

Clase 19 - Redes neuronales (parte 3)

Aprendizaje de Máquinas - MA5204

Felipe Tobar

Department of Mathematical Engineering &
Center for Mathematical Modelling
Universidad de Chile

14 de marzo de 2021



UNIVERSIDAD
DE CHILE

Algoritmos de Optimización - Minibatch

Los algoritmos de optimización para aprendizaje de máquinas típicamente actualizan los parámetros usando un valor esperado del costo, obtenido a través de un subconjunto de los términos de la función de costos. La propiedad más usada respecto de la función objetivo es que

$$\nabla_{\theta} J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\theta} \log p_{\text{modelo}}(y|\mathbf{x})$$

Los algoritmos de optimización que usan el set de entrenamiento completo para actualizar los parámetros en cada iteración se conocen como **métodos de batch** o **determinísticos** y tienden a quedar atrapados en óptimos locales.

Los algoritmos que usan un solo ejemplo a la vez se conocen como **métodos estocásticos** y son tremendamente ineficientes para una cantidad grande de datos. Por lo tanto, la mayoría de los algoritmos usados pertenecen a una categoría intermedia, estos son los **métodos de minibatch** o **minibatch estocástico**, los cuales usan un subconjunto de tamaño reducido y calculan un promedio de gradientes para obtener el valor esperado.

Algoritmos de Optimización - Minibatch

Los algoritmos de optimización para aprendizaje de máquinas típicamente actualizan los parámetros usando un valor esperado del costo, obtenido a través de un subconjunto de los términos de la función de costos. La propiedad más usada respecto de la función objetivo es que

$$\nabla_{\theta} J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\theta} \log p_{\text{modelo}}(y|\mathbf{x})$$

Los algoritmos de optimización que usan el set de entrenamiento completo para actualizar los parámetros en cada iteración se conocen como **métodos de batch** o **determinísticos** y tienden a quedar atrapados en óptimos locales.

Los algoritmos que usan un solo ejemplo a la vez se conocen como **métodos estocásticos** y son tremendamente ineficientes para una cantidad grande de datos. Por lo tanto, la mayoría de los algoritmos usados pertenecen a una categoría intermedia, estos son los **métodos de minibatch** o **minibatch estocástico**, los cuales usan un subconjunto de tamaño reducido y calculan un promedio de gradientes para obtener el valor esperado.

Algoritmos de Optimización - Minibatch

Los algoritmos de optimización para aprendizaje de máquinas típicamente actualizan los parámetros usando un valor esperado del costo, obtenido a través de un subconjunto de los términos de la función de costos. La propiedad más usada respecto de la función objetivo es que

$$\nabla_{\theta} J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\theta} \log p_{\text{modelo}}(y|\mathbf{x})$$

Los algoritmos de optimización que usan el set de entrenamiento completo para actualizar los parámetros en cada iteración se conocen como **métodos de batch** o **determinísticos** y tienden a quedar atrapados en óptimos locales.

Los algoritmos que usan un solo ejemplo a la vez se conocen como **métodos estocásticos** y son tremendamente ineficientes para una cantidad grande de datos. Por lo tanto, la mayoría de los algoritmos usados pertenecen a una categoría intermedia, estos son los **métodos de minibatch** o **minibatch estocástico**, los cuales usan un subconjunto de tamaño reducido y calculan un promedio de gradientes para obtener el valor esperado.

Algoritmos de Optimización - Minibatch

Los algoritmos de optimización para aprendizaje de máquinas típicamente actualizan los parámetros usando un valor esperado del costo, obtenido a través de un subconjunto de los términos de la función de costos. La propiedad más usada respecto de la función objetivo es que

$$\nabla_{\theta} J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\theta} \log p_{\text{modelo}}(y|\mathbf{x})$$

Los algoritmos de optimización que usan el set de entrenamiento completo para actualizar los parámetros en cada iteración se conocen como **métodos de batch** o **determinísticos** y tienden a quedar atrapados en óptimos locales.

Los algoritmos que usan un solo ejemplo a la vez se conocen como **métodos estocásticos** y son tremendamente ineficientes para una cantidad grande de datos. Por lo tanto, la mayoría de los algoritmos usados pertenecen a una categoría intermedia, estos son los **métodos de minibatch** o **minibatch estocástico**, los cuales usan un subconjunto de tamaño reducido y calculan un promedio de gradientes para obtener el valor esperado.

Algoritmos de Optimización - Minibatch

Los algoritmos de optimización para aprendizaje de máquinas típicamente actualizan los parámetros usando un valor esperado del costo, obtenido a través de un subconjunto de los términos de la función de costos. La propiedad más usada respecto de la función objetivo es que

$$\nabla_{\theta} J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\theta} \log p_{\text{modelo}}(y|\mathbf{x})$$

Los algoritmos de optimización que usan el set de entrenamiento completo para actualizar los parámetros en cada iteración se conocen como **métodos de batch** o **determinísticos** y tienden a quedar atrapados en óptimos locales.

Los algoritmos que usan un solo ejemplo a la vez se conocen como **métodos estocásticos** y son tremendamente ineficientes para una cantidad grande de datos. Por lo tanto, la mayoría de los algoritmos usados pertenecen a una categoría intermedia, estos son los **métodos de minibatch** o **minibatch estocástico**, los cuales usan un subconjunto de tamaño reducido y calculan un promedio de gradientes para obtener el valor esperado.

Algoritmos de Optimización - Variantes del SGD

Promedio móvil exponencial: Esta modificación consiste en promediar el gradiente con su valor en la iteración anterior con el fin de seguir la dirección de gradientes pesados (momentum) , su implementación es como sigue

$$v_t = (1 - \beta) \left(\frac{\partial J}{\partial \theta_t} \right) + \beta v_{t-1}; \quad v_0 = 0$$

La actualización de parámetros viene dada por

$$\theta_{t+1} = \theta_t - \lambda v_t$$

Adagrad: Esta modificación es sobre el learning rate pues nos gustaría avanzar en alguna dirección incluso cuando el gradiente es pequeño. Definimos

$$v_t = v_{t-1} + \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

Con actualizaciones dadas por

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Donde ϵ es un parámetro para estabilidad numérica

Algoritmos de Optimización - Variantes del SGD

Promedio móvil exponencial: Esta modificación consiste en promediar el gradiente con su valor en la iteración anterior con el fin de seguir la dirección de gradientes pesados (momentum) , su implementación es como sigue

$$v_t = (1 - \beta) \left(\frac{\partial J}{\partial \theta_t} \right) + \beta v_{t-1}; \quad v_0 = 0$$

La actualización de parámetros viene dada por

$$\theta_{t+1} = \theta_t - \lambda v_t$$

Adagrad: Esta modificación es sobre el learning rate pues nos gustaría avanzar en alguna dirección incluso cuando el gradiente es pequeño. Definimos

$$v_t = v_{t-1} + \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

Con actualizaciones dadas por

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Donde ϵ es un parámetro para estabilidad numérica

Algoritmos de Optimización - Variantes del SGD

Promedio móvil exponencial: Esta modificación consiste en promediar el gradiente con su valor en la iteración anterior con el fin de seguir la dirección de gradientes pesados (momentum) , su implementación es como sigue

$$v_t = (1 - \beta) \left(\frac{\partial J}{\partial \theta_t} \right) + \beta v_{t-1}; \quad v_0 = 0$$

La actualización de parámetros viene dada por

$$\theta_{t+1} = \theta_t - \lambda v_t$$

Adagrad: Esta modificación es sobre el learning rate pues nos gustaría avanzar en alguna dirección incluso cuando el gradiente es pequeño. Definimos

$$v_t = v_{t-1} + \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

Con actualizaciones dadas por

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Donde ϵ es un parámetro para estabilidad numérica

Algoritmos de Optimización - Variantes del SGD

Promedio móvil exponencial: Esta modificación consiste en promediar el gradiente con su valor en la iteración anterior con el fin de seguir la dirección de gradientes pesados (momentum) , su implementación es como sigue

$$v_t = (1 - \beta) \left(\frac{\partial J}{\partial \theta_t} \right) + \beta v_{t-1}; \quad v_0 = 0$$

La actualización de parámetros viene dada por

$$\theta_{t+1} = \theta_t - \lambda v_t$$

Adagrad: Esta modificación es sobre el learning rate pues nos gustaría avanzar en alguna dirección incluso cuando el gradiente es pequeño. Definimos

$$v_t = v_{t-1} + \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

Con actualizaciones dadas por

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Donde ϵ es un parámetro para estabilidad numérica

Algoritmos de Optimización - Variantes del SGD

Promedio móvil exponencial: Esta modificación consiste en promediar el gradiente con su valor en la iteración anterior con el fin de seguir la dirección de gradientes pesados (momentum) , su implementación es como sigue

$$v_t = (1 - \beta) \left(\frac{\partial J}{\partial \theta_t} \right) + \beta v_{t-1}; \quad v_0 = 0$$

La actualización de parámetros viene dada por

$$\theta_{t+1} = \theta_t - \lambda v_t$$

Adagrad: Esta modificación es sobre el learning rate pues nos gustaría avanzar en alguna dirección incluso cuando el gradiente es pequeño. Definimos

$$v_t = v_{t-1} + \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

Con actualizaciones dadas por

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Donde ϵ es un parámetro para estabilidad numérica

Algoritmos de Optimización - Variantes del SGD

Promedio móvil exponencial: Esta modificación consiste en promediar el gradiente con su valor en la iteración anterior con el fin de seguir la dirección de gradientes pesados (momentum) , su implementación es como sigue

$$v_t = (1 - \beta) \left(\frac{\partial J}{\partial \theta_t} \right) + \beta v_{t-1}; \quad v_0 = 0$$

La actualización de parámetros viene dada por

$$\theta_{t+1} = \theta_t - \lambda v_t$$

Adagrad: Esta modificación es sobre el learning rate pues nos gustaría avanzar en alguna dirección incluso cuando el gradiente es pequeño. Definimos

$$v_t = v_{t-1} + \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

Con actualizaciones dadas por

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Donde ϵ es un parámetro para estabilidad numérica

Algoritmos de Optimización - Variantes del SGD

RMSprop: Esta modificación intenta ser una mejora a la anterior, Adagrad, y lo que hace es introducir promedios móviles exponenciales sobre el learning rate de la siguiente forma

$$v_t = \beta v_{t-1} + (1 - \beta) \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

y actualiza

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Adam: Finalmente esta modificación es simplemente una combinación de las ideas de learning rate adaptativo RMSprop y momentum. Actualmente es una de las más utilizadas.

Observación

Hasta ahora hemos visto varias modificaciones del descenso de gradiente estocástico pero cabe destacar que no existen grandes resultados teóricos que las sustenten, es decir, corresponden a heurísticas.

Algoritmos de Optimización - Variantes del SGD

RMSprop: Esta modificación intenta ser una mejora a la anterior, Adagrad, y lo que hace es introducir promedios móviles exponenciales sobre el learning rate de la siguiente forma

$$v_t = \beta v_{t-1} + (1 - \beta) \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

y actualiza

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Adam: Finalmente esta modificación es simplemente una combinación de las ideas de learning rate adaptativo RMSprop y momentum. Actualmente es una de las más utilizadas.

Observación

Hasta ahora hemos visto varias modificaciones del descenso de gradiente estocástico pero cabe destacar que no existen grandes resultados teóricos que las sustenten, es decir, corresponden a heurísticas.

Algoritmos de Optimización - Variantes del SGD

RMSprop: Esta modificación intenta ser una mejora a la anterior, Adagrad, y lo que hace es introducir promedios móviles exponenciales sobre el learning rate de la siguiente forma

$$v_t = \beta v_{t-1} + (1 - \beta) \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

y actualiza

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Adam: Finalmente esta modificación es simplemente una combinación de las ideas de learning rate adaptativo RMSprop y momentum. Actualmente es una de las más utilizadas.

Observación

Hasta ahora hemos visto varias modificaciones del descenso de gradiente estocástico pero cabe destacar que no existen grandes resultados teóricos que las sustenten, es decir, corresponden a heurísticas.

Algoritmos de Optimización - Variantes del SGD

RMSprop: Esta modificación intenta ser una mejora a la anterior, Adagrad, y lo que hace es introducir promedios móviles exponenciales sobre el learning rate de la siguiente forma

$$v_t = \beta v_{t-1} + (1 - \beta) \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

y actualiza

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Adam: Finalmente esta modificación es simplemente una combinación de las ideas de learning rate adaptativo RMSprop y momentum. Actualmente es una de las más utilizadas.

Observación

Hasta ahora hemos visto varias modificaciones del descenso de gradiente estocástico pero cabe destacar que no existen grandes resultados teóricos que las sustenten, es decir, corresponden a heurísticas.

Algoritmos de Optimización - Variantes del SGD

RMSprop: Esta modificación intenta ser una mejora a la anterior, Adagrad, y lo que hace es introducir promedios móviles exponenciales sobre el learning rate de la siguiente forma

$$v_t = \beta v_{t-1} + (1 - \beta) \frac{\partial J}{\partial \theta_t} * \frac{\partial J}{\partial \theta_t}; \quad v_0 = 0$$

y actualiza

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} * \frac{\partial J}{\partial \theta_t}$$

Adam: Finalmente esta modificación es simplemente una combinación de las ideas de learning rate adaptativo RMSprop y momentum. Actualmente es una de las más utilizadas.

Observación

Hasta ahora hemos visto varias modificaciones del descenso de gradiente estocástico pero cabe destacar que no existen grandes resultados teóricos que las sustenten, es decir, corresponden a heurísticas.

Redes Neuronales Convolucionales

Las **redes neuronales convolucionales** (o **CNNs**) son un tipo de redes neuronales que fueron diseñadas para procesar datos con una tipología tipo *grid* (grilla). Una serie de tiempo que tiene observaciones en intervalos regulares de tiempo se puede pensar como un *grid* de 1 dimensión. Las imágenes se pueden pensar como *grids* de 2-D de pixeles. El nombre de esta arquitectura hace referencia a que usan una operación matemática conocida como convolución.

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

En el contexto de CNNs, el primer argumento a convolucionar, x , es el **input**, y el segundo argumento, w , se conoce como el **kernel**. Usualmente, el **kernel** es un arreglo multidimensional de parámetros entrenables de menor dimensionalidad que el input y que moviéndose a través de este según un valor de *stride*, extrae las *features* de alto nivel (como bordes por ejemplo).

Redes Neuronales Convolucionales

Las **redes neuronales convolucionales** (o **CNNs**) son un tipo de redes neuronales que fueron diseñadas para procesar datos con una tipología tipo *grid* (grilla). Una serie de tiempo que tiene observaciones en intervalos regulares de tiempo se puede pensar como un *grid* de 1 dimensión. Las imágenes se pueden pensar como *grids* de 2-D de pixeles. El nombre de esta arquitectura hace referencia a que usan una operación matemática conocida como convolución.

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

En el contexto de CNNs, el primer argumento a convolucionar, x , es el **input**, y el segundo argumento, w , se conoce como el **kernel**. Usualmente, el **kernel** es un arreglo multidimensional de parámetros entrenables de menor dimensionalidad que el input y que moviéndose a través de este según un valor de *stride*, extrae las *features* de alto nivel (como bordes por ejemplo).

Redes Neuronales Convolucionales

Las **redes neuronales convolucionales** (o **CNNs**) son un tipo de redes neuronales que fueron diseñadas para procesar datos con una tipología tipo *grid* (grilla). Una serie de tiempo que tiene observaciones en intervalos regulares de tiempo se puede pensar como un *grid* de 1 dimensión. Las imágenes se pueden pensar como *grids* de 2-D de pixeles. El nombre de esta arquitectura hace referencia a que usan una operación matemática conocida como convolución.

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

En el contexto de CNNs, el primer argumento a convolucionar, x , es el **input**, y el segundo argumento, w , se conoce como el **kernel**. Usualmente, el **kernel** es un arreglo multidimensional de parámetros entrenables de menor dimensionalidad que el input y que moviéndose a través de este según un valor de *stride*, extrae las *features* de alto nivel (como bordes por ejemplo).

Redes Neuronales Convolucionales

Las **redes neuronales convolucionales** (o **CNNs**) son un tipo de redes neuronales que fueron diseñadas para procesar datos con una tipología tipo *grid* (grilla). Una serie de tiempo que tiene observaciones en intervalos regulares de tiempo se puede pensar como un *grid* de 1 dimensión. Las imágenes se pueden pensar como *grids* de 2-D de pixeles. El nombre de esta arquitectura hace referencia a que usan una operación matemática conocida como convolución.

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

En el contexto de CNNs, el primer argumento a convolucionar, x , es el **input**, y el segundo argumento, w , se conoce como el **kernel**. Usualmente, el **kernel** es un arreglo multidimensional de parámetros entrenables de menor dimensionalidad que el input y que moviéndose a través de este según un valor de *stride*, extrae las *features* de alto nivel (como bordes por ejemplo).

Redes Neuronales Convolucionales

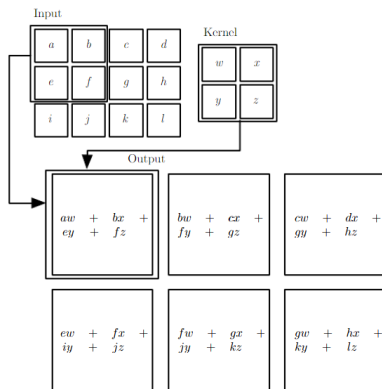


Fig.. Ejemplo de una convolución 2-D, el valor de *stride* es (1,1)
(Goodfellow y cols., 2016)

Una vez extraído el mapa de características (*feature map*) mediante la utilización de variados kernel según se requiera, este es aplanado a una dimensión y sirve como input para una *feedforward neural network* que se encargará de entregar el output correspondiente.

Redes Neuronales Convolucionales

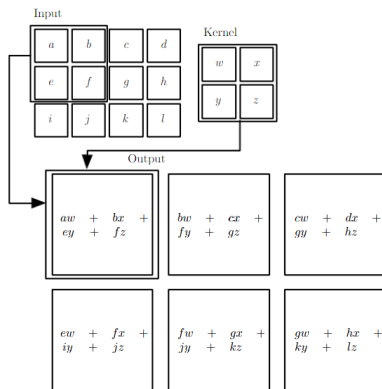


Fig.. Ejemplo de una convolución 2-D, el valor de *stride* es (1,1)
(Goodfellow y cols., 2016)

Una vez extraído el mapa de características (*feature map*) mediante la utilización de variados kernel según se requiera, este es aplanado a una dimensión y sirve como input para una *feedforward neural network* que se encargará de entregar el output correspondiente.

Redes Neuronales Recurrentes

Las **redes neuronales recurrentes** (o **RNNs**) son una familia de modelos especializados en procesamiento de datos secuenciales, $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$. Las RNNs también comparten parámetros, pero en una forma muy distinta que las CNNs. En una RNN, cada miembro del output en una etapa es una función de cada miembro del output de la etapa anterior.

Se denota por $\mathbf{h}^{(t)}$ al estado de un sistema dinámico que involucra una recurrencia conducido por un input externo $\mathbf{x}^{(t)}$:

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

Fig.. Ejemplo de una red recurrente sin output (Goodfellow y cols., 2016)

Redes Neuronales Recurrentes

Las **redes neuronales recurrentes** (o **RNNs**) son una familia de modelos especializados en procesamiento de datos secuenciales, $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$. Las RNNs también comparten parámetros, pero en una forma muy distinta que las CNNs. En una RNN, cada miembro del output en una etapa es una función de cada miembro del output de la etapa anterior.

Se denota por $\mathbf{h}^{(t)}$ al estado de un sistema dinámico que involucra una recurrencia conducido por un input externo $\mathbf{x}^{(t)}$:

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

Fig.. Ejemplo de una red recurrente sin output (Goodfellow y cols., 2016)

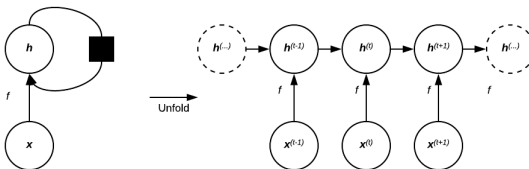
Redes Neuronales Recurrentes

Las **redes neuronales recurrentes** (o **RNNs**) son una familia de modelos especializados en procesamiento de datos secuenciales, $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$. Las RNNs también comparten parámetros, pero en una forma muy distinta que las CNNs. En una RNN, cada miembro del output en una etapa es una función de cada miembro del output de la etapa anterior.

Se denota por $\mathbf{h}^{(t)}$ al estado de un sistema dinámico que involucra una recurrencia conducido por un input externo $\mathbf{x}^{(t)}$:

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

Fig.. Ejemplo de una red recurrente sin output (Goodfellow y cols., 2016)



Redes Neuronales Recurrentes

Existen varios tipos de RNNs que se han diseñado para distintos fines.

- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre todas las unidades escondidas
- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre el output a la unidad escondida del siguiente instante
- ▶ Redes recurrentes con conexiones entre las unidad escondidas, que procesan una secuencia entera antes de producir el output

Redes Neuronales Recurrentes

Existen varios tipos de RNNs que se han diseñado para distintos fines.

- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre todas las unidades escondidas
- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre el output a la unidad escondida del siguiente instante
- ▶ Redes recurrentes con conexiones entre las unidad escondidas, que procesan una secuencia entera antes de producir el output

Redes Neuronales Recurrentes

Existen varios tipos de RNNs que se han diseñado para distintos fines.

- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre todas las unidades escondidas
- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre el output a la unidad escondida del siguiente instante
- ▶ Redes recurrentes con conexiones entre las unidades escondidas, que procesan una secuencia entera antes de producir el output

Redes Neuronales Recurrentes

Existen varios tipos de RNNs que se han diseñado para distintos fines.

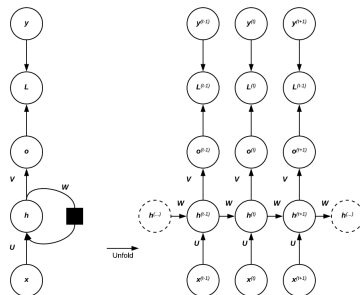
- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre todas las unidades escondidas
- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre el output a la unidad escondida del siguiente instante
- ▶ Redes recurrentes con conexiones entre las unidades escondidas, que procesan una secuencia entera antes de producir el output

Redes Neuronales Recurrentes

Existen varios tipos de RNNs que se han diseñado para distintos fines.

- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre todas las unidades escondidas
- ▶ Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre el output a la unidad escondida del siguiente instante
- ▶ Redes recurrentes con conexiones entre las unidad escondidas, que procesan una secuencia entera antes de producir el output

Fig.. Red recurrente para el primer ejemplo (Goodfellow y cols., 2016)



Autoencoders

Un **autoencoder** es una red neuronal que busca replicar el input hacia el output, es decir, busca que la información que entra a la red sea lo más parecida posible a la de salida, para lo cual cuenta con una capa interna $\mathbf{h} = f(\mathbf{x})$ (**encoder**) con menor dimensión que el input y que lo codifica (genera una representación de este) y una función que produce la reconstrucción $\mathbf{r} = g(\mathbf{h})$, el **decoder**.

Un autoencoder buscará aprender los *encoder* y *decoder* tales que $g(f(\mathbf{x})) = \mathbf{x}$ para todo \mathbf{x} . Como el modelo está forzado a aprender los atributos más importantes para que pueda efectivamente reproducir el input en su output, este aprenderá en general propiedades útiles de los datos de entrenamiento. Los *autoencoders* modernos modelan mappings estocásticos $p_{\text{encoder}}(\mathbf{h}|\mathbf{x})$ y $p_{\text{decoder}}(\mathbf{x}|\mathbf{h})$, en vez de funciones determinísticas.

Autoencoders

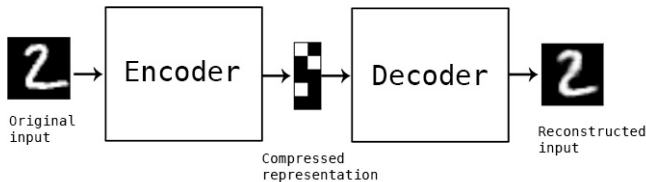
Un **autoencoder** es una red neuronal que busca replicar el input hacia el output, es decir, busca que la información que entra a la red sea lo más parecida posible a la de salida, para lo cual cuenta con una capa interna $\mathbf{h} = f(\mathbf{x})$ (**encoder**) con menor dimensión que el input y que lo codifica (genera una representación de este) y una función que produce la reconstrucción $\mathbf{r} = g(\mathbf{h})$, el **decoder**.

Un autoencoder buscará aprender los *encoder* y *decoder* tales que $g(f(\mathbf{x})) = \mathbf{x}$ para todo \mathbf{x} . Como el modelo está forzado a aprender los atributos más importantes para que pueda efectivamente reproducir el input en su output, este aprenderá en general propiedades útiles de los datos de entrenamiento. Los *autoencoders* modernos modelan mappings estocásticos $p_{\text{encoder}}(\mathbf{h}|\mathbf{x})$ y $p_{\text{decoder}}(\mathbf{x}|\mathbf{h})$, en vez de funciones determinísticas.

Autoencoders

Un **autoencoder** es una red neuronal que busca replicar el input hacia el output, es decir, busca que la información que entra a la red sea lo más parecida posible a la de salida, para lo cual cuenta con una capa interna $\mathbf{h} = f(\mathbf{x})$ (**encoder**) con menor dimensión que el input y que lo codifica (genera una representación de este) y una función que produce la reconstrucción $\mathbf{r} = g(\mathbf{h})$, el **decoder**.

Un autoencoder buscará aprender los *encoder* y *decoder* tales que $g(f(\mathbf{x})) = \mathbf{x}$ para todo \mathbf{x} . Como el modelo está forzado a aprender los atributos más importantes para que pueda efectivamente reproducir el input en su output, este aprenderá en general propiedades útiles de los datos de entrenamiento. Los *autoencoders* modernos modelan mappings estocásticos $p_{\text{encoder}}(\mathbf{h}|\mathbf{x})$ y $p_{\text{decoder}}(\mathbf{x}|\mathbf{h})$, en vez de funciones determinísticas.



Redes Generativas Adversariales

Una **red generativa adversarial** (o **GAN**) se basa en un escenario de teoría de juegos, en donde una **red generadora** debe competir con un adversario. La red generadora produce muestras $\mathbf{x} = g(\mathbf{z}; \boldsymbol{\theta}^{(g)})$, mientras que una **red discriminadora** trata de distinguir entre muestras obtenidas de los datos de entrenamiento y muestras generadas por la red generadora. El discriminador retorna una probabilidad, $d(\mathbf{x}; \boldsymbol{\theta}^{(d)})$, indicando la probabilidad de que \mathbf{x} sea un dato real y no uno simulado.

Para formular el aprendizaje, se describe un juego de suma cero en donde una función $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ determina el pago del discriminador, y el generador recibe $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ como pago. Así, durante el entrenamiento cada jugador intenta maximizar su propio pago, para que en convergencia se tenga

$$g^* = \operatorname{argmin}_g \max_d v(g, d)$$

Esto motiva a que el discriminador aprenda a clasificar correctamente entre muestras reales y falsas y, simultáneamente, el generador intenta engañar al clasificador para que crea que las muestras generadas son reales. En convergencia, las muestras del generador son indistinguibles de los datos reales. Una motivación del uso de GANs es que cuando $\max_d v(g, d)$ es convexa en $\boldsymbol{\theta}^{(g)}$, el procedimiento asegura la convergencia.

Redes Generativas Adversariales

Una **red generativa adversarial** (o **GAN**) se basa en un escenario de teoría de juegos, en donde una **red generadora** debe competir con un adversario. La red generadora produce muestras $\mathbf{x} = g(\mathbf{z}; \boldsymbol{\theta}^{(g)})$, mientras que una **red discriminadora** trata de distinguir entre muestras obtenidas de los datos de entrenamiento y muestras generadas por la red generadora. El discriminador retorna una probabilidad, $d(\mathbf{x}; \boldsymbol{\theta}^{(d)})$, indicando la probabilidad de que \mathbf{x} sea un dato real y no uno simulado.

Para formular el aprendizaje, se describe un juego de suma cero en donde una función $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ determina el pago del discriminador, y el generador recibe $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ como pago. Así, durante el entrenamiento cada jugador intenta maximizar su propio pago, para que en convergencia se tenga

$$g^* = \operatorname{argmin}_g \max_d v(g, d)$$

Esto motiva a que el discriminador aprenda a clasificar correctamente entre muestras reales y falsas y, simultáneamente, el generador intenta engañar al clasificador para que crea que las muestras generadas son reales. En convergencia, las muestras del generador son indistinguibles de los datos reales. Una motivación del uso de GANs es que cuando $\max_d v(g, d)$ es convexa en $\boldsymbol{\theta}^{(g)}$, el procedimiento asegura la convergencia.

Redes Generativas Adversariales

Una **red generativa adversarial** (o **GAN**) se basa en un escenario de teoría de juegos, en donde una **red generadora** debe competir con un adversario. La red generadora produce muestras $\mathbf{x} = g(\mathbf{z}; \boldsymbol{\theta}^{(g)})$, mientras que una **red discriminadora** trata de distinguir entre muestras obtenidas de los datos de entrenamiento y muestras generadas por la red generadora. El discriminador retorna una probabilidad, $d(\mathbf{x}; \boldsymbol{\theta}^{(d)})$, indicando la probabilidad de que \mathbf{x} sea un dato real y no uno simulado.

Para formular el aprendizaje, se describe un juego de suma cero en donde una función $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ determina el pago del discriminador, y el generador recibe $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ como pago. Así, durante el entrenamiento cada jugador intenta maximizar su propio pago, para que en convergencia se tenga

$$g^* = \operatorname{argmin}_g \max_d v(g, d)$$

Esto motiva a que el discriminador aprenda a clasificar correctamente entre muestras reales y falsas y, simultáneamente, el generador intenta engañar al clasificador para que crea que las muestras generadas son reales. En convergencia, las muestras del generador son indistinguibles de los datos reales. Una motivación del uso de GANs es que cuando $\max_d v(g, d)$ es convexa en $\boldsymbol{\theta}^{(g)}$, el procedimiento asegura la convergencia.

Redes Generativas Adversariales

Una **red generativa adversarial** (o **GAN**) se basa en un escenario de teoría de juegos, en donde una **red generadora** debe competir con un adversario. La red generadora produce muestras $\mathbf{x} = g(\mathbf{z}; \boldsymbol{\theta}^{(g)})$, mientras que una **red discriminadora** trata de distinguir entre muestras obtenidas de los datos de entrenamiento y muestras generadas por la red generadora. El discriminador retorna una probabilidad, $d(\mathbf{x}; \boldsymbol{\theta}^{(d)})$, indicando la probabilidad de que \mathbf{x} sea un dato real y no uno simulado.

Para formular el aprendizaje, se describe un juego de suma cero en donde una función $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ determina el pago del discriminador, y el generador recibe $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ como pago. Así, durante el entrenamiento cada jugador intenta maximizar su propio pago, para que en convergencia se tenga

$$g^* = \operatorname{argmin}_g \max_d v(g, d)$$

Esto motiva a que el discriminador aprenda a clasificar correctamente entre muestras reales y falsas y, simultáneamente, el generador intenta engañar al clasificador para que crea que las muestras generadas son reales. En convergencia, las muestras del generador son indistinguibles de los datos reales. Una motivación del uso de GANs es que cuando $\max_d v(g, d)$ es convexa en $\boldsymbol{\theta}^{(g)}$, el procedimiento asegura la convergencia.

Redes Generativas Adversariales



Fig.. Imágenes generadas por una GAN entrenada con el set de datos LSUN. (Izquierda) Imágenes de dormitorios generadas por el modelo DCGAN (imagen de Radford et al., 2015). (Derecha) Imágenes de iglesias generadas por el modelo LAPGAN (imagen de Denton et al., 2015)

Clase 19 - Redes neuronales (parte 3)

Aprendizaje de Máquinas - MA5204

Felipe Tobar

Department of Mathematical Engineering &
Center for Mathematical Modelling
Universidad de Chile

14 de marzo de 2021



UNIVERSIDAD
DE CHILE

Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep learning*. MIT Press.
(<http://www.deeplearningbook.org>)