Stopping Strategie besteht aus den zusätzlichen Evaluationen auf dem Validierungs-Set und dem Speichern der soweit besten Modell-Parameter. Um auch das Validierungs-Set beim Training auszunutzen, können wir das Modell nochmals inklusive dem Validierungs-Set trainieren und die Anzahl Iterationen verwenden, die wir mit der Early Stopping Methode erhalten haben. Alternativ können wir auch das Training noch etwas weiterführen und das Validierungs-Set mitverwenden.

Frage: Wenn wir sowohl mit dem Trainings- als auch dem Validierungsset trainieren - was könnte dann ein Problem mit der zuvor bestimmten Anzahl Optimierungsschritte sein?

Early Stopping als ein Regularisierer?

Man kann Early Stopping auch als eine Einschränkung des Optimierungsverfahrens auf ein relativ kleines Volumen im Parameterraum in der Nachbarschaft der initialen Parameterwerte betrachten. Wenn wir τ Optimierungsschritte mit learning rate ϵ nehmen, dann ist das Produkt $\epsilon \cdot \tau$ ein Mass für die effektive Modellkapazität (gegeben, dass sowohl ϵ als auch τ beschränkt sind).

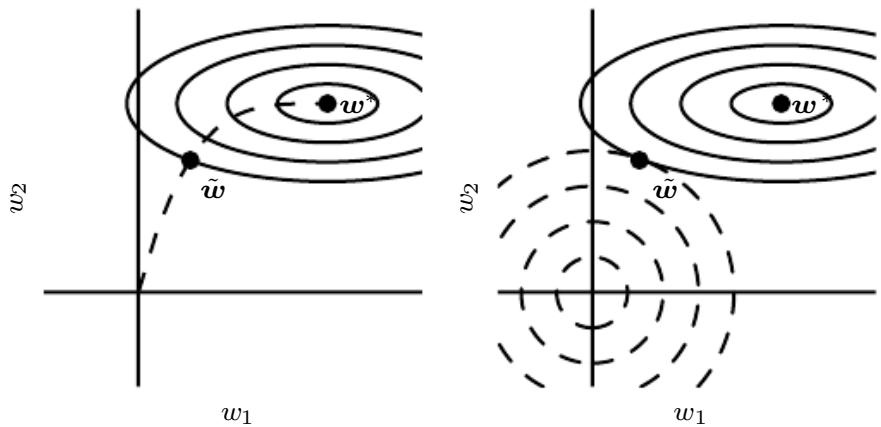


Figure 7.4: An illustration of the effect of early stopping. *(Left)*The solid contour lines indicate the contours of the negative log-likelihood. The dashed line indicates the trajectory taken by SGD beginning from the origin. Rather than stopping at the point $\boldsymbol{w}^*$ that minimizes the cost, early stopping results in the trajectory stopping at an earlier point $\tilde{\boldsymbol{w}}$. *(Right)*An illustration of the effect of L^2 regularization for comparison. The dashed circles indicate the contours of the L^2 penalty, which causes the minimum of the total cost to lie nearer the origin than the minimum of the unregularized cost.

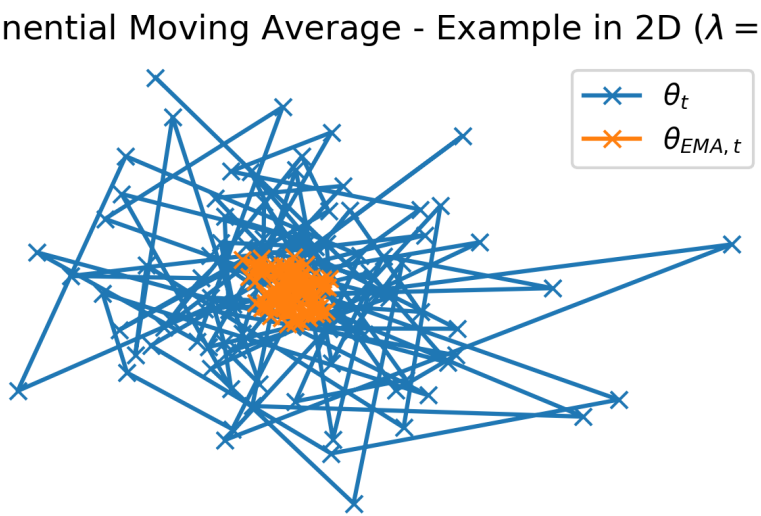
Figure: von Goodfellow.

Exponential Moving Average

Eine weitere Methode, um stabilere Resultate bei stochastischen Gradientenabstiegsverfahren zu erhalten, ist, einen Exponential Moving Average (EMA) θ_{EMA} der Netzwerkparameter θ während des Trainings zu berechnen:

$$\theta_{EMA,t+1} = (1 - \lambda)\theta_{EMA,t} + \lambda\theta_t$$

Hierbei ist λ ein Hyperparameter, der beschreibt, wie stark θ_{EMA} in jedem Schritt geupdatet wird. Er wird auch als Momentum bezeichnet und filtert Rauschen aus den Modellupdates. Beachte, dass θ_{EMA} nur während der Validierung und Tests verwendet wird, jedoch ansonsten keinen direkten Einfluss auf das Training hat. Weitere Infos zur Implementierung in Pytorch gibt es [hier].



Parameter Sharing

Die Einschränkung von Parametern, die von einem festen Wert abweichen (z.B. 0 bei der L_2 oder L_1 Regularisierung), setzt voraus, dass wir diese festen Werte kennen, was nicht immer der Fall ist.

Basierend auf Domänenwissen und der Modellarchitektur können wir manchmal weitere Priors und Abhängigkeiten formulieren.

Betrachte zum Beispiel ein Modell A mit Parametern $\boldsymbol{\theta}_A$ und ein Modell mit Parametern $\boldsymbol{\theta}_B$, wobei beide Modelle die gleiche Klassifizierungsaufgabe lösen sollen, jedoch unterschiedliche Eingabeverteilungen haben. Wenn diese Aufgaben ähnlich genug sind, können wir annehmen, dass $\boldsymbol{\theta}_A$ nahe bei $\boldsymbol{\theta}_B$ liegen müsste und einen Parameter-Penalty der Form: $\Omega(\boldsymbol{\theta}_A, \boldsymbol{\theta}_B) = \|\boldsymbol{\theta}_A - \boldsymbol{\theta}_B\|_2^2$ anwenden.

Parameter Sharing

Ein direkterer (und populärerer) Ansatz ist, die Gleichheit der Parameter durch *Parameter sharing* (manchmal auch *Weight sharing* oder *Parameter tying*) zu erzwingen. Verglichen mit der vorherigen Norm penalty hat dies den Vorteil, dass nur ein Satz an Parametern im Speicher gehalten werden muss.

Convolutional Neural Networks (CNNs - mehr dazu nächste Woche) sind eine der am weitesten Verbreiteten Anwendungen von Parameter sharing. In Bilderkennungsaufgaben möchten wir z.B. Orts-Invarianz haben [1]. CNNs handhaben diese Invarianz indem sie die Netzwerkgewichte an vielen Orten im Bild teilen. Ein weitere Vorteil von Weight sharing ist, dass die Netzwerke wesentlich kleiner werden und mit viel weniger Daten trainiert werden können.

[1] Wenn wir z.B. eine Katze Klassifizieren möchten, dann spielt der Ort der Katze im Bild keine Rolle

Part 5

Trainings-Daten

Chapter 7.4, 7.5, 7.11 **Goodfellow-et-al-2016-Book**.
Chapter 5.5.3 of **bishop2006pattern**

Data Augmentation

Um Invarianz (z.B. bezüglich Translationen / Rotationen / Rauschen / etc) in ein Machine Learning Modell (z.B. ein NN) zu integrieren, können wir Trainingsdaten mit besagten Invarianzen verwenden. Da das Generieren solcher Trainingsdaten speicherintensiv ist, können wir stattdessen auch neue Daten on-the-fly synthetisieren, indem wir die ursprünglichen Daten transformieren.

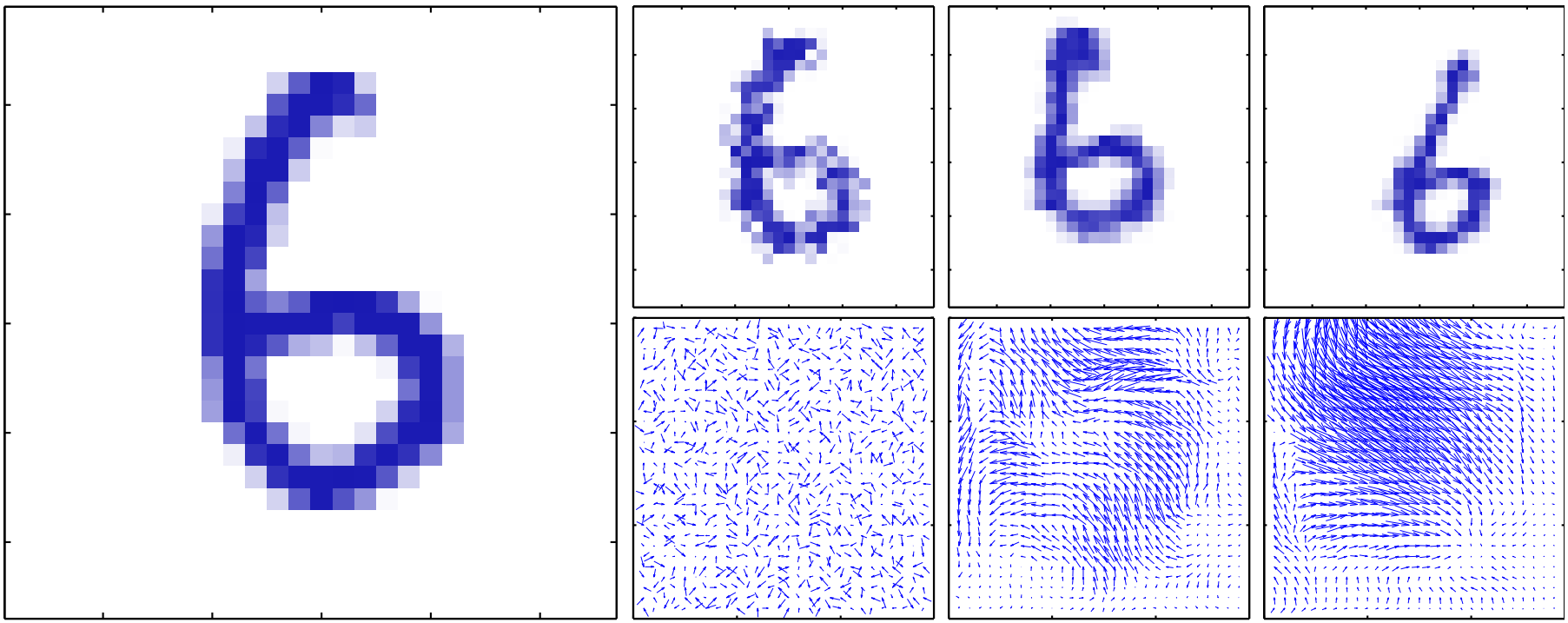


Figure: 5.14 von Bishop; Drei synthetisch gewarpte Bilder von Ziffern wurden durch Sampling mit einem zufälligen Displacement-Field generiert.

Data Augmentation

Die Daten-Synthese ist sehr anwendungsabhängig!

Augmenting Visual Data

In Bilderkennungsaufgaben funktioniert Data augmentation gut, da viele Bildtransformationen einfach synthetisiert werden können. Zum Beispiel verbessern kleine zufällige Translationen / Rotationen / Skalierungen oftmals deutlich die Generalisierungsperformanz. Dabei muss man jedoch aufpassen, dass solche Transformationen nicht das ground truth Klassenlabel ändern.

Frage: Welche Transformationen können wir nicht in einem Ziffernerkennungsproblem verwenden?

Injecting Noise

In vielen Problemstellungen möchten wir eine Aufgabe lösen, auch wenn etwas zufälliges Rauschen (Noise) in den Eingabedaten vorhanden ist. Neuronale Netze sind typischerweise nicht sehr robust gegen Noise, wenn keine zusätzlichen Maßnahmen dagegen unternommen werden.

Noisy Inputs

Eine Möglichkeit um die Robustheit eines NN zu verbessern ist, die Trainingsdaten mit Rauschen zu augmentieren. Man kann auch den hidden units Rauschen hinzufügen, was als Data Augmentation auf mehreren Abstraktionsebenen interpretiert werden kann.

Noisy Weights

Rauschen kann auch den Netzwerkgewichten hinzugefügt werden. In diesem Fall kann man die Modellgewichte als Zufallsvariablen betrachten.

Noisy Outputs

Datensets können Fehler in den Ground-Truth labels haben. Solche Fehler können einen schädlichen Einfluss haben, wenn wir $-\log p(y|\boldsymbol{x})$ minimieren möchten ohne solche Fehler explizit zu modellieren.

Um solche Fehler zu modellieren können wir annehmen, dass ein Trainingslabel lediglich mit einer Wahrscheinlichkeit $1-\epsilon$ korrekt ist, wobei ϵ ein kleiner konstanter Wert ist. Diese Annahme kann direkt in die Kostenfunktion aufgenommen werden. Wenn wir z.B. eine Klassifizierungsaufgabe mit k Klassenlabels haben, dann verwenden wir nicht 1 und 0 für korrekte und inkorrekte labels sondern $1-\epsilon$ bzw. $\frac{\epsilon}{k-1}$.

Part 6

Bagging und Dropout

Kapitel 7.12 in **Goodfellow-et-al-2016-Book**.

Bagging

Bagging (von **B**ootstrap **agg**regat**ing**) kann den Generalisierungsfehler reduzieren indem ein Ensemble von Modellen kombiniert wird. Dabei werden mehrere Modelle separat trainiert und jedes Modell stimmt dann über die Ausgabe zu einem Test-Sample (gleichberechtigt) ab.
Nehmen wir an, dass jedes Modell i eine Wahrscheinlichkeitsfunktion $p_i(y|\boldsymbol{x})$ produziert, dann entspricht die Vorhersage des Ensembles dem Mittelwert dieser Wahrscheinlichkeitsfunktionen:

$$p_{\text{bagging}} = \frac{1}{k} \sum_{i}^{k} p_i(y|\boldsymbol{x}).$$

Erwarteter Fehler von Bagging

Betrachte k Regressoren, wobei jedes Modell einen Fehler ϵ_i pro Test-Sample macht. Wenn diese ϵ_i aus zentrierten Normalverteilungen mit Varianzen $\mathbb{E}[\epsilon_i^2] = v$ und Kovarianzen $\mathbb{E}[\epsilon_i \epsilon_j] = c$ gezogen werden, dann ergibt sich für den erwarteten quadrierten Fehler der Mittelwert-Vorhersage des Ensembles $1/k \sum_i \epsilon_i$:

$$\mathbb{E}\Big[\big(\frac{1}{k}\sum_i \epsilon_i\big)^2\Big] = \frac{1}{k^2}\mathbb{E}\Big[\sum_i \big(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j\big)\Big] = \frac{1}{k}v + \frac{k-1}{k}c.$$

Wenn die einzelnen Fehler ϵ_i komplett korreliert sind $(c = v)$, dann reduziert sich der MSE des ensembles zu v. Wenn die Fehler jedoch unkorreliert sind $(c = 0)$, dann ist der MSE nur noch $\frac{1}{k}v$. Dies bedeutet, dass das Ensemble im Mittel mindestens so gut wie einzelne Modelle performt und deutlich besser, wenn die einzelnen Modelle unkorrelierte Fehler machen.

Bagging

Wie sollten wir dieses Ensemble von Modellen erstellen?

Eine Standardpraxis beim Bagging ist, k "*unterschiedliche*" Datensets der Größe N zu erstellen, indem Samples aus dem ursprünglichen Datenset mit Zurücklegen gezogen werden. Alle k Modelle werden dann auf ihren jeweiligen Datensets auf identische Art und Weise trainiert.

Frage: Wie unterscheiden sich die k gesampleten Datensets vom urspünglichen Datenset?

Model Averaging in Neuronalen Netzwerken

Beim Training von Neuronalen Netzen können eine Vielzahl von Lösungen gefunden werden , weshalb Bagging auch funktionieren kann, wenn das Training auf dem selben Datenset stattfindet. Unterschiede in der zufälligen Initialisierung der Gewichte, Mini-Batches, Hyperparametern und weiteren nicht-deterministischen Einflüssen können zu unterschiedlichen Modell-Resultaten führen. Bei großen NN kann Bagging jedoch sehr rechenaufwändig werden.

Bagging

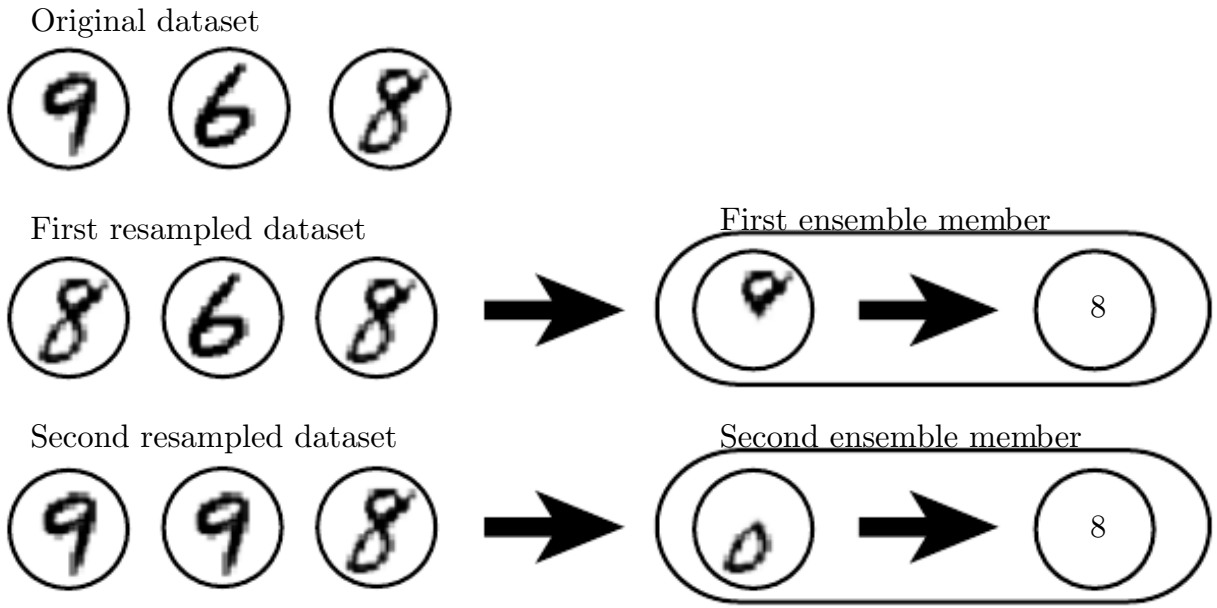


Figure 7.5: A cartoon depiction of how bagging works. Suppose we train an 8 detector on the dataset depicted above, containing an 8, a 6 and a 9. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. The first dataset omits the 9 and repeats the 8. On this dataset, the detector learns that a loop on top of the digit corresponds to an 8. On the second dataset, we repeat the 9 and omit the 6. In this case, the detector learns that a loop on the bottom of the digit corresponds to an 8. Each of these individual classification rules is brittle, but if we average their output then the detector is robust, achieving maximal confidence only when both loops of the 8 are present.

Figure: 7.5 von Goodfellow.

Dropout

Eine vom Rechenaufwand her effiziente Annäherung an Bagging und dessen regularisierenden Effekt für Neuronale Netze ist *Dropout*. In Bagging werden k unterschiedliche Modelle auf k unterschiedlichen Datensets trainiert. Dropout nähert diesen Prozess an, jedoch mit einer exponentiell großen Anzahl von Modellen.

Dropout trainiert ein Ensemble aller möglichen Sub-Netzwerke, welche durch zufälliges Ausschalten von Eingaben / Hidden-Units eines ursprünglichen Netzwerkes erstellt werden können. Eine Unit wird dabei "ausgeschaltet", indem ihre Ausgabe mit 0 multipliziert wird.

Dropout

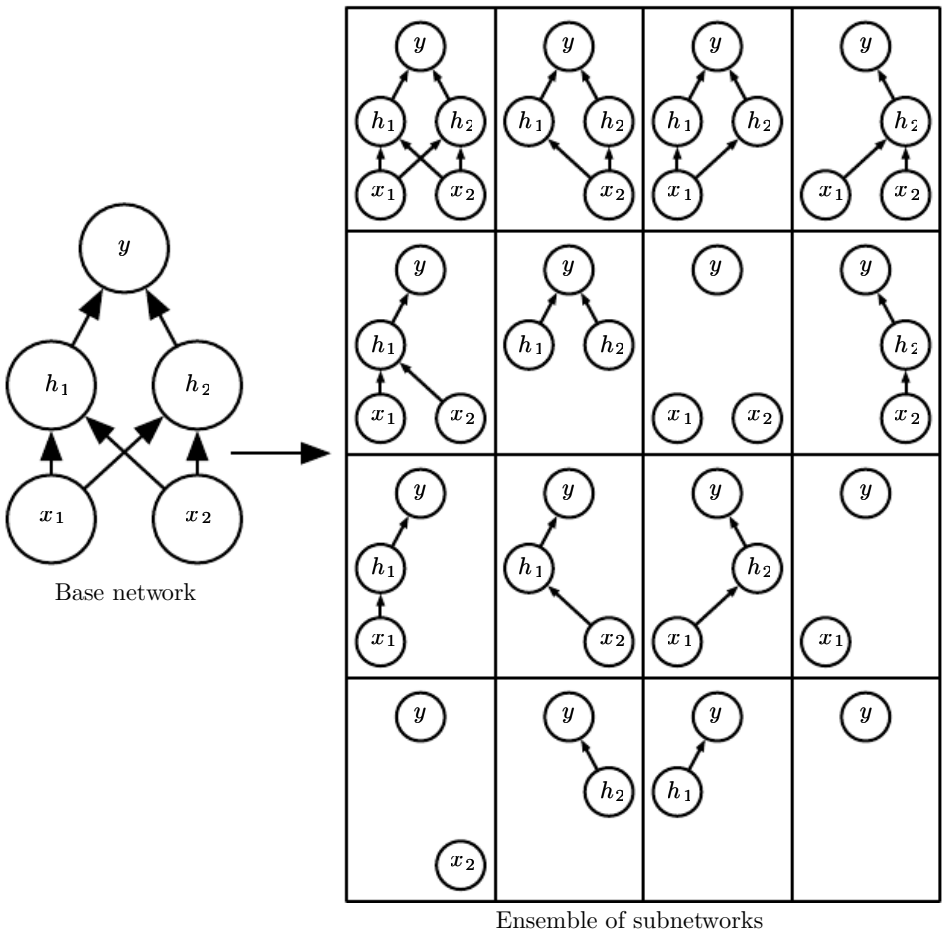


Figure: aus Deep Learning. $2^4 = 16$ Sub-Netzwerke können erstellt werden, indem einzelne Eingaben / Hidden-Units ausgeschaltet werden. Ein Grossteil dieser Netzwerke macht in diesem Beispiel keinen Sinn, da die Eingaben nicht mit der Ausgabe verbunden sind; in größeren Netzwerken ist dies jedoch üblicherweise kein Problem.

Dropout

Randomly Sampled Unit Masks

Um mit Dropout zu trainieren, können wir wieder einen Mini-batch basierten Optimierungsalgorithmus wie z.B. SGD verwenden. Für jedes Sample eines Mini-batches ziehen wir dabei unabhängig eine zufällige Binärmaske welche an die Eingaben / Hidden-Units multipliziert wird. Die Wahrscheinlichkeit, dass ein Eintrag der Binärmaske 1 ist, wird duch einen Hyperparameter bestimmt[2]. Danach wird das Trainingsupdate wie üblich fortgesetzt.

Formal können wir sagen, wenn $\boldsymbol{\mu}$ die Binärmaske spezifiziert und $L(\boldsymbol{\theta}, \boldsymbol{\mu})$ die Modellkosten (Loss) definiert, dann minimiert das Dropout-Training $\mathbb{E}_{\boldsymbol{\mu}} L(\boldsymbol{\theta}, \boldsymbol{\mu})$.

[2] Typischerweise 0.8 für Eingabewerte und 0.5 für Hidden-Units

Inferenz in Dropout

Beim Bagging werden die Stimmen von einzelnen Modellen eines Ensembles akkumuliert. Ähnlich dazu wird in Dropout die produzierte Wahrscheinlichkeitsverteilung $p(y|\boldsymbol{x}, \boldsymbol{\mu})$ jedes Sub-Modells durch eine Binärmaske $\boldsymbol{\mu}$ definiert und wir müssen den Mittelwert über alle $\boldsymbol{\mu}$ nehmen:

$$p_{\text{dropout}} = \sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y|\boldsymbol{x}, \boldsymbol{\mu}),$$

wobei $p(\boldsymbol{\mu})$ der Wahrscheinlichkeitsverteilung entspricht, eine Maske $\boldsymbol{\mu}$ während des Trainings zu samplen. Diese Summe kann jedoch nicht explizit berechnet werden, da sie eine exponentielle Anzahl Terme enthält. Eine Option ist, die Summe zu samplen (z.B. reichen 10-20 zufällig gezogene Masken oft bereits aus, um die Summe gut anzunähern)

Weight Scaling Inferenz-Regel

p_{ensemble} kann auch approximiert werden, indem wir $p(y|\boldsymbol{x})$ mit dem vollen Modell auswerten. In diesem Fall müssen allerdings die Gewichte mit (1-"Dropout-Wahrscheinlichkeit") skaliert werden. Wenn die Dropout-Wahrscheinichkeit z.B. 0.2 ist, dann sollten die Gewichte des vollen Modells für die Inferenz mit 0.8 skaliert werden.

Bagging vs. Dropout

Unabhängige Parameter vs. Parameter Sharing

Bei Bagging werden alle Modelle unabhängig trainiert, wohingegen bei Dropout die einzelnen Sub-Netzwerke ihre Parameter teilen. Dieses Parameter sharing erlaubt es, mit Dropout eine exponentielle Anzahl an Modellen mit einer handhabbaren Anzahl Parametern zu repräsentieren.

Explizites vs. "Implizites" Training

In Bagging wird jedes Modell bis zur Konvergenz auf dem jeweiligen Trainingsset trainiert. Bei Dropout, werden die meisten Modelle nicht explizit trainiert, da es unmöglich ist alle (exponentiell vielen) Sub-Netzwerke zu samplen. Stattdessen wird nur ein kleiner Bruchteil der möglichen Sub-Netzwerke für einen einzelnen Schritt trainiert; die verbleibenden Sub-Netzwerke werden durch Parameter-Sharing optimiert.

Ähnlichkeiten

Abgesehen von diesen Differenzen folgt Dropout dem Bagging-Algorithmus, z.B. können auch die Trainingsets der einzelnen Sub-Netzwerke als Subsets des originalen Trainings-Sets (gezogen mit Zurücklegen) betrachtet werden.

Effektivität von Dropout

Experimentell zeigt Dropout sehr gute Regularisierungseigenschaften und kann mit weiteren Strategien (z.B. weight decay) für eine noch bessere Performance kombiniert werden.

Günstiger Rechenaufwand

Während des Trainings benötigt Dropout nur $O(n)$ berechnungsschritte pro Trainingssample um die Binärmaske $\boldsymbol{\mu}$ zu berechnen und an die Gewichte zu multiplizieren. Je nach Implementierung benötigt Dropout $O(n)$ Speicher um die Binärmaske für den Backpropagation algorithmus zwischenzuspeichern. Während der Inferenz hat Dropout keinen Einfluss auf den Rechenaufwand, da das Weight-Scaling nur einmal auf die Modellgewichte angewendet werden muss.

Effektivität von Dropout

Model-Agnostic

Im Gegensatz zu vielen anderen Regularisierungsstrategien ist Dropout sehr effektiv für eine Vielzahl an Modellen, inklusive feedforward NN, probabilistischen Modellen und rekurrenten NN.

Dropout benötigt größere Modelle

Da Dropout die effektive Kapatzität eines Modells reduziert muss man die Größe des Netzwerkes (Anzahl hidden Units pro Layer) vergrößern um den Effekt zu kompensieren. Wir können wesentlich kleinere Validierungsfehler mit Dropout erzielen, müssen dies aber mit einem größeren Modell und mehr Trainingsiterationen kompensieren.

Alternative Sichtweise auf den Dropout-Regularisierer

Dropout trainiert ein Ensemble von Modellen mit Parameter-Sharing in den hidden Units. Dies bedeutet, dass jede hidden Unit gut funktionieren muss, auch wenn andere Units ausgeschaltet sind. Dadurch erzwingt Dropout, dass eine hidden Unit nicht nur ein einzelnes Feature lernt sondern eine Vielzahl an Features repräsentiert um die fehlenden Informationen anderer ausgeschalteter Units auszugleichen.

Dropout vs. Noise-Augmentation der Hidden Units

Dropout kann auch als eine Form der intelligenten adaptiven Entfernung von Informationen betrachtet werden, anstatt die Werte der Eingabe / hidden Units mit Noise zu augmentieren.

Wenn zum Beispiel eine hidden Unit h_i des Modells trainiert wird, um die Nase eines Gesichts zu erkennen, dann entspricht das Weglassen von h_i einer Entfernung dieser Information über die Nase, so dass das Modell ein anderes h_i lernen muss, das entweder die Nase redundant kodiert oder das Gesicht durch ein anderes Merkmal (z.B. den Mund) erkennt.

Herkömmliche Noise-Augmentation-Strategien, die den Inputs unstrukturiertes Rauschen hinzufügen, könnten die Informationen über eine Nase in einem Gesicht nicht zufällig löschen, es sei denn, das Rauschen ist so stark, dass fast alle Bildinformationen entfernt werden.

Durch Dropout wird das Modell gezwungen alle Informationen der hidden Units effektiv zu nutzen, da einzelne Units ausgeschlatet werden können.

../deeplearn