Aprendizaje Profundo

Maestría en Modelación y Optimización de Procesos
CIMAT-Aguascalientes

Dra. Lilí Guadarrama Bustos[1]
Dr. Isidro Gómez-Vargas[2]

[1]Cátedra CONACyT CIMAT-Aguascalientes
[2]Investigador posdoctoral en ICF-UNAM

Clase del semestre enero-julio 2022

Contenido

1. Bases de Aprendizaje Automático
2. Redes de propagación hacia adelante
3. Regularización
4. Optimización para modelos de entrenamiento profundo
5. Redes neuronales convolucionales
6. Redes neuronales recurrentes y recursivas

Aprendizaje profundo

Bibliografía

Bibliografía principal

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Diapositivas:

- `https://www.deeplearningbook.org/lecture_slides.html`
- `https://github.com/InfolabAI/DeepLearning`

Yoshua Bengio in 2019

Born	March 5, 1964 (age 57) Paris, France
Citizenship	Canada
Alma mater	McGill University
Known for	Deep learning, neural machine translation, generative adversarial networks, "attention model", word embeddings, denoising auto-encoders, neural language models, learning to learn
Awards	Marie-Victorin Prize (2017) Turing Award (2018) AAAI Fellow (2019)
Scientific career	

Born	1985/1986 (age 35–36)
Nationality	American
Alma mater	Stanford University Université de Montréal
Known for	Generative adversarial networks, Adversarial Examples
Scientific career	
Fields	Computer science
Institutions	Apple Inc. Google Brain OpenAI
Thesis	*Deep Learning of Representations and its Application to Computer Vision* (2014)
Doctoral advisor	Yoshua Bengio Aaron Courville
Website	www.iangoodfellow.com

Aaron Courville

Université de Montréal
Dirección de correo verificada de umontreal.ca - Página pr
Machine learning Artificial Intelligence

TÍTULO	CITADO POR
Generative adversarial nets I Goodfellow, J Pouget-Abadie, M Mirza, B Xu, D Warde-Farley, S Ozair, ... Advances in neural information processing systems 27	40148
Deep learning IJ Goodfellow, Y Bengio, A Courville Nature	36121
Representation learning: A review and new perspectives Y Bengio, A Courville, P Vincent IEEE transactions on pattern analysis and machine intelligence 35 (8), 1798-1828	10333

Temario

- Se abordarán los capítulos 5, 6, 7, 8, 9 y 10.
- Opcionales: capítulos 11 y 12.
- Temas elegidos por el grupo de la parte III del libro (capítulos 13-20).

Temas de la parte 1 del curso

1. Bases de Aprendizaje Automático
   - Algoritmos de aprendizaje.
   - Capacidad, subajuste y sobreajuste
   - Hiperparámetros y conjuntos de validación.
   - Estimadores, sesgo y varianza.
   - Estimación de Máxima Verosimilitud(MLE).
   - Estadística Bayesiana.
   - Algoritmos de aprendizaje supervisado.
   - Descenso del gradiente estocástico.
   - Desafios actuales Aprendizaje Profundo.
2. Redes de propagación hacia adelante
3. Regularización
4. Optimización para modelos de entrenamiento profundo

Algoritmos de aprendizaje.

> “Un programa de computadora se dice que aprende de la experiencia E con respecto a cierta clase de tareas T y con medida de rendimiento P, si su rendimiento en las tareas en T, medido por P, mejora con la experiencia E.”

Tom Mitchell, 1997

Glosario

- **Ejemplo**. $x \in \mathbf{R}^n$, colección de n características.
- **Conjunto de datos**. Colección de ejemplos.
- **Características**. Atributos.
- **Matriz de datos**.

Tareas T

- Clasificación
- Regresión
- Transcripción
- Traducción
- Salidas estructuradas
- Detección de anomalias
- Síntesis y muestreo
- Imputing valores eprdidos
- Quitar ruido
- Estimación de densidad o probabilidad

La medida de rendimiento P

- Exactitud $\frac{TP+TN}{TP+TN+FP+FN}$
- Error cuadrático medio $MSE = \frac{1}{n}\sum_i^n ||\hat{y}_i - y_i||$
- Densidad de probabilidad
- etc

La experiencia, E

Fuente: analystprep.com/study-notes

La experiencia, E

Fuente: medium.com/@recrosoft.io

Ejemplo: regresión lineal

Ver páginas 107-108 del libro.

Capacidad, subajuste y sobreajuste

- Algo que separa al aprendizaje automático de la optimización es el uso de un error de generalización o error de prueba (*test error*).
- Hay que minimizar tanto el error de entrenamiento como el error de generalización.
- La brecha entre ambos errores debe ser pequeña.

Capacidad, subajuste y sobreajuste

Capacidad, subajuste y sobreajuste

Capacidad, subajuste y sobreajuste: No free lunch theorem

Capacidad, subajuste y sobreajuste: No free lunch theorem

"No Free Lunch" :(

D. H. Wolpert. The supervised learning no-free-lunch theorems. In Soft Computing and Industry, pages 25–42. Springer, 2002.

Our model is a simplification of reality

⇩

Simplification is based on assumptions (model bias)

⇩

Assumptions fail in certain situations

Roughly speaking:

"No one model works best for all possible situations."

Fuente: analyticsindiamag.com

Capacidad, subajuste y sobreajuste: Regularización

Por ejemplo, decaimiento del peso:

$$J(w) = MSE + \lambda \omega^T \omega$$

Def.

Regularización es cualquier mecanismo que ayuda al agoritmo de aprendizaje a recudir su error de generalización.

Hiperparámetros y conjuntos de validación.

Fuente: kdnuggets.com

Hiperparámetros y conjuntos de validación: validación cruzada

Fuente: `https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada`

Estimadores, sesgo y varianza.

Scott Fortmann-Roe, Understanding the Bias-Variance Tradeoff, 2012

Estimadores, sesgo y varianza.

Estimadores, sesgo y varianza.

Visitar los siguientes enlaces:

- http://scott.fortmann-roe.com/docs/BiasVariance.html
- https://ml.berkeley.edu/blog/posts/crash-course/part-4/
- https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-per

Estimación de Máxima Verosimilitud(MLE).

- Estimación del Máximo Likelihood (MLE):

$$\ln \mathcal{L}(D, \theta) = \sum_{i=1}^{n} \ln f(x_i; \theta),$$

$$\theta_{MLE} = arg\ max(\mathcal{L}(\theta, D))$$

Teorema de Bayes

Considerando funciones de densidad de probabilidad:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}, \tag{1}$$

donde:

$$P(D) = \int_{\mathbb{R}^N} P(D|\theta)P(\theta)d\theta, \tag{2}$$

Estadística Bayesiana.

- Estimación del A Posteriori (MAP) ó estimación de parámetros ó inferencia Bayesiana:

$$\theta_{MAP} = arg\ max(\mathcal{L}(\theta, D)P(\theta))$$

Estadística Bayesiana.

- Estimación del A Posteriori (MAP) ó estimación de parámetros ó inferencia Bayesiana:

$$\theta_{MAP} = arg\ max(\mathcal{L}(\theta, D)P(\theta))$$

- Comparación de modelos (puede ser parte de la inferencia Bayesiana).

Estadística frecuentista vs Bayesiana.

Fuente: laptrinhx.com

Estadística frecuentista vs Bayesiana.

Suppose we have data $\mathcal{D} = \{x^{(i)}\}_{i=1}^{N}$

Maximum Likelihood Estimate (MLE)

$$\boldsymbol{\theta}^{\text{MLE}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{N} p(\mathbf{x}^{(i)}|\boldsymbol{\theta})$$

Maximum *a posteriori* (MAP) estimate

$$\boldsymbol{\theta}^{\text{MAP}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{N} p(\mathbf{x}^{(i)}|\boldsymbol{\theta})\underbrace{p(\boldsymbol{\theta})}_{\text{Prior}}$$

Fuente: medium.com/@tzjy

Algoritmos de aprendizaje supervisado.

Fuente: arXiv:2003.10146

Algoritmos de aprendizaje no supervisado.

Fuente: `arXiv:2003.10146`

Descenso del gradiente estocástico.

Fuente: slideshare.net/microlife

Descenso del gradiente estocástico.

Tarea propuesta

Programar un descenso del gradiente estocástico y minimizar una función con él.

Retos que motivan al Aprendizaje Profundo.

- Curso de la dimensionalidad.
- Local Constancy and Smoothness Regularization.
- Manifold Learning.

Temas de la parte 2 del curso

1. Bases de Aprendizaje Automático
2. Redes de propagación hacia adelante
   - 6.1 Ejemplo: Aprendiendo XOR
   - Gradient-based learning
   - Unidades ocultas
   - Diseño de arquitectura
   - Backpropagation y otros algoritmos de diferenciación
3. Regularización
4. Optimización para modelos de entrenamiento profundo
5. Redes neuronales convolucionales
6. Redes neuronales recurrentes y recursivas

Introducción al Aprendizaje Profundo

- Inicia la parte II del libro de referencia.
- Nos centraremos en el capítulo 6.

Perceptrón multicapa

Deep feedforward networks /feedforward neural networks

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

$f^{(i)}$ es llamada la i-ésima pared de la red. $f(x)$ debe coincidir con $f^*(x)$, cada ejemplo x tiene asociada una etiqueta $y = f^*(x)$

Perceptrón multicapa

Para que obtener un algoritmo no-lineal, hay que emplear una función $\phi(x)$ para modificar las entradas la las funciones.

Truco del kernel (descrito en 5.7.2)

$$w^T x + b = b \sum_{i=1}^{m} \alpha_i x^T x^{(i)}$$

se puede cambiar $x->\phi(x)$ y el producto punto por un kernel $k(x, x^{(i)}) = \phi(x)\dot{\phi}(x^{(i)})$. Entonces se pueden hacer predicciones mediante:

$$f(x) = b + \sum_{i=1}^{m} \alpha_i k(x, x^{(i)})$$

donde la función es no-lineal respecto a x, pero la relación entre $\phi(x)$ y $f(x)$ es lineal. También entre α y $f(x)$ es lineal.

Truco del kernel (descrito en 5.7.2)

La función basada en kernel es exactamente equivalente a preprocesar los datos aplicando $\phi(x)$ a todas las entradas, y luego aprender un modelo lineal en el nuevo espacio transformado.
Ventajas:

1. Permite aprender modelos que son funciones no lineales de x usando técnicas de optimización convexas.
2. La función kernel, generalmente, permite una implementación que es significativamente más eficiente, en términos computacionales, que construir dos vectores $\phi(x)$ y luego tomar explícitamente sus productos.

Funciones de activación

Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$

Leaky ReLU

$\max(0.1x, x)$

tanh

$\tanh(x)$

ReLU

$\max(0, x)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Funciones de activación

Multilayer Feedforward Networks are Universal Approximators

KURT HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE

University of California, San Diego

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Abstract—*This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.*

Keywords—Feedforward networks, Universal approximation, Mapping networks, Network representation capability, Stone-Weierstrass Theorem, Squashing functions, Sigma-Pi networks, Back-propagation networks.

Múltiples arquitecturas

Cronología ML

Aprendizaje máquina

1940s

El problema de la compuerta XOR

OR

XOR

Ver notebooks: https://github.com/igomezv/DLCIMATAGS/tree/main/notebooks/clase%202

Gradient-based learning

Como preámbulo: repasar secciones 4.3 y 5.9, correspondientes al aprendizaje por medio del descenso del gradiente y al descenso del gradiente estocástico, respectivamente.

Aprendizaje basado en descenso del gradiente

Optimización:

Maximizar o minimizar una función $f(x)$.

- La función a maximizar o minimizar se conoce como *función objetivo* o *criterio*.
- Cuando esta función se está minimizando, se le suele nombrar *función de costo*, *función de pérdida* o *función de error.*

Aprendizaje basado en descenso del gradiente

Aprendizaje basado en descenso del gradiente

Aprendizaje basado en descenso del gradiente

Descenso del gradiente

$$x' = x - \epsilon \nabla_x f(x)$$

Descenso del gradiente estocástico

En el descenso del gradiente, si:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

entonces el costo computacional del cálculo es $O(m)$.

Descenso del gradiente estocástico

Sea $B = x^{(1)}, ..., x^{(m')}$ con $x^{(i)}$ extraídos uniformemente del conjunto de entrenamiento. B se conoce como *mini-batch* con tamaño m', regularmente $1 < m' <\sim 1000$. Entonces:

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

y

$$\theta \leftarrow \theta - \epsilon g$$

Descenso del gradiente estocástico

En general, el descenso del gradiente se ha considerado lento o poco fiable. En el pasado, la aplicación del descenso de gradiente a problemas de optimización no convexos se consideraba temeraria o sin principios. Hoy en día, sabemos que los modelos de aprendizaje automático funcionan muy bien cuando se entrenan con el descenso de gradiente.

El algoritmo de optimización puede no estar garantizado para llegar incluso a un mínimo local en un tiempo razonable, pero a menudo encuentra un valor muy bajo de la función de coste lo suficientemente rápido como para ser útil.

Unidades ocultas

Unidades ocultas

¿Cómo elegir el tipo de unidades ocultas en las capas intermedias?
Se trata de un área de investigación muy activa y que todavía no está guiada por principios teóricos.

Recordemos...

$$h = g(W^T x + b)$$

Rectified Linear Unit (ReLU)

Generalizaciones de ReLU

$$h_i = g(z, \alpha)_i = max(0, z_i) + \alpha_i min(0, z_i)$$

- Absolute value rectivication: $\alpha_i = -1 \implies g(z) = |z|$.

Generalizaciones de ReLU

$$h_i = g(z, \alpha)_i = max(0, z_i) + \alpha_i min(0, z_i)$$

- Absolute value rectivication: $\alpha_i = -1 \implies g(z) = |z|$.
- Leaky ReLU: α_i tiene valores cercanos a 0,01.

Generalizaciones de ReLU

$$h_i = g(z, \alpha)_i = max(0, z_i) + \alpha_i min(0, z_i)$$

- Absolute value rectivication: $\alpha_i = -1 \implies g(z) = |z|$.
- Leaky ReLU: α_i tiene valores cercanos a 0,01.
- Parametric ReLU: α es un parámetro a aprender.

Generalizaciones de ReLU

Unidades Maxout

$$g(z)_i = maxz_j,\ j \in G^{(i)}$$

donde $G^{(i)}$ es el conjunto de índices dentro de las entradas para el grupo i, $(i-1)k+1, ..., i_k$. A mayor valor k, unidades maxout pueden aprender a aproximar cualquier función convexa con fidelidad arbitraria.

Unidades Maxout

$$g(z)_i = maxz_j,\ j \in G^{(i)}$$

donde $G^{(i)}$ es el conjunto de índices dentro de las entradas para el grupo i, $(i-1)k+1, ..., i_k$. A mayor valor k, unidades maxout pueden aprender a aproximar cualquier función convexa con fidelidad arbitraria.

Ayudan a...

Evitar el olvido catastrófico.

Unidades Maxout

Figure 1. Graphical depiction of how the maxout activation function can implement the rectified linear, absolute value rectifier, and approximate the quadratic activation function. This diagram is 2D and only shows how maxout behaves with a 1D input, but in multiple dimensions a maxout unit can approximate arbitrary convex functions.

Sigmoide logístico y tangente hiperbólica

Antes de exister ReLu, la mayoría de las redes neuronales usaban la función sigmoide logístico como función de activación

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

o la función tangente hiperbólica

$$g(z) = tanh(z)$$

Sigmoide logístico y tangente hiperbólica

Otras unidades ocultas: ventajas de función lineal

Consideremos una red neuronal con algunas paredes con n entradas y p salidas. $h = g(W^T x + b)$. Sean U, V son dos matrices de pesos, si la primera no tiene función de activación se puede sustituir $h = g(V^T U^T x + b)$.

Otras unidades ocultas: ventajas de función lineal

Consideremos una red neuronal con algunas paredes con n entradas y p salidas. $h = g(W^T x + b)$. Sean U, V son dos matrices de pesos, si la primera no tiene función de activación se puede sustituir $h = g(V^T U^T x + b)$. Si U produce q salidas, entonces juntas contienen $(n + p)q$ parámetros, mientras que W tendría np.

Otro tipo de unidades ocultas

- Función de base radial: $h_i = exp(-\frac{1}{\sigma_i^2}||W_{:}, i - x||^2)$. Más activa cuando x se acerca a un modelo $W_{:}, i$. Satura en 0 para la mayoría de los casos, es díficil de optimizar.
- Softplus: $g(a) = log(1 + e^a)$ (versión suavizada de un rectificador). Recomendada en la última capa, es mejor ReLU en capas ocultas.
- Hard tanh: $g(a) = max((-1, min(1, a))$, similar a tanh y ReLU, pero a diferencia de ReLU, está acotada.

Otro tipo de unidades ocultas

- Función de base radial: $h_i = exp(-\frac{1}{\sigma_i^2}||W_{:}, i - x||^2)$. Más activa cuando x se acerca a un modelo $W_{:}, i$. Satura en 0 para la mayoría de los casos, es díficil de optimizar.
- Softplus: $g(a) = log(1 + e^a)$ (versión suavizada de un rectificador). Recomendada en la última capa, es mejor ReLU en capas ocultas.
- Hard tanh: $g(a) = max((-1, min(1, a))$, similar a tanh y ReLU, pero a diferencia de ReLU, está acotada.

¿Nuevos tipos de unidades ocultas?

Hay gran variedad comparables con los tipos conocidos y son tan comunes que no resultan interesantes.

Diseño de arquitectura

Primera capa:

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)}),$$

Diseño de arquitectura

Primera capa:

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)}),$$

la segunda capa:

$$h^{(2)} = g^{(2)}(W^{(2)T}x + b^{(2)}),$$

Diseño de arquitectura

Primera capa:

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)}),$$

la segunda capa:

$$h^{(2)} = g^{(2)}(W^{(2)T}x + b^{(2)}),$$

y así, sucesivamente.

Diseño de arquitectura

Las redes más profundas suelen ser capaces de utilizar muchas menos unidades por capa y muchos menos parámetros y suelen generalizar en el conjunto de pruebas, pero también suelen ser más difíciles de optimizar. La arquitectura de red ideal para una tarea debe encontrarse mediante la experimentación guiada por el control del error del conjunto de validación.

Diseño de arquitectura

El teorema de aproximación universal establece que una red feedforward con una capa de salida lineal capa de salida lineal y al menos una capa oculta con cualquier función de activación (como la función de activación sigmoide logística) puede aproximar cualquier función de Borel medible de un espacio finito a otro con cualquier cantidad de error no nula deseada, siempre que la red tenga suficientes unidades ocultas.

Diseño de arquitectura

El teorema de aproximación universal establece que una red feedforward con una capa de salida lineal capa de salida lineal y al menos una capa oculta con cualquier función de activación (como la función de activación sigmoide logística) puede aproximar cualquier función de Borel medible de un espacio finito a otro con cualquier cantidad de error no nula deseada, siempre que la red tenga suficientes unidades ocultas.

Función de Borel

Cualquier función continua sobre un subconjunto cerrado y acotado de R^n es Borel medible y, por tanto, puede ser aproximada por una red neuronal.

Diseño de arquitectura

Las redes lineales a trozos (que pueden obtenerse de ReLU y sus variantes o las unidades maxout) pueden representar funciones con un número de regiones que es exponencial en la profundidad de la red.

Diseño de arquitectura

Las redes lineales a trozos (que pueden obtenerse de ReLU y sus variantes o las unidades maxout) pueden representar funciones con un número de regiones que es exponencial en la profundidad de la red.

Diseño de arquitectura: más profundo, mejor

Diseño de arquitectura

La elección de un modelo profundo codifica un modelo muy general de que la función que queremos aprender debe implicar la composición de varias funciones más simples.

Diseño de arquitectura

https://playground.tensorflow.org

Backpropagation y otros algoritmos de diferenciación

Backpropagation solo se refiere a calcular el gradiente, mientras que otro algoritmo, que podría ser del tipo del descenso del gradiente, se usa para llevar a cabo el aprendizaje de este gradiente.

Grafos computacionales

Backpropagation y regla de la cadena

Sea $y = g(x)$, $z = f(g(x)) = f(y)$

$$\frac{dz}{dx} = \frac{dz}{dy}$$

Sean $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$, si $y = g(x)$ y $z = f(y)$:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x_i}$$

En forma vectorial: $\nabla_x z = (\frac{\partial y}{\partial x})^T \nabla_y z$, donde $(\frac{\partial y}{\partial x}) = J_y \in \mathbb{M}^{n \times m}$.

Subexpresiones repetidas

$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$

Propagación hacia adelante general

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

Backpropagation simple

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

`grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return $\{$`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i\}$

Propagación hacia adelante en un Perceptrón Multicapa

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Backpropagation en un Perceptrón Multicapa

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ **do**
  Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term, where needed):
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for

Símbolo a símbolo / símbolo a número

Ejemplo: Representación de un MLP

1 capa oculta, relu, weight decay, cross entropy:

$$J = J_{\mathbb{MLE}} + \lambda \left(\sum_{i,j} (W_{i,j}^{(}1))^2 + \sum_{i,j} (W_{i,j}^{(}2))^2 \right)$$

Temas de la parte 3 del curso

1. Bases de Aprendizaje Automático
2. Redes de propagación hacia adelante
3. Regularización
   - Regularización
   - Parameter norm penalties
   - Inferencia bayesiana
   - 7.3-7.4
   - Robustez del ruido (noise robustness)
   - Aprendizaje semi-supervisado
   - Aprendizaje multi-tarea
   - Detención temprana (early stopping)
   - Dropout
4. Optimización para modelos de entrenamiento profundo

Regularización para AP

La regularización es cualquier modificación que se realiza en el algoritmo de aprendizaje para reducir el error de generalización.

$$\text{mayor sesgo} \leftrightarrow \text{menor varianza}$$

Idea:

El mejor modelo ajuste es aquel modelo general regularizado adecuadamente

Parameter norm penalties

Sea $J = J(\theta; X, y)$ la función objetivo, denotamos como $\tilde{J}$ a la función objetivo regularizada definida como

$$\tilde{J}((\theta; X, y) = J(\theta; X, y) + \alpha\Gamma(\theta)$$

- $\alpha \geq 0$
- Γ norm penalty

En AP se utilizaran normas que penalicen solamente los pesos en cada capa dejando los sesgos sin regularizar (evitando que aumente la vafianza & underfitting)

Parámetro de regularización L^2 (weight decay)

Sea

$$\Omega(\theta) = \frac{1}{2}||\omega||_2^2,$$

de esta forma

$$\tilde{J}((\omega; X, y) = J(\omega; X, y) + \frac{\alpha}{2}\omega^T\omega.$$

Como gradiente con respecto a los pesos es

$$\nabla_\omega \tilde{J}((\omega; X, y) = \nabla_\omega J(\omega; X, y) + \alpha\omega,$$

entonces actualizamos los pesos de la siguiente forma,

$$\omega \leftarrow (1 - \epsilon\alpha)\omega - \epsilon\nabla_\omega J(\omega; X, y)$$

Inferencia bayesiana

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8699938/

7.2-7.4

7.5. Robustez del ruido (noise robustness)

- Se analizó previamente la adición de ruido como estrategia para aumentar datos.

7.5. Robustez del ruido (noise robustness)

- Se analizó previamente la adición de ruido como estrategia para aumentar datos.
- La adición de ruido con una varianza infinitesimal en las entradas de los modelos es equivalente a imponer una penalización sobre las normas de los pesos (Bishop 1995a, Bishop 1995b)

7.5. Robustez del ruido (noise robustness)

- Se analizó previamente la adición de ruido como estrategia para aumentar datos.
- La adición de ruido con una varianza infinitesimal en las entradas de los modelos es equivalente a imponer una penalización sobre las normas de los pesos (Bishop 1995a, Bishop 1995b)
- La adición de ruido en las capas ocultas puede ser mucho más poderoso que simplemente compactar los parámetros. El caso más general de esto es el *dropout*.

7.5. Robustez del ruido (noise robustness)

También se suele agregar ruido a los pesos para regularizar modelos.

7.5. Robustez del ruido (noise robustness)

También se suele agregar ruido a los pesos para regularizar modelos.

- Se usa principalmente en redes recurrentes.

7.5. Robustez del ruido (noise robustness)

También se suele agregar ruido a los pesos para regularizar modelos.

- Se usa principalmente en redes recurrentes.
- Esto se puede interpretar como una implementación estocástica de inferencia Bayesiana sobre los pesos.

7.5. Robustez del ruido (noise robustness)

También se suele agregar ruido a los pesos para regularizar modelos.

- Se usa principalmente en redes recurrentes.
- Esto se puede interpretar como una implementación estocástica de inferencia Bayesiana sobre los pesos.
- El tratamiento Bayesiano del aprendizaje considera que los pesos tienen incertidumbre mediante una distribución de probabilidad.

7.5. Robustez del ruido (noise robustness)

También se suele agregar ruido a los pesos para regularizar modelos.

- Se usa principalmente en redes recurrentes.
- Esto se puede interpretar como una implementación estocástica de inferencia Bayesiana sobre los pesos.
- El tratamiento Bayesiano del aprendizaje considera que los pesos tienen incertidumbre mediante una distribución de probabilidad.
- Añadir ruido a los pesos es una forma estocástica de manifestar dicha incertidumbre.

7.5. Ruido como interpretación Bayesiana

1424 IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 7, NO. 6, NOVEMBER 1996

An Analysis of Noise in Recurrent Neural Networks: Convergence and Generalization

Kam-Chuen Jim, *Member, IEEE,* C. Lee Giles, *Senior Member, IEEE,* and Bill G. Horne, *Member, IEEE*

Practical Variational Inference for Neural Networks

Alex Graves
Department of Computer Science
University of Toronto, Canada
graves@cs.toronto.edu

7.5. Robustez del ruido (noise robustness)

El ruido aplicado a los pesos también puede ser interpretado como una forma de regularización tradicional (bajo ciertas suposiciones), garantizando estabilidad a la función objetivo.

7.5. Robustez del ruido (noise robustness)

El ruido aplicado a los pesos también puede ser interpretado como una forma de regularización tradicional (bajo ciertas suposiciones), garantizando estabilidad a la función objetivo.

Sean $\hat{y}$ predicciones y y valores reales:

$$J = \mathbb{E}_{p(x,y)}[(\hat{y}(x) - y)^2]$$

con m elementos etiquetados (x_i, y_i) del conjunto de entrenamiento.

7.5. Robustez del ruido (noise robustness)

El ruido aplicado a los pesos también puede ser interpretado como una forma de regularización tradicional (bajo ciertas suposiciones), garantizando estabilidad a la función objetivo.

Sean $\hat{y}$ predicciones y y valores reales:

$$J = \mathbb{E}_{p(x,y)}[(\hat{y}(x) - y)^2]$$

con m elementos etiquetados (x_i, y_i) del conjunto de entrenamiento.

Se puede asumir que cada entrada incluye una ruido (perturbación) aleatorio en los pesos: $\epsilon_W \sim N(\epsilon; 0, \eta I$. En un MLP, consideremos el modelo perturbado como $\hat{y}_{\epsilon_W}(x)$.

7.5. Robustez del ruido (noise robustness)

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\epsilon_W)}\left[(\hat{y}_{\epsilon_W}(x) - y)^2\right]$$

7.5. Robustez del ruido (noise robustness)

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\epsilon_W)}\left[(\hat{y}_{\epsilon_W}(x) - y)^2\right]$$

$$= \mathbb{E}_{p(x,y,\epsilon_W)}\left[(\hat{y}^2_{\epsilon_W}(x) - 2y\hat{y}_{\epsilon_W}(x) + y^2\right]$$

7.5. Robustez del ruido (noise robustness)

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\epsilon_W)}\left[(\hat{y}_{\epsilon_W}(x) - y)^2\right]$$

$$= \mathbb{E}_{p(x,y,\epsilon_W)}\left[(\hat{y}^2_{\epsilon_W}(x) - 2y\hat{y}_{\epsilon_W}(x) + y^2\right]$$

para pequeñas varianzas η, la minimización de J con ruido en los pesos (covarianza ηI) es equivalente a minimizar J con un término adicional de regularización:

$$\eta\mathbb{E}_{p(x,y)}\left[||\nabla_w \hat{y}(x)||^2\right]$$

7.5. Robustez del ruido (noise robustness)

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\epsilon_W)}\left[(\hat{y}_{\epsilon_W}(x) - y)^2\right]$$

$$= \mathbb{E}_{p(x,y,\epsilon_W)}\left[(\hat{y}^2_{\epsilon_W}(x) - 2y\hat{y}_{\epsilon_W}(x) + y^2\right]$$

para pequeñas varianzas η, la minimización de J con ruido en los pesos (covarianza ηI) es equivalente a minimizar J con un término adicional de regularización:

$$\eta\mathbb{E}_{p(x,y)}\left[||\nabla_w \hat{y}(x)||^2\right]$$

Esta forma de regularización empuja al modelo hacia regiones donde es relativamente insensible a pequeñas variaciones en los pesos, encontrando puntos que no son simplemente mínimos, sino mínimos rodeados de regiones planas.

Ejemplo simple en playground.tensorflow

Sin regularización, ni ruido en las entradas.

Ejemplo simple en playground.tensorflow

Con regularización, sin ruido en las entradas.

Ejemplo simple en playground.tensorflow

Sin regularización, con ruido en las entradas.

7.5.1 Agregando ruido a las salidas objetivo

Consideraciones:

- La mayoría de los conjuntos de datos tienen algunos errores en las etiquetas y.
- Puede ser perjudicial maximizar $logp(y|x)$ cuando y es un error.
- Una forma de evitarlo es modelar explícitamente modelar explícitamente el ruido en las etiquetas.

7.5.1 Agregando ruido a las salidas objetivo

Se puede asumir que...

para una pequeña constante ϵ, la etiqueta y del conjunto de entrenamiento es correcta con una probabilidad $1 - \epsilon$, en caso contrario cualquier otra posibilidad podría ser correcta.

7.5.1 Agregando ruido a las salidas objetivo

Se puede asumir que...

para una pequeña constante ϵ, la etiqueta y del conjunto de entrenamiento es correcta con una probabilidad $1 - \epsilon$, en caso contrario cualquier otra posibilidad podría ser correcta.

Esta suposición es más fácil de incorporar analíticamente a la función objetivo, que explícitamente muestrear ruido.

7.5.1 Agregando ruido a las salidas objetivo

Ejemplo: Label smoothing (suavizado de etiquetas)

Regulariza un modelo basado en softmax con k clases, remplazando las clasificaciones 0 y 1 con $\frac{\epsilon}{k-1}$ y $1-\epsilon$ respectivamente. Se usa desde la década de 1980 hasta la fecha.

7.5.1 Agregando ruido a las salidas objetivo

Ejemplo: Label smoothing (suavizado de etiquetas)

Regulariza un modelo basado en softmax con k clases, remplazando las clasificaciones 0 y 1 con $\frac{\epsilon}{k-1}$ y $1-\epsilon$ respectivamente. Se usa desde la década de 1980 hasta la fecha.

Se puede usar la función de pérdida entropía-cruzada para estos objetivos suavizados. Aprendizaje MLE con clasificador softmax puede nunca converger, al no poder predecir 0 y 1 de manera exacta e intentar ser cada vez más exacto.

7.5.1 Agregando ruido a las salidas objetivo

Ejemplo: Label smoothing (suavizado de etiquetas)

Regulariza un modelo basado en softmax con k clases, remplazando las clasificaciones 0 y 1 con $\frac{\epsilon}{k-1}$ y $1-\epsilon$ respectivamente. Se usa desde la década de 1980 hasta la fecha.

Se puede usar la función de pérdida entropía-cruzada para estos objetivos suavizados. Aprendizaje MLE con clasificador softmax puede nunca converger, al no poder predecir 0 y 1 de manera exacta e intentar ser cada vez más exacto.

Ventaja

Previene la búsqueda de probabilidades rígidas sin desalentar la clasificación correcta.

7.6 Aprendizaje semi-supervisado

Utiliza tanto ejemplos no etiquetados de $P(x)$ como ejemplos etiquetados de $P(x, y)$ para estimar $P(y|x)$ o predecir y a partir de x.

7.6 Aprendizaje semi-supervisado

En el contexto del aprendizaje profundo, el aprendizaje semi-supervisado, por lo regular se refiere a una representación $h = f(x)$.

7.6 Aprendizaje semi-supervisado

En el contexto del aprendizaje profundo, el aprendizaje semi-supervisado, por lo regular se refiere a una representación $h = f(x)$.

Objetivo

Aprender una representación en la cual los ejemplos de una misma clase tengan representaciones similares.

7.6 Aprendizaje semi-supervisado

En el contexto del aprendizaje profundo, el aprendizaje semi-supervisado, por lo regular se refiere a una representación $h = f(x)$.

Objetivo

Aprender una representación en la cual los ejemplos de una misma clase tengan representaciones similares.

Ejemplo

Usar PCA como preprocesamiento antes de aplicar un clasificador.

7.6 Aprendizaje semi-supervisado

Es pueden combinar los componentes supervisados y no supervisados en el modelo, de suerte que se tenga un modelo generativo de $P(x)$ o $P(x, y)$ compartan parámetros con un modelo discriminante de $P(y|x)$.

7.6 Aprendizaje semi-supervisado

Es pueden combinar los componentes supervisados y no supervisados en el modelo, de suerte que se tenga un modelo generativo de $P(x)$ o $P(x, y)$ compartan parámetros con un modelo discriminante de $P(y|x)$.

El uso de ejemplos no etiquetados para modelar $P(x)$ mejora significativamente $P(y|x)$.

7.6 Aprendizaje semi-supervisado

Journal Club

Comentar Capítulo 1 de Chapelle et. al (2006).

7.6 Aprendizaje semi-supervisado

Journal Club

Comentar Capítulo 1 de Chapelle et. al (2006).

Algunas conclusiones de la lectura con relación a regularización

Utilizar aprendizaje semi-supervisado permite generar modelos más robustos para los datos, pues en lugar de solo apuntalar esfuerzos a $p(y|x)$ también se considera la estructura intrínseca de los datos mediante $p(x)$.

7.6 Aprendizaje semi-supervisado

Journal Club

Comentar Capítulo 1 de Chapelle et. al (2006).

Algunas conclusiones de la lectura con relación a regularización

Utilizar aprendizaje semi-supervisado permite generar modelos más robustos para los datos, pues en lugar de solo apuntalar esfuerzos a $p(y|x)$ también se considera la estructura intrínseca de los datos mediante $p(x)$. Dado que la regularización tiene como objetivo evitar malos modelos (sobreajustados o subajustados) se podría considerar el aprendizaje semi-supervisado como un enfoque para ello.

7.7 Aprendizaje multi-tarea

Def.

Es una forma de mejorar la generalización mediante la puesta en común (pooling) de ejemplos (que pueden verse como restricciones blandas impuestas a los parámetros) que surjen de varias tareas.

7.7 Aprendizaje multi-tarea

Se puede plantear como dos tipos de tareas con sus asociados parámetros.

1. Parámetros de tareas específicas, que solo se benefician de los ejemplos de su tarea para lograr una buena generalización.
2. Parámetros genéricos compartidos a lo largo de todas las tareas, que se benefician de los datos agrupados de todas las tareas.

7.7 Aprendizaje multi-tarea

7.7 Aprendizaje multi-tarea

Premisa desde el punto de vista del aprendizaje profundo

Entre los factores que explican las variaciones observadas en los datos asociados a las distintas tareas, algunos son compartidos entre dos o más tareas.

7.7 Aprendizaje multi-tarea

Premisa desde el punto de vista del aprendizaje profundo

Entre los factores que explican las variaciones observadas en los datos asociados a las distintas tareas, algunos son compartidos entre dos o más tareas.

Ejemplo

Clasificador softmax incluye neuronas de tareas específicas.

Detención temprana (early stopping)

Motivación:

- A menudo hay modelos con la capacidad suficiente para provocar sobreajuste.
- El error de validación empieza a aumentar a medida que el de entrenamiento disminuye.
- Se pueden guardar los errores, pesos y etapas del entrenamiento para cada época y regresar a la mejor configuración, en lugar de a la última.

Detención temprana

Detención temprana - Ejemplo gráfico

Detención temprana - Ejemplo gráfico

Detención temprana - Algoritmo 7.1, primera parte

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$

Detención temprana - Algoritmo 7.1, segunda parte

while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*

Detención temprana (early stopping)

Detención temprana (early stopping)

Detención temprana

Generalidades:

- Es el método más común de regularización debido a su simplicidad.

Detención temprana

Generalidades:

- Es el método más común de regularización debido a su simplicidad.
- Los pasos de entrenamiento (épocas) se pueden ver como otro hiper-parámetro.

Detención temprana

Costos:

- Se debe evaluar también la función de costo durante el entrenamiento, aunque sea de manera periódica.

Detención temprana

Costos:

- Se debe evaluar también la función de costo durante el entrenamiento, aunque sea de manera periódica.
- Se debe guardar una copia de los mejores hiper-parámetros.

Detención temprana

Algunas ventajas:

- Es una forma de regularización poco trasgresora, ya que no requiere cambiar el procedimiento de aprendizaje, función objetivo o el valor de los parámetros disponibles.

Detención temprana

Algunas ventajas:

- Es una forma de regularización poco trasgresora, ya que no requiere cambiar el procedimiento de aprendizaje, función objetivo o el valor de los parámetros disponibles.
- Por ejemplo, en el decaimiento de pesos, se debe ser muy cuidadoso de no generar una red que caiga en un mínimo local malo en el cual haya, de manera patológica, pequeños pesos.

Detención temprana

Algunas ventajas:

- Es una forma de regularización poco trasgresora, ya que no requiere cambiar el procedimiento de aprendizaje, función objetivo o el valor de los parámetros disponibles.
- Por ejemplo, en el decaimiento de pesos, se debe ser muy cuidadoso de no generar una red que caiga en un mínimo local malo en el cual haya, de manera patológica, pequeños pesos.
- Se puede usar junto con otras estrategias de regularización (esto no lo habían destacado los autores en otros métodos).

Detención temprana - Algoritmo 7.2 - Estrategia 1

Ver algoritmo de la página 117.

Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This returns i^*, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for i^* steps.

Detención temprana - Algoritmo 7.3 - Estrategia 2

Algorithm 7.3 Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.
$\epsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \epsilon$ **do**
  Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for n steps.
end while

Ver algoritmo de la página 117.

Regularización con detención temprana

Izquierda.- Las líneas de contorno sólidas indican los contornos de la función de costo. La línea punteada indica la trayectoria tomada por el SGD a partir del origen. En lugar de detenerse en el punto w^* que minimiza el coste, la parada anticipada hace que la trayectoria se detenga en un punto $\tilde{w}$ anterior. **Derecha.**- Regularización L2.

Dropout

Dropout proporciona un método de regularización computacionalmente no costoso y capaz de regularizar una amplia familia de modelos.

Dropout

Dropout proporciona un método de regularización computacionalmente no costoso y capaz de regularizar una amplia familia de modelos. El Dropout se puede pensar como un método para embolsar múltiples ensambles de modelos. Lo cual conlleva a entrenar y evaluar múltiples modelos en cada ejemplo de prueba.

Dropout

Dropout entrena el ensamble que consiste en todas las sub-redes que pueden ser formadas al remover neuronas (no de salida) a partir de una red neuronal de base.

Dropout

Dropout entrena el ensamble que consiste en todas las sub-redes que pueden ser formadas al remover neuronas (no de salida) a partir de una red neuronal de base. De manera efectiva, se puede remover una neurona multiplicando por cero.

Dropout

Dropout

Dropout aproxima el siguiente proceso:

Para aprender con embolsado, definimos k modelos diferentes, construimos k conjuntos de datos diferentes mediante el muestreo del conjunto de entrenamiento con reemplazo, y luego entrenamos el modelo i en el conjunto de datos i.

Dropout

La probabilidad de cada unidad (mask), por lo regular es, 0.5 para unidades ocultas y 0.8 para unidades de entrada.

dropout

- μ especifica cuáles unidades incluir y $J(\theta, \mu)$ define el costo del modelo determinado por los parámetros θ y la máscara (mask) μ.

dropout

- μ especifica cuáles unidades incluir y $J(\theta, \mu)$ define el costo del modelo determinado por los parámetros θ y la máscara (mask) μ.
- El entrenamiento con dropout consiste en minimizar $\mathbb{E}_\mu J(\theta, \mu)$.

Dropout

Dropout es más efectivo que otros regularizadores estándar computacionalmente económicos, como el decaimiento de peso, las restricciones de norma y la regularización de actividad escasa. Dropout también puede combinarse con otras formas de regularización para lograr una mejora adicional.

Dropout

Ejecutar la inferencia* de μ en el modelo entrenado tiene el mismo costo por ejemplo que si no se usara el abandono, aunque debemos pagar el costo de dividir los pesos por 2 una vez antes de comenzar a ejecutar la inferencia en los ejemplos.

Dropout

Por lo general, el error de conjunto de validación óptimo es mucho menor cuando se usa Dropout, pero esto tiene el costo de un modelo mucho más grande y muchas más iteraciones del algoritmo de entrenamiento.

Dropout

Dropout ha inspirado a otros métodos:

- DropConnect
- Stochastic pooling

Dropout sigue siendo el método de ensamble implícito más utilizado.

La tarea es convertir un problema de entrenamiento de una red neuronal en un problema de optimización, aunque los modelos de optimización empleados para aprendizaje profundo difieren de muchas maneras de los algoritmos tradicionales de optimización.

En la mayoría de los casos, nos interesa una medida desempeño P que está definida con el conjunto de prueba y puede no ser manipulable directamente.

Por ello, se trata de minimizar una función de costo distinta a $\mathbf{J}(\theta)$, con el afán de mejorar P. Normalmente se quiere simplemente minimizar $\mathbf{J}(\theta)$.

La función típica de costo es:

$$J(\theta) = E_{(x,y)\sim\hat{p}_{data}} L(f(x:\theta), y)$$

- L - la función de pérdida
- $f(x:\theta)$ - el valor de salida predicho con el valor de entrada x
- $\hat{p}_{data}$ - una distribución empírica
- y - el valor objetivo de salida

Se desarrollará un caso supervisado sin regularización. Por lo que la ecuación anterior se preferirá minimizar la función objetivo correspondiente don de la esperanza es tomada sobre una distribución generada de la p_{data} en vez de solo un conjunto finito de entrenamiento.

$$J^*(\theta) = E_{(x,y)\sim p_{data}} L(f(x:\theta), y)$$

Minimización Empírica de Riesgo

La meta del algoritmo de aprendizaje automatizado es reducir el error dado generalizado esperado en la última ecuación, a esta cantidad se le conoce como *Riesgo*.

Si se conociera la distribución p_{data} el problema sería un clásico problema de optimización, pero si por el contrario solo se tiene una muestra de datos, tenemos un problema de aprendizaje automatizado.

De aquí que se puede reemplazar la distribución real por la distribución empírica definida por el conjunto de entrenamiento. Por lo que devolvemos el problema a una optimización del riesgo empírico dado por:

$$E_{(x,y)\sim\hat{p}_{data}} L(f(x:\theta), y) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)}:\theta), y^{(i)})$$

- con m el número de muestras

Este proceso basado en la minimización del promedio del error de entrenamiento es conocido como *minimización empírica de riesgo*.

Si el número de muestras m incrementa considerablemente se podría garantizar la convergencia haciendo uso de la Ley de los Grandes Números.

Por otro lado, está técnica es propensa al sobre ajuste, modelos con alta capacidad tienden a memorizar el conjunto de entrenamiento, y en muchos casos no es factible.

Además, los algoritmos modernos más efectivos están basados en el descenso del gradiente, pero muchas funciones de pérdida no tienen derivadas útiles.

Por estos dos problemas, raramente es utilizada la minimización empírica de riesgo.

Funciones Sustitutas de Pérdida y Detención Temprana

En ciertos casos, dependiendo de la función de pérdida es conveniente tomar funciones sustitutas para asegurar un mejor aprendizaje de la red, pues pueden tomar mucho tiempo en alcanzar algún valor esperado o avanzar muy lentamente al mismo, por ello se suelen usar funciones sustitutas pare eficientar el proceso.

Volviendo a las diferencias entre la optimización clásica y la aplicada al lenguaje automatizado, en la primera se busca generalmente alcanzar un valor mínimo o máximo local, por el contrario en la segunda ya sea en la función de pérdida original o en la sustituta, si se logra satisfacer la condición de detención temprana sería suficiente para el algoritmo.

Otro factor en el que se diferencian con algoritmos clásico de optimización es que se en este caso se actualizan en cada paso sin hacer uso del conjunto entero de términos de la función de costo. La mayoría de las propiedades de la función objetivo J usadas por la mayoría de algoritmos de optimización son a su vez esperanzas sobre el conjunto de entrenamiento.

La propiedad más común usada es el gradiente:

$$\bigtriangledown_\theta J(\theta) = E_{(x,y)\sim\hat{p}_{data}} \bigtriangledown_\theta \, logp_{model}(x, y : \theta)$$

Evaluar esta esperanza en el conjunto entero de entrenamiento puede ser muy costoso, por lo que en la práctica se calcula sobre un muestreo aleatorio de un número pequeño de datos de nuestro conjunto y se toma únicamente un promedio de estos.

Los algoritmos de optimización que usan el conjunto entero de entrenamiento se les conoce como *batch* o *métodos de gradiente determinista*.

Los algoritmos que usan solo un dato por iteración es conocido como *método estocástico* o también conocido como *métodos online*.

La mayoría de los algoritmos usados en el aprendizaje profundo, se encuentran en un punto medio de estos dos tipos, pues usan más que solo un dato pero menos del conjunto entero. A estos se les conoce como métodos *minibatch* o *minibatch estocástico*. El ejemplo clásico es el descenso del gradiente.

Para el tamaño de Minibatch generalmente se tiene los siguientes factores:

- Lotes grandes proveen estimaciones del gradiente más precisas, pero con menos salidas lineales
- Estructuras multinúcleo usualmente usan tamaños extremadamente pequeños.
- Si todas las muestras se procesaran en paralelo, entonces la cantidad de memoria escala con el tamaño del lote.
- Algunos tipos de hardware logran mejores tiempos de ejecución con tamaños específicos de arreglos (con GPU's es común que esto suceda con potencias de 2)
- Tamaños pequeños de lote pueden hacer un efecto de regularización quizá a causa del ruido que estos agregan al proceso de aprendizaje. De ser así se requiere también un tasa pequeña de aprendizaje con el objetivo de mantener la estabilidad.

Dependiendo de los datos y los algoritmos los lotes pueden tener distintos comportamientos, algunos más o menos sensibles al error de muestreo.

Los métodos que actualizan cálculos basados únicamente en en el gradiente $\mathbf{g}$ son usualmente más robustos y manejan tamaños de lotes más pequeños (al rededor de 100).

Los métodos de segundo orden que usan también la matriz Hessiana $\mathbf{H}$ y actualizan cálculos por medio de $H^1 g$, normalmente requieren de mucho más grades tamaños de lotes(al rededor de 10,000). Estos últimos tienden a amplificar aún más los errores pues aún pequeños cambios son propagados en las múltiples operaciones que deberán realizarse.

También es vital que los minibatches sean elegidos aleatoriamente. A su vez también se requiere de la independencia de las muestras para evitar sesgos. Muchos conjuntos de datos suelen contener datos cercanos altamente correlacionados, por lo que es necesario elegir siempre de manera aleatoria los elementos a tomar en cuenta para el minibatch.

Afortunadamente en la práctica comúnmente es suficiente aleatorizar la base de datos y almacenarlo de esta forma aleatoria.

Además, es posible realizar cálculos en paralelo de distintos minibatches, dichas aproximaciones aproximaciones desincronizadas son discutidas en el capítulo 12.

Una motivación interesante para el descenso de gradiente estocástico de minibatches es que sigue el gradiente del error de generalización siempre que no se repitan muestras. En vez de recibir un conjunto fijo de entrenamiento, el aprendizaje ve una nueva muestra a cada instante, con cada muestra (x,y) provenientes de la distribución generadora $p_{data}(x; y)$, bajo estas condiciones, las muestras nunca se repiten, cada obtención es un a muestra razonable de p_{data}.

La equivalencia más sencilla es deducir cuando ambas x y y son discretas, en este caso $J^*(\theta)$ puede ser escrita como:

$$J^*(\theta) = \sum_x \sum_y p_{data}(x, y) L(f(x : \theta), y)$$

Con el gradiente exacto:

$$g = \bigtriangledown_\theta J^*(\theta) = \sum_x \sum_y p_{data}(x, y) \bigtriangledown_\theta L(f(x : \theta), y)$$

Así que, se puede obtener un estimador insesgado de el gradiente exacto de la generalización del error obteniendo muestras de un minibatch de muestras $\{x^1, x^2, ..., x^m\}$ con sus correspondientes y^i tomados de la distribución generadora *pdata*, y calcular el gradiente de la perdida con respecto a los parámetros para este minibatch:

$$\hat{g} = \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$$

Actualizando θ en dirección de $\hat{g}$ realiza un descenso del gradiente estocástico en la generalización del error.

Es recomendable hacer varios recorridos, a menos que la base de datos sea demasiado grande. Cuando múltiples épocas son usadas, solo la primera tendrá un gradiente insesgado de la generalización del error, aunque épocas extras usualmente generan suficientes beneficios para disminuir el error del entrenamiento pero provoca un aumento en el gap entre el error de entrenamiento y el de prueba.

Malas condiciones

Tradicionalmente, se suele evitar la dificultad de la optimización diseñando cuidadosamente la función objetivo y sus restricciones para asegurar que el problema de optimización sea convexo.

Una mala condición se puede manifestar en un estancamiento del DGS (ligeros incrementos aumentan la función de costo). Una serie de Taylor de segundo orden sobre la función de costo predice que el descenso del gradiente con un paso de $-\epsilon g$ agregará al costo:

$$\frac{1}{2}\epsilon^2 g^T H g - \epsilon g^T g$$

Esto se vuelve un problema cuando $\frac{1}{2}\epsilon^2 g^T Hg$ supera a $\epsilon g^T g$.Para determinar cuando la condición es perjudicial al objetivo de la red, se puede monitorear la norma cuadrada del gradiente $g^T g$ y el termino $g^T Hg$. En muchos casos la norma del gradiente no se decrementa considerablemente pero $g^T Hg$ crece más de un orden de magnitud. El resultado es que el aprendizaje se vuelve lento a pesar de un gradiente robusto pues la taza de aprendizaje debe disminuir compensando para una curvatura más pronunciada.

Mínimo Local

Un problema en optimización es que podría tener múltiples puntos mínimos. Si la función es convexa con una región plana se tendría que cualquier mínimo podría ser considerado el mínimo global, pues para el objetivo que se busca esta sería una solución aceptable.

Con funciones no convexas con múltiples mínimos locales, tampoco representan un gran problema. Se tiene algo conocido como *identificabilidad del modelo*. Se dice que un modelo es identificable si un conjunto de entrenamiento lo suficientemente grande puede descartar todos los parámetros del modelo excepto uno.

Modelos con variables latentes son a menudo no identificables, por ejemplo en una red neuronal bastaría modificar el orden de las capas para generar un nuevo arreglo y generando una nueva configuración que propicia esta no identificabilidad para el resultado de los pesos. Esto se le conoce como *espacio de simetría de los pesos*.

Estos problemas de identificabilidad significan que puede haber una cantidad extremadamente grande o incontable de mínimos locales en la función de costo de una red neuronal, sin embargo, todos estos son equivalentes para la función de costo, por lo que en efecto no son un problema real causado de la no convexidad.

El mínimo local puede ser un problema cuando tiene un alto costo de comparación con respecto al global.

Por mucho tiempo se creyó que este era un gran problema, actualmente sigue siendo un tema de investigación pero se sospecha que en en redes neuronales suficientemente grandes la mayoría de los puntos mínimos tienen una función de costo con valores muy pequeños por lo que no es necesario encontrar el mínimo global.

Acantilados y gradientes explosivos

Las redes neuronales con múltiples capas a menudo tiene grandes cambios parecidos a acantilados, esto es resultado de la multiplicación de múltiples grandes pesos en conjunto.

Estos acantilados pueden ser muy peligrosos sin importar la dirección de aproximamiento, por fortuna estos pueden ser evitados haciendo uso de la heurística del *gradiente recortado* que se describe en el capítulo 10.

El descenso del gradiente propone dar grandes pasos mientras que el gradiente recortado da pasos suficientemente pequeños evitando salir fuera de la región a donde el gradiente indica su minimización. Este algoritmo no provee de un tamaño de paso únicamente el sentido.

Los acantilados son más propensos en las funciones de costo para las redes neuronales recurrentes, ya que estas involucran múltiples operaciones de varios factores con un por cada paso de tiempo.

Dependencias de largo plazo

Otra dificultada se presenta con grafos demasiados profundos, lo que ocurre en las redes neuronales recurrentes por la repetida aplicación de la misma operación a cada paso de una larga secuencia temporal. Por ejemplo, suponemos que se requiere multiplicar repetidamente la matriz W, después de t pasos, esto será equivalente a W^t.

Suponemos que W tiene una descomposición con eigenvectores tal que $W = Vdiag(\lambda)V^(- 1)$. Entonces se tiene que:

$$W^t = (Vdiag(\lambda)V^(- 1))^t = Vdiag(\lambda)^t V^{(-1)}$$

Cualquier λ_i que no este demasiado próximo a 1 tenderá a explotar o a desvanecerse.

Si se desvanece provocará que los parámetros no sepan hacia donde moverse para disminuir la función de costo.

Las redes recurrentes, usan la misma matriz W en cada paso, pero la retroalimentación de la red no, así inclusive con grandes grafos se puede evitar los puntos explosivos o los desvanecientes.

Gradientes inexactos

La mayoría de los algoritmos de optimización se construyen haciendo la suposición de que se tiene acceso al gradiente exacto o al Hessiano. En la práctica, se suele tener únicamente acceso a una cantidad sesgada o con ruido de estos valores, basados en estimaciones por medio de los minibatches.

En otros casos la función objetivo es intratable, y en esta situación suele ser de igual manera intratable el gradiente. Bajo estas condiciones se suele trabajar solo con aproximaciones.

Este problema incrementa más en modelos más avanzados, como por ejemplo la divergencia contrastiva proporciona una técnica para aproximar el gradiente de la log-likelihood intratable de una máquina de Boltzmann.

Varias algoritmos de optimización de redes neuronales están diseñados con imperfecciones en la estimación del gradiente. Esto puede ser evitado seleccionando una función de pérdida sustituta que es más fácil de aproximar que la función real de pérdida.

Pobre correspondencia entre la estructura global y local

Hasta el momento se a puesto atención a la optimización en la función de costo. Es posible sobrellevar estas adversidades como se ha visto. Mucha de la investigación están basada en estos problemas, aunque en la realidad

Mucha de la investigación están basados en alcanzar algún tipo de punto mínimo o crítico, pero en la practica no alcanzan ninguno de estos, inclusive, pueden no existir.

Futuras investigaciones deben de desarrollar un mejor entendimiento sobre los factores que influyen en el tamaño de la trayectoria de aprendizaje y una mejor caracterización del proceso.

Varias de las investigaciones actuales están enfocadas en encontrar buenos puntos iniciales donde se presenten problemas para la estructura global.

La función objetivo puede tener problemas tanto por una condición pobre o alguna discontinuidad en el gradiente, provocando que la región donde el gradiente de una buena solución sea muy pequeña.

En este caso el descenso local con pasos de tamaño ϵ puede definir razonablemente un camino a la solución pero solo en en una dirección de descenso con tamaño de pasos $\delta << \epsilon$, aunque pueda llevarnos a una solución, también puede no suceder así, y dado que el tamaño de paso es muy pequeño será muy costoso computacionalmente.

El objetivo a encontrar en esta sección es encontrar puntos iniciales donde se sea capaz de tener una región que tenga un buen comportamiento.

Límites Teóricos de la Optimización

Teóricamente se reafirman varios de los problemas ya mencionados. Aunque en el uso todos los problemas den aproximaciones razonables para la solución del problema en cuestión.

Un análisis teórico nos indica que encontrar mínimos puede ser extremadamente difícil o imposible, pero la experiencia nos dice que aún así se puede obtener resultados que reduzcan suficiente su valor sin lograr el mínimo lo cual es aceptable para el algoritmo.

Desarrollar cotas más realistas en el desempeño de los algoritmos de optimización todavía se mantiene como una meta importante para la investigación del aprendizaje automatizado.

Descenso del Gradiente Estocástico

Previamente se introdujo el algoritmo del descenso del gradiente, este puede ser acelerado siguiendo el gradiente de manera aleatoria en minibatches seleccionados (SGD, o DGS en español).

El DGS y sus variantes es probablemente el algoritmo de optimización para aprendizaje automático más usado. Como se vio previamente, se puede obtener un estimador insesgado para el gradiente tomando los promedios en un minibatch de m muestras de la distribución generadora.

Pseudocódigo para la actualización del entrenamiento en la iteración k

- Dar un learning rate $\leftarrow \epsilon_k$
- Parámetro inicial $\leftarrow \theta$
- **Mientras** NO se logre el criterio de parada **Hacer**
  Muestrear un minibatch de m muestras del conjunto de entrenamiento $\{x^{(1)}, ..., x^{(m)}\}$ con sus correspondientes objetivos $y^{(i)}$
  Calcular el gradiente estimado: $\hat{g} \leftarrow \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$
  Actualizar $\theta \leftarrow \theta - \epsilon \hat{g}$
- **Terminar Mientras**

Un parámetro crucial para el DGS es el learning rate. Anteriormente se describió el algoritmo usando un learning rate ϵ fijo. En la práctica es necesario disminuirlo gradualmente por lo que se denotará como ϵ_k.

Esto es debido a que el estimador del gradiente introduce una fuente de ruido que no desaparece ni siquiera cuando se alcanza el mínimo.

Comparando, el gradiente real del total de la función de costo se vuelve pequeño y luego $\mathbf{0}$ cuando nos acercamos y alcanzamos el mínimo usando el descenso del gradiente en el batch, tal descenso puede usar un learning rate fijo. Una condición suficiente para garantizar la convergencia es:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty$$

y

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

En la práctica, es común que el decaimiento del learning rate sea lineal hasta la iteración τ:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$$

con $\alpha = \frac{k}{\tau}$

Después de la iteración τ, es común que se deje ϵ constante.
La elección de este hiperparámetro es más un arte que algo científico. Se puede escoger vía ensayo-error, o en proporciones, o muy comúnmente por monitoreo de la función objetivo como una función del tiempo . La cuestión principal es elegir ϵ_0.

Para el valor de ϵ_0 los valores muy pequeños tienden a provocar un sobrecosto y tendrá una reducción muy pequeñas del error, haciendo muy lento el proceso. Si el valor es muy grande puede tenderá a hacer cambios muy drásticos pudiendo provocar oscilaciones divergentes.

Un óptimo valor para el learning rate es uno mayor al mejor de los que se pudiera conseguir en las primeras 100 iteraciones, pero no que sobrepase demasiado ese valor pues puede provocar inestabilidades.

La propiedad más importante del DGS y relacionada con el minibatch es que el tiempo de computo por actualización no crece con el número de muestras. Esto permite la convergencia inclusive cuando el conjunto de muestras se vuelve muy grande.

Para medir la tasa de convergencia se usa la medida del **error de exceso** $J(\theta) - min_\theta J(\theta)$. Cuando el DGS se aplica a un problema convexo este error de exceso anda en el $O(\frac{1}{\sqrt{k}})$ después de k iteraciones, mientras que en el caso fuertemente convexo es del $O(\frac{1}{k})$. Convergencias más rápidas que estas son presumiblemente sobreajustes.

Momentum

Mientras que el DGS se mantiene como una estrategia de optimización muy popular, su aprendizaje puede ser lento en ocasiones. El método de *momentum* está diseñado para acelerar el aprendizaje, especialmente para altas curvaturas, pequeños pero consistentes gradientes y gradientes ruidosos.

Este algoritmo introduce la variable v que juega el rol de velocidad. La velocidad genera una decaimiento exponencial del negativo del gradiente. Además, su nombre deriva de una analogía física, en donde el gradiente negativo es una fuerza moviendo una partícula a traves del espacio, de acuerdo con la las leyes de movimiento de Newton.

Un hiperparámetro $\alpha \in [0, 1)$ determina cuan rápido la contribución del gradiente anterior disminuye. La actualización se da de la siguiente manera:

$$v \leftarrow \alpha v - \epsilon \bigtriangledown_\theta \left(\frac{1}{m} \sum_i L(f(x^{(i)} : \theta), y^{(i)})\right)$$

$$\theta \leftarrow \theta + v$$

La velocidad acumula los elementos del gradiente. El más grande α está relacionado con ϵ, la mayoría de los gradientes previos afectan la dirección actual.

Representación gráfica del algoritmo de momentum

Anteriormente, el tamaño de paso era simplemente la norma del gradiente multiplicada por el learning rate. Actualmente, depende de que tan grande o alineado a secuencias se encuentra el gradiente. El tamaño de paso es el más grande cuando sucesivos puntos del gradiente apuntan en la misma dirección.

$$\frac{\epsilon||g||}{1-\alpha}$$

Se suele pensar en el hiperparámetro del momentum en términos de $\frac{1}{1-\alpha}$. Los valores más comunes para dicho α son: .5, .9 y .99.

Pseudocódigo de DGS con momentum

- Dar un learning rate $\leftarrow \epsilon$, dar un parámetro de momentum $\leftarrow \alpha$
- Parámetro inicial $\leftarrow \theta$, parámetro inicial $\leftarrow v$
- **Mientras** NO se logre el criterio de parada **Hacer**
  Muestrear un minibatch de m muestras del conjunto de entrenamiento $\{x^{(1)}, ..., x^{(m)}\}$ con sus correspondientes objetivos $y^{(i)}$
  Calcular el gradiente estimado: $g \leftarrow \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$
  Actualizar $v \leftarrow \alpha v - \epsilon g$
  Actualizar $\theta \leftarrow \theta + v$
- **Terminar Mientras**

Se puede observar que el algoritmo de momentum simula una partícula sujeta a un tiempo continuo con dinámicas newtonianas.

Nesterov Momentum

Se introduce una variante de momentum inspirada en el método de aceleración de Nesterov, la actualización está dada por:

$$v \leftarrow \alpha v - \epsilon \bigtriangledown_\theta [\frac{1}{m} \sum_i L(f(x^{(i)} : \theta + \alpha v), y^{(i)})]$$

$$\theta \leftarrow \theta + v$$

La diferencia radica en que el gradiente esta siendo evaluado, por lo que se puede identificar como un factor corrector

Estrategias de inicialización de parámetros

Algunos algoritmos de optimización no son iterativos por naturaleza y simplemente resuelven para un único punto. Otros pueden ser iterativos por naturaleza pero convergen a una solución aceptable en un tiempo considerable dependiendo de la inicialización.

En el aprendizaje automatizado puede ser muy compleja la tarea de estos algoritmos y son fuertemente afectados por la inicialización. Inclusive cuando cuentan con convergencia el punto inicial puede determinar la velocidad y el costo de esta convergencia.

Desarrollar mejoras en estrategias para inicializar es una tarea difícil pues aún hoy en día la optimización de las redes neuronales no es bien entendida. Nuestro entendimiento de como el punto inicial afecta la generalización es aún primitivo, ofrece poca o una nula guía para su elección.

Quizá la única propiedad bien conocida es que los parámetros iniciales necesitan *romper simetría* entre diferentes unidades. Si dos unidades ocultas con la misma función de activación están conectadas con el mismo valor de entrada, entonces deben de tener distintos parámetros de inicialización.

Se suele inicializar casi siempre los pesos con un modelo de valores escogidos aleatoriamente de una distribución Gaussiana o uniforme.

Pesos mas grandes producen un efecto ruptura de simetría más fuerte, ayudando a evitar las unidades redundantes. Al mismo tiempo ayudan a evitar la perdida de señal durante la propagación hacia adelante o hacia atrás. Si son demasiado grandes pueden provocar valores explosivos en las propagaciones, además que, podrían causar valores extremos que provoquen que la función de activación se sature.

En casos generales, el descenso del gradiente con early stopping no es lo mismo que el decaimiento de pesos, pero provee una leve analogía para pensar acerca del efecto de la inicialización

Algunas heurísticas poseen la posibilidad de escoger los parámetros iniciales. Una heurística es la inicialización de los pesos de una capa enteramente conectada de m entradas con n salidas por un muestreo para cada peso de $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$ mientras que *Golrot y Bengio* sugieren usar la inicialización normalizada.

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Esta heurística está diseñada para hacer un compromiso entre la inicialización de todas las capas que tengan la misma varianza de activación y la misma varianza de gradiente.

También se recomienda inicializar con matrices ortogonales aleatorias, con cuidado de escoger el factor de escala o ganancia g. Bajo este modelo se garantiza que el número total de iteraciones de entrenamiento requerirá para la convergencia es independiente de la profundidad.

Incrementar este factor de escala empuja a la red a que las activaciones incrementen en norma como se propaga hacia adelante, y el gradiente incrementa en norma como se propaga hacia atrás. *Susillo* muestra que elegir bien este factor es suficiente entrenar la red tan profundo como 1000 capas sin la necesidad de hacer inicializaciones ortogonales.

Esto genera una caminata aleatoria perfilada a preservar las normas, así la retroalimentación de la red puede evitar puntos de desvanecimiento o explosivos. Desafortunadamente, este criterio de optimalidad a menudo no conduce a un desempeño óptimo.

Martnes introdujo otro criterio de inicialización llamado *inicialización dispersa* en la cual cada unidad esta inicializada para tener exactamente k pesos distintos de cero. Esta ayuda a alcanzar una cantidad mayor de diversidad de unidades en cada inicialización.

La elección ya sea de una inicialización densa o dispersa puede ser hecha mediante hiperparámetros, y alternativamente también se podrían buscar manualmente la mejor escala de inicialización. Una buena manera es echar un vistazo al comportamiento del un solo minibatch.

Algoritmos con Learning Rate adaptativo

Las investigaciones en redes neuronales muestran que realmente uno de los hiperparámetros más difícil de escoger es el learning rate, pues tiene un alto impacto en el desempeño del modelo.

Si se cree que la dirección de la sensibilidad son de alguna manera alineadas a un eje, esto haría sentido para separar el learning rate para cada parámetro, y automáticamente adaptarlo en el transcurso del aprendizaje.

Un acercamiento temprano a esto fue el algoritmo *delta-bar-delta*, cuya idea es que si las derivadas parciales de la pérdida permanecían con la misma señal entonces el learning rate debería incrementarse, caso contrario disminuirse.

AdaGrad

Este algoritmo escala el learning rate inversamente proporcional a la raíz cuadrada de la suma de todos sus valores cuadrados de sus históricos. Los parámetros con derivadas parciales más grandes de la pérdida tiene una reducción de su learning rate.

Empíricamente se ha encontrado que la acumulación de los gradientes cuadrados al comienzo del entrenamiento puede resultar en un prematuro y excesivo decremento en un learning rate efectivo.

AdaGrad se desempeña bien para algunos modelos de aprendizaje profundo, pero no para todos.

Pseudocódigo AdaGrad

- Dar un learning rate global $\leftarrow \epsilon$
- Parámetro inicial $\leftarrow \theta$
- Constante pequeña $\leftarrow \delta$ (10^{-7})
- Inicializar la acumulación del gradiente a 0 $\leftarrow$ r
- **Mientras** NO se logre el criterio de parada **Hacer**
  Muestrear un minibatch de m muestras del conjunto de entrenamiento $\{x^{(1)}, ..., x^{(m)}\}$ con sus correspondientes objetivos $y^{(i)}$
  Calcular el gradiente: $g \leftarrow \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$
  Acumular el gradiente cuadrado $r \leftarrow r + g \bigodot g$
  Calcular la actualización $\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \bigodot g$ (división y raíz cuadrada aplicado a cierto elemento)
  Actualizar $\theta \leftarrow \theta + \Delta\theta$
- **Terminar Mientras**

RMSProp

Modifica el AdaGrad (diseñada para converger rápidamente con funciones convexas) para desempeñarse mejor en ajustes no convexos, cambiando la acumulación del gradiente en una media móvil ponderada exponencialmente.

AdaGrad encoje el learning rate de acuerdo con histórico entero del cuadrado del gradiente y puede hacer un learning rate muy pequeño antes de llegar a una estructura convexa. RMSProp usa un promedio decreciente exponencialmente para descartar el histórico y así poder lograr llegar a la estructura convexa.

El promedio móvil introduce un nuevo hiperpárametro ρ, que controla el tamaño de la escala del promedio móvil.

Empíricamente, el algoritmo ha mostrado ser efectivo y práctico para redes neuronales profundas. Actualmente es uno de los métodos de optimización de referencia que emplean de forma rutinaria los profesionales del aprendizaje profundo.

Pseudocódigo RMSProp

- Dar un learning rate global $\leftarrow \epsilon$
- Dar una tasa de decaimiento $\leftarrow \rho$
- Parámetro inicial $\leftarrow \theta$
- Constante pequeña $\leftarrow \delta$ (10^{-6})
- Inicializar la acumulación del gradiente a 0 $\leftarrow$ r
- **Mientras** NO se logre el criterio de parada **Hacer**
  Muestrear un minibatch de m muestras del conjunto de entrenamiento $\{x^{(1)}, ..., x^{(m)}\}$ con sus correspondientes objetivos $y^{(i)}$
  Calcular el gradiente: $g \leftarrow \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$
  Acumular el gradiente cuadrado $r \leftarrow \rho r + (1 - \rho) g \bigodot g$
  Calcular la actualización $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \bigodot g$ (división y raíz cuadrada aplicado a cierto elemento)
  Actualizar $\theta \leftarrow \theta + \Delta\theta$
- **Terminar Mientras**

Adam(adaptative moments)

Este es quizá la mejor variante en combinación de RMSProp y Momentum, con algunas importantes distinciones. Momentum esta directamente incorporado como una estimación de el momento de primer orden del gradiente. Primero, la forma más directa de agregar momentum al RMSProp es aplicar momentum a los gradientes reescalados, lo cual no tiene una motivación teórica clara. Segundo, este incluye una corrección del sesgo para estimar tanto los momentos de primero y segundo orden.

Es generalmente considerado como suficientemente robusto para escoger hiperparámetros, aunque el learning rate a veces necesita ser cambiado del sugerido default.

Pseudocódigo Adam

- Dar un tamaño de paso $\leftarrow \epsilon$ (sugerido default 0.001)
- Dar una tasa de decaimiento exponencial para estimación de momentos $\leftarrow \rho_1$ y ρ_2 en [0,1) (sugerido .9 y .999)
- Constante pequeña $\leftarrow \delta$ (10^{-8})
- Parámetro inicial $\leftarrow \theta$
- Inicializar variables de 1er y 2do momento a 0 $\leftarrow$ r,s
- Inicializar el paso de tiempo 0 $\leftarrow t$

- **Mientras** NO se logre el criterio de parada **Hacer**
  Muestrear un minibatch de m muestras del conjunto de entrenamiento $\{x^{(1)}, ..., x^{(m)}\}$ con sus correspondientes objetivos $y^{(i)}$
  Calcular el gradiente: $g \leftarrow \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$

$$t \leftarrow t + 1$$

  Actualizar estimación primer momento sesgado $s \leftarrow \rho_1 s + (1 - \rho_1) g$
  Actualizar estimación segundo momento sesgado
  $r \leftarrow \rho_2 r + (1 - \rho_2) g \bigodot g$
  Corregir sesgo del primer momento $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
  Corregir el sesgo del segundo momento $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
  Calcular la actualización $\Delta\theta \leftarrow -\epsilon \frac{\hat{s}}{\delta + \sqrt{\hat{r}}}$
  Actualizar $\theta \leftarrow \theta + \Delta\theta$
- **Terminar Mientras**

Métodos de aproximación de segundo orden

Para una exposición simple, la única función objetivo que se examina es el riesgo empírico:

$$J(\theta) = E_{x,y\sim\hat{p}_{data(x,y)}}[L(f(x;\theta), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})$$

Los métodos que se discutirán pueden ser fácilmente extendidos a funciones objetivos más generales.

Método de Newton

El método de Newton es un esquema de optimización basado en el uso de la aproximación en una expansión en serie de $J(\theta)$ por un polinomio de Taylor de segundo orden cercano a un punto θ_0, ignorando las derivadas de ordenes mayores.

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \bigtriangledown_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta - \theta_0)$$

Si resolvemos para un punto critico para esta función, obtenemos la regla del parámetro de Newton:

$$\theta^* = \theta_0 - H^{-1} \bigtriangledown_\theta J(\theta_0)$$

Así, para una función cuadrática local, reescalando el gradiente por H^{-1}, el método de Newton brinca directamente al mínimo.

Si la función objetivo es convexa pero no cuadrática, esta actualización se puede hacer iterativa, produciendo un algoritmo de entrenamiento asociado con el método de Newton, dado por el siguiente:

- Parámetro inicial $\leftarrow \theta$
- m muestras del conjunto de entrenamiento
- **Mientras** NO se logre el criterio de parada **Hacer**
  Calcular el gradiente: $g \leftarrow \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$
  Calcular el Hessiano: $H \leftarrow \frac{1}{m} \bigtriangledown_\theta^2 \sum_i L(f(x^{(i)} : \theta), y^{(i)})$
  Calcular la inversa del Hessiano: H^{-1}
  Calcula la actualización: $\Delta_\theta = -H^{-1}g$
  Aplicar actualización: $\theta \leftarrow \theta + \Delta\theta$
- **Terminar Mientras**

En el aprendizaje profundo, la superficie de la función objetivo típicamente es no convexa con varias caracteristicas como puntos silla que son problemáticos para el método. Si los valores propios del Hessiano no son todos positivos el método podría moverse en la dirección incorrecta. Esta situación puede se evitada con la regularización del Hessiano. Una regularización común incluye el agregar una constante a lo largo de la diagonal del Hessiano. La regularización se convierte en:

$$\theta^* = \theta_0 - [H(f(\theta_0)) + \alpha I]^{-1} \bigtriangledown_\theta f(\theta_0)$$

El Hessiano se encuentra dominado por αI

Gradientes Conjugados

Este método evita eficientemente los cálculos de la matriz inversa del Hessiano por el descenso iterativo de las *direcciones conjugadas*.

La figura ilustra como el método el descenso más precipitado, cuando se aplica a un cuenco cuadrático, los progresos pueden ser bastante ineficaces patrones de atrás y adelante y/o zig-zag. Esto ocurre porque cada linea de dirección de búsqueda, cuando está dado por el gradiente, está garantizado de ser ortogonal a la linea previa de busqueda de dirección.

En el método se busca encontrar una dirección de búsqueda que es *conjugada* a la dirección previa. En la iteración t del entrenamiento, la siguiente dirección de búsqueda d_t toma la forma:

$$d_t = \bigtriangledown_\theta J(\theta) + \beta_t d_{t-1}$$

donde β_t es un coeficiente cuya magnitud controla cuánto de la dirección d_{t-1}, se debe agregar a la actual dirección.

Dos direcciones d_t y d_{t-1} están definidas como el conjugadas si

$$d_t^T H d_{t-1} = 0$$

Se puede obtener dicho β_t sin hacer sus cálculos, en este sentido se tiene:

- Fletcher-Reeves:

$$\beta_t = \frac{\bigtriangledown_\theta J(\theta_t)^T \bigtriangledown_\theta J(\theta_t)}{\bigtriangledown_\theta J(\theta_{t-1})^T \bigtriangledown_\theta J(\theta_{t-1})}$$

- Polak-Ribière:

$$\beta_t = \frac{(\bigtriangledown_\theta J(\theta_t) - \bigtriangledown_\theta J(\theta_{t-1}))^T \bigtriangledown_\theta J(\theta)}{\bigtriangledown_\theta J(\theta_{t-1})^T \bigtriangledown_\theta J(\theta_{t-1})}$$

Para superficies cuadradas este algoritmo garantiza que el gradiente a lo largo de la dirección previa no incremente en magnitud.

Pseudocódigo método del gradiente conjugado:

- Parámetro inicial $\leftarrow \theta$
- m muestras del conjunto de entrenamiento
- Inicializar $\rho_0 = 0$
- Inicializar $g_0 = 0$
- Inicializar t $= 1$
- **Mientras** NO se logre el criterio de parada **Hacer**
  Inicializar el gradiente $g_t = 0$
  Calcular el gradiente: $g_t \leftarrow \frac{1}{m} \bigtriangledown_\theta \sum_i L(f(x^{(i)} : \theta), y^{(i)})$
  Calcular $\beta_t = \frac{(g_t - g_{t-1})^T g_t}{g_{t-1}^T g_{t-1}}$ (Polak-Ribière)
  Calcular la dirección de búsqueda: $\rho_t = -g_t + \beta_t \rho_{t-1}$
  Realizar búsqueda en linea para encontrar:
  $\epsilon^* = argmin_\epsilon \frac{1}{m} \sum_i L(f(x^{(i)} : \theta_t + \epsilon \rho_t), y^{(i)})$
  Aplicar actualización: $\theta_{t+1} = \theta_t + \epsilon^* \rho_t$
  Actualizar $t \leftarrow t + 1$
- **Terminar Mientras**

Estrategias de Optimización y Meta-Algoritmos

- Normalización del Batch
  Se sugiere una normalización haciendo uso de la media y la desviación de los parámetros, con el objetivo de reparametrizar casi cualquier red profunda, la cual reduce significantemente el problema de la coordinación de actualización entre capas
- Descenso Coordinado
  Hace referencia a llevar a cabo la optimización de $f(x)$ haciéndolo por cada x_i a la vez, garantizando llegar a un mínimo probalemente local.
- Promedio Polyak
  Consiste en dos variantes para casos convexos y no convexos que toma en cuenta un promedio dependiendo de la iteración en la que se encuentre, ayuda a una mejor convergencia.

- Preentrenamiento Supervisado
  Básicamente consiste en subdividir la tarea principal en pequeñas tareas con modelos simples.
- Diseño de Modelos para Ayudar en la Optimización
  Es deseable un mejor modelo a base de partes simples que un algoritmo muy poderoso.
- Métodos de Continuación y Currículum de Aprendizaje

Convolucionales (Parte 0)

Primer acercamiento: Ver notebook en el repositorio de la clase: Intro Conv nets

¿Qué tipos de datos procesan las redes convolucionales?

`Grid-like topology` (tipo malla) $\begin{cases} \text{Series de tiempo 1D grid} \\ \text{Imágenes 2D grid} \end{cases}$

En las series de tiempo, tomando tomando intervalos regulares de tiempo se puede obtener un arreglo tipo malla 1D.

Operaciones fundamentales en las redes convolucionales

- Convolución
- Pooling

Operación de convolución (wiki)

En matemáticas (en particular, análisis funcional), la convolución es una operación matemática en dos funciones (f y g) que produce una tercera función ($f * g$) que expresa cómo la forma de uno es modificado por el otro.

Operación de convolución (wiki)

En matemáticas (en particular, análisis funcional), la convolución es una operación matemática en dos funciones (f y g) que produce una tercera función ($f * g$) que expresa cómo la forma de uno es modificado por el otro. El término convolución se refiere tanto a la función de resultado

como al proceso de calcularla.

Operación de convolución (wiki)

Operación de convolución

Sea $x(t), t \in \mathbf{R}$, t tiempo.

Operación de convolución

Sea $x(t), t \in \mathbf{R}$, t tiempo.

En un experimento imaginario, las mediciones más recientes en el tiempo, son más relevantes. Sea a una variable que mide la edad de cada medición, entonces podemos usar una función de promedio pesado $w(a)$ que le asigne mayor relevancia a las mediciones recientes.

$$s(t) = \int x(a)w(t-a)da \tag{3}$$

Representación común de una convolución

$$s(t) = (x * w)(t)$$

Características de w

- w debe ser una función de densidad de probabilidad válida. De lo contrario, la salida no sería un promedio pesado.
- w debe ser cero para para todo argumento negativo (para el ejemplo imaginario). En general la convolución solo requiere que la integral esté definida.

Terminología de redes neuronales

$$s(t) = \int x(a)w(t-a)da$$

- La función x es la entrada.
- $w(t-a)$ es el kernel.
- La salida se conoce como mapa de características (feature map).

Terminología de redes neuronales

Convolución

Las convoluciones se usan en más de un eje a la vez. Considérese una imagen 2D I y un kernel 2D K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{4}$$

Convolución

Las convoluciones se usan en más de un eje a la vez. Considérese una imagen 2D I y un kernel 2D K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{4}$$

La convolución es conmutativa:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{5}$$

La Ec. 5 es la forma más habitual en ML.

Convolución

La propiedad conmutativa de convolución surge debido al cambio en el kernel. Si m crece, el índice dentro de la entrada crece, pero el índice del kernel decrece.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

La propiedad conmutativa es útil para probar algunas cuestiones matemáticas, pero en la implementación de una red neuronal no es relevante.

Convolución

Muchas librerías de redes neuronales implementan una función llamada **correlacion-cruzada**, que es la misma que la convolución pero sin cambiar el kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{6}$$

Las librerías que implementan esta función la llaman convolución.

Convolución en ML

La convolución se usa simultáneamente con otras funciones, y la combinación de estas funciones no conmuta independientemente de si la operación de convolución invierte su núcleo o no.

Convolución 2D sin flip (inversion de kernel)

Convoluciones discretas

Se pueden ver como una multiplicación por una matriz. La matriz suele tener muchas entradas repetidas. En 1D, la matriz es de Toeplitz, en 2D como una matriz cirulante.

$$T = \begin{pmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ j & h & g & f & a \end{pmatrix} \qquad \mathrm{circ}(\mathbf{h}) = \mathbf{H} = \begin{bmatrix} h_0 & h_3 & h_2 & h_1 \\ h_1 & h_0 & h_3 & h_2 \\ h_2 & h_1 & h_0 & h_3 \\ h_3 & h_2 & h_1 & h_0 \end{bmatrix}.$$

Convoluciones en redes neuronales

- La convolución generalmente corresponde a una matriz muy dispersa (una matriz cuyas entradas son en su mayoría iguales a cero).

Convoluciones en redes neuronales

- La convolución generalmente corresponde a una matriz muy dispersa (una matriz cuyas entradas son en su mayoría iguales a cero).
- Esto se debe a que el kernel suele ser mucho más pequeño que la imagen de entrada.

Convoluciones en redes neuronales

- La convolución generalmente corresponde a una matriz muy dispersa (una matriz cuyas entradas son en su mayoría iguales a cero).
- Esto se debe a que el kernel suele ser mucho más pequeño que la imagen de entrada.
- Cualquier algoritmo de red neuronal que funcione con la multiplicación de matrices y que no dependa de propiedades específicas de la estructura de la matriz debería funcionar con convolución, sin necesidad de realizar más cambios en la red neuronal.

Convoluciones en redes neuronales

- La convolución generalmente corresponde a una matriz muy dispersa (una matriz cuyas entradas son en su mayoría iguales a cero).
- Esto se debe a que el kernel suele ser mucho más pequeño que la imagen de entrada.
- Cualquier algoritmo de red neuronal que funcione con la multiplicación de matrices y que no dependa de propiedades específicas de la estructura de la matriz debería funcionar con convolución, sin necesidad de realizar más cambios en la red neuronal.
- Las redes neuronales convolucionales típicas hacen uso de las propiedades de las matrices mencionadas para manejar grandes entradas de manera eficiente, pero no son estrictamente necesarias desde una perspectiva teórica.

Convolución 2D, 1 entrada

Convolución 2D

Tamaño de los filtros

Stacking convolutions vertically

We can replace 5x5 or 7x7 convolution kernels with multiple 3x3 convolutions on top of one another.

5x5 convolution vs the equivalent stacked 3x3 convolutions

The original throughput is kept: a block of 2 convolutional layers of kernel size 3x3 behaves as if a 5x5 convolutional window were scanning the input.

(2 + 1 + 2) (5x5) is equivalent to *1 + (1 + 1 + 1) + 1* (3x3, 3x3).

But it results in a lighter number of parameters:

$n_5x5 = 5^2 * c^2 > 2 * n_3x3 = 2 * 3^2 * c^2$.

This is a ratio of *1.4* for 5x5 convolution kernel, *2* for 7x7 convolution kernel.

Fuente:medium.com/analytics-vidhya

¿Por qué los filtros tienen dimensiones impares?

- Se quiere extraer información de todos los vecinos de determinado píxel.
- Se requiere un píxel fuente.

¿Por qué los filtros tienen dimensiones impares?

- Se quiere extraer información de todos los vecinos de determinado píxel.
- Se requiere un píxel fuente.

¿Por qué los filtros tienen dimensiones impares?

- Para un filtro de tamaño impar, todos los píxeles de la capa anterior estarían simétricamente alrededor del píxel de salida.

¿Por qué los filtros tienen dimensiones impares?

- Para un filtro de tamaño impar, todos los píxeles de la capa anterior estarían simétricamente alrededor del píxel de salida.
- Sin esta simetría, tendremos que tener en cuenta las distorsiones en las capas, lo que ocurre cuando se utiliza un núcleo de tamaño par.

¿Por qué los filtros tienen dimensiones impares?

- Para un filtro de tamaño impar, todos los píxeles de la capa anterior estarían simétricamente alrededor del píxel de salida.
- Sin esta simetría, tendremos que tener en cuenta las distorsiones en las capas, lo que ocurre cuando se utiliza un núcleo de tamaño par.
- Por lo tanto, la mayoría de los filtros de tamaño par se omiten para promover la simplicidad de la implementación.

¿Por qué los filtros tienen dimensiones impares?

- Para un filtro de tamaño impar, todos los píxeles de la capa anterior estarían simétricamente alrededor del píxel de salida.
- Sin esta simetría, tendremos que tener en cuenta las distorsiones en las capas, lo que ocurre cuando se utiliza un núcleo de tamaño par.
- Por lo tanto, la mayoría de los filtros de tamaño par se omiten para promover la simplicidad de la implementación.
- Si se piensa en la convolución como una interpolación desde los píxeles dados a un píxel central, no se puede interpolar a un píxel central utilizando un filtro de tamaño par.

Fuente: towardsdatascience.com

Kernel vs filtro

Un kernel es una matriz de pesos que se multiplican con la entrada para extraer las características relevantes. Las dimensiones de la matriz del núcleo es lo que da nombre a la convolución.

[1] Fuente: towardsdatascience.com

Kernel vs filtro

Un kernel es una matriz de pesos que se multiplican con la entrada para extraer las características relevantes. Las dimensiones de la matriz del núcleo es lo que da nombre a la convolución.

Un filtro es una concatenación de múltiples núcleos, cada uno de ellos asignado a un canal concreto de la entrada. Los filtros tienen siempre una dimensión más que los núcleos. Por ejemplo, en las convoluciones 2D, los filtros son matrices 3D (que son una concatenación de matrices 2D) [1].

[1] Fuente: towardsdatascience.com

Visitar enlaces

- Notebook
- Efecto de los kernels

Motivaciones: 3 importantes ideas

- Interacciones dispersas

Motivaciones: 3 importantes ideas

- Interacciones dispersas
- Parámetros compartidos

Motivaciones: 3 importantes ideas

- Interacciones dispersas
- Parámetros compartidos
- Representaciones equivariantes

Interacciones dispersas

En las redes neuronales tradicionales...

Se usa multiplicación matricial por una matriz de parámetros con un parámetro separado que describe la interacción entre cada unidad de entrada y cada unidad de salida. Esto significa que cada unidad de salida interactúa con cada unidad de entrada.

Interacciones dispersas

En las redes neuronales tradicionales...

Se usa multiplicación matricial por una matriz de parámetros con un parámetro separado que describe la interacción entre cada unidad de entrada y cada unidad de salida. Esto significa que cada unidad de salida interactúa con cada unidad de entrada.

En las redes convolucionales...

Suelen tener interacciones dispersas (también denominadas conectividad dispersa o pesos dispersos). Esto se consigue haciendo el núcleo (kernel) más pequeño que la entrada.

Conectividad dispersa, vista desde abajo

Conectividad dispersa, vista desde arriba

Capas más profundas pueden estar conectadas indirectamente a toda o la mayor parte de la imagen de entrada.

Parámetros compartidos o pesos atados

Se trata de utilizar el mismo parámetro para más de una función en el modelo. función en un modelo.

En una red neuronal tradicional

Cada elemento de la matriz de pesos se utiliza exactamente una vez al calcular la salida de una capa. Se multiplica por un elemento de la entrada y nunca se vuelve a consultar.

Parámetros compartidos o pesos atados

Se trata de utilizar el mismo parámetro para más de una función en el modelo. función en un modelo.

En una red neuronal tradicional

Cada elemento de la matriz de pesos se utiliza exactamente una vez al calcular la salida de una capa. Se multiplica por un elemento de la entrada y nunca se vuelve a consultar.

En una red neuronal convolucional...

cada miembro del núcleo se utiliza en cada posición de la entrada (excepto quizás algunos de los píxeles del límite, dependiendo de las las decisiones de diseño relativas al límite).

Parámetros compartidos

Ventaja de conectividad dispersa y parámetros compartidos

La figura de la derecha se obtiene tomando cada píxel de la imagen original y quitando el valor del píxel vecino a la izquierda.

Nótese la fuerza de todos los bordes orientados verticalmente en la imagen de entrada, operación útil para la detección de objetos. Ambas imágenes tienen 280 píxeles de altura. La imagen de entrada tiene 320 píxeles de ancho, mientras que la imagen de salida tiene 319 píxeles de ancho.

Ventaja de conectividad dispersa y parámetros compartidos

- Esta transformación puede describirse mediante un núcleo de convolución que contiene dos elementos, y requiere $319 \times 280 \times 3 = 267,960$ operaciones de punto flotante (dos multiplicaciones y una adición por píxel de salida) para calcular utilizando la convolución.

Ventaja de conectividad dispersa y parámetros compartidos

- Esta transformación puede describirse mediante un núcleo de convolución que contiene dos elementos, y requiere $319 \times 280 \times 3 = 267,960$ operaciones de punto flotante (dos multiplicaciones y una adición por píxel de salida) para calcular utilizando la convolución.
- Describir la misma transformación con una multiplicación matricial requeriría $320 \times 280 \times 319 \times 280$, o más de 8×10^9de entradas en la matriz, así que la convolución es cuatro mil millones de veces más eficiente para representar esta transformación.

Ventaja de conectividad dispersa y parámetros compartidos

- Esta transformación puede describirse mediante un núcleo de convolución que contiene dos elementos, y requiere $319 \times 280 \times 3 = 267,960$ operaciones de punto flotante (dos multiplicaciones y una adición por píxel de salida) para calcular utilizando la convolución.
- Describir la misma transformación con una multiplicación matricial requeriría $320 \times 280 \times 319 \times 280$, o más de 8×10^9de entradas en la matriz, así que la convolución es cuatro mil millones de veces más eficiente para representar esta transformación.
- El sencillo algoritmo de multiplicación de matrices realiza más de 16 mil millones de operaciones de punto flotante, lo que hace que la convolución sea aproximadamente 60000 veces más eficiente computacionalmente.

Ventaja de conectividad dispersa y parámetros compartidos

- Esta transformación puede describirse mediante un núcleo de convolución que contiene dos elementos, y requiere $319 \times 280 \times 3 = 267,960$ operaciones de punto flotante (dos multiplicaciones y una adición por píxel de salida) para calcular utilizando la convolución.
- Describir la misma transformación con una multiplicación matricial requeriría $320 \times 280 \times 319 \times 280$, o más de 8×10^9de entradas en la matriz, así que la convolución es cuatro mil millones de veces más eficiente para representar esta transformación.
- El sencillo algoritmo de multiplicación de matrices realiza más de 16 mil millones de operaciones de punto flotante, lo que hace que la convolución sea aproximadamente 60000 veces más eficiente computacionalmente.
- La mayoría de las entradas de la matriz serían cero. Si almacenamos sólo las entradas no nulas de la matriz, entonces tanto la multiplicación de la matriz y la convolución requerirían el mismo número de operaciones de punto flotante para calcularlas.

Ventaja de conectividad dispersa y parámetros compartidos

- La mayoría de las entradas de la matriz serían cero. Si almacenamos sólo las entradas no nulas de la matriz, entonces tanto la multiplicación de la matriz y la convolución requerirían el mismo número de operaciones de punto flotante para calcularlas.

Ventaja de conectividad dispersa y parámetros compartidos

- La mayoría de las entradas de la matriz serían cero. Si almacenamos sólo las entradas no nulas de la matriz, entonces tanto la multiplicación de la matriz y la convolución requerirían el mismo número de operaciones de punto flotante para calcularlas.
- La matriz tendría que seguir conteniendo $2 \times 319 \times 280 = 178,640$entradas. La convolución es una forma extremadamente eficiente de describir las transformaciones que aplican la misma transformación lineal lineal de una pequeña región local en toda la entrada.

Equivariancia a traslaciones

En el caso de la convolución, la forma particular de compartir los parámetros hace que la capa tenga una propiedad llamada equidistancia a la traslación.

Equivariancia a traslaciones

En el caso de la convolución, la forma particular de compartir los parámetros hace que la capa tenga una propiedad llamada equidistancia a la traslación.
Una función es equivariante si cuando la entrada cambia, la salida cambia de la misma manera.

Equivariancia a traslaciones

En el caso de la convolución, la forma particular de compartir los parámetros hace que la capa tenga una propiedad llamada equidistancia a la traslación.
Una función es equivariante si cuando la entrada cambia, la salida cambia de la misma manera.

$f(x)$ es equivariante a una función g

si $f(g(x)) = g(f(x))$.

En el caso de la convolución, si g es cualquier función que desplace la entrada, entonces la función de convolución es equivariante respecto a g.

Equivariancia

Ejemplo:

Sea I sea una función que da el brillo de la imagen en coordenadas enteras. Sea g una función que mapea una función de imagen a otra función de imagen, de manera que $I = g(I)$ es la función de imagen con $I(x, y) = I(x - 1, y)$.

Equivariancia

Ejemplo:

Sea I sea una función que da el brillo de la imagen en coordenadas enteras. Sea g una función que mapea una función de imagen a otra función de imagen, de manera que $I = g(I)$ es la función de imagen con $I(x, y) = I(x - 1, y)$.
Esto desplaza cada píxel de I una unidad a la derecha. Si aplicamos esta transformación a I, y luego aplicamos la convolución, el resultado será el mismo que si aplicamos la convolución a I , y luego aplicamos la transformación g a la salida.

Equivariancia a traslaciones

La convolución no es naturalmente equivariante a otro tipo de transformaciones, como rotaciones o cambios en la escala. En estos casos, es necesario valerse de otros mecanismos para manejar estos tipos de transformaciones.

Ventaja de la equivariancia a traslaciones

Cuando se procesan imágenes, es útil detectar los bordes en la primera capa de una red convolucional. Los mismos bordes aparecen más o menos

en toda la imagen, por lo que resulta práctico compartir los parámetros en toda la imagen. En algunos casos, es posible que no queramos compartir

los parámetros en toda la imagen.

Ventaja de la equivariancia a traslaciones

Fuente: https://chriswolfvision.medium.com/

Ventaja de la equivariancia a traslaciones

Ver giff animado. Fuente: https://chriswolfvision.medium.com/

Pooling

Función de pooling

Reemplaza la salida de una red a cierta ubicación con estadística de resumen de sus salidas más cercanas.

Ayuda a tener invarianza a traslaciones dados pequeños cambios en la entrada.

Puede considerarse como la adición de un prior infinitamente fuerte de que la función que aprende la capa debe ser invariable a pequeñas traslaciones. Cuando esta suposición es correcta, puede mejorar mucho la eficiencia estadística de la red.

Ejemplo: Max-pooling, stride 1

Fuente: Implement pooling from scratch in Python.

Ejemplo: Max-pooling, stride 2

Fuente: Implement pooling from scratch in Python.

Invarianza pooling

Invarianza pooling

Pooling, stride 2 (downsampling)

Redes neuronales recurrentes y recursivas

Análogo a las CNN con imágenes.

- Especializadas en procesar una secuencia de valores $x^1, ..., x^T$.
- Pueden escalar a secuencias mucho más grandes que lo que son prácticas en otras redes neuronales (p.ej. MLP).
- Pueden procesar secuencias de longitud variable.

Aplicaciones prácticas de RNNs

- Problemas de predicción.
- Modelización del lenguaje y generación de textos.
- Traducción automática.
- Reconocimiento del habla.
- Generación de descripciones de imágenes.
- Etiquetado de vídeos.
- Resumen de textos.
- Detección de rostros.
- Otras aplicaciones como Composición musical.

Compartir parámetros

Del MLP a RNN

1980's -¿ compartir parámetros através de diferentes puntos del modelo.

Sin esto no se podría generalizar a secuencias de diferente longitud no vistas durante el entrenamiento o tener diferentes piezas de información a diferentes tiempos.

Ejemplo

La independencia de México inició **en 1810**.
En 1810, inició la independencia de México.

- Un MLP procesaría 1 parámetro por pieza de información.
- Una RNN comparte parámetros en varios intervalos/pasos de tiempo.

Primera idea: convoluciones 1D

.

Time-delayed neural networks

- Comparte parámetros.
- No es profunda.

Compartir parámetros de manera diferente a una CNN

- Cada salida es una función de los miembros previos de la salida.

Compartir parámetros de manera diferente a una CNN

- Cada salida es una función de los miembros previos de la salida.
- Cada miembro de la salida se produce usando la misma regla de actualización de pesos aplicada a las salidas previas.

Compartir parámetros de manera diferente a una CNN

- Cada salida es una función de los miembros previos de la salida.
- Cada miembro de la salida se produce usando la misma regla de actualización de pesos aplicada a las salidas previas.
- Esto permite compartir parámetros en grafos muy profundos.

Desdoblando (unfolding) grafos computacionales

Considérese un sistema dinámico.

$$s^t = f(s^{(t-1)}; \theta),$$

donde s^t es llamado el estado del sistema y la ecuación es recurrente. Para un número finito de pasos de tiempo τ, el grafo se puede desdoblar aplicando $\tau - 1$ veces la definición.

Desdoblando (unfolding) grafos computacionales

Considérese un sistema dinámico.

$$s^t = f(s^{(t-1)}; \theta),$$

donde s^t es llamado el estado del sistema y la ecuación es recurrente. Para

un número finito de pasos de tiempo τ, el grafo se puede desdoblar aplicando $\tau - 1$ veces la definición.

Si $\tau = 3$

$$s^{(3)} = f(s^{(2)}; \theta)$$

$$= f(f(s^{(1)}; \theta); \theta)$$

Desdoblamiento (unfolding) de grafos

Figure 10.1: The classical dynamical system described by equation 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t and the function f maps the state at t to the state at $t+1$. The same parameters (the same value of $\boldsymbol{\theta}$ used to parametrize f) are used for all time steps.

Desdoblamiento RNN

Así como cualquier función se puede considerar una FFNN, cualquier función que incluya recurrencia puede considerarse una RNN.

Desdoblamiento RNN

Así como cualquier función se puede considerar una FFNN, cualquier función que incluya recurrencia puede considerarse una RNN. Considérese un sistema dinámico con una señal externa x^t:

$$s^t = f(s^{(t-1)}, x^t; \theta),$$

En términos de red neuronal, definamos los valores de las unidades ocultas:

$$h^t = f(h^{(t-1)}, x^t; \theta),$$

Desdoblamiento RNN

El grafo desdoblado, tiene un tamaño que depende de la longitud de la secuencia.

Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)*Circuit diagram. The black square indicates a delay of a single time step. *(Right)*The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

Ventajas del proceso de desdoblamiento

$$h^t = g^t(x^{(t)}, x^{(t-1)}, x^{(t-2)}, ..., x^{(2)}, x^{(1)})$$
$$= f(h^{(t-1)}, x^{(t)}; \theta)$$

1. Independientemente de la longitud de la secuencia, el modelo aprendido siempre tiene el mismo tamaño de entrada, porque se especifica en términos de transición de un estado a otro estado, en lugar de especificarse en términos de una historia de longitud variable de estados.

Ventajas del proceso de desdoblamiento

$$h^t = g^t(x^{(t)}, x^{(t-1)}, x^{(t-2)}, ..., x^{(2)}, x^{(1)})$$
$$= f(h^{(t-1)}, x^{(t)}; \theta)$$

1. Independientemente de la longitud de la secuencia, el modelo aprendido siempre tiene el mismo tamaño de entrada, porque se especifica en términos de transición de un estado a otro estado, en lugar de especificarse en términos de una historia de longitud variable de estados.

2. Es posible utilizar la misma función de transición f con los mismos parámetros en cada paso de tiempo.

Redes neuronales recurrentes

Redes neuronales recurrentes

Fuente: towardsdatascience.com

Redes neuronales recurrentes

Fuente: towardsdatascience.com