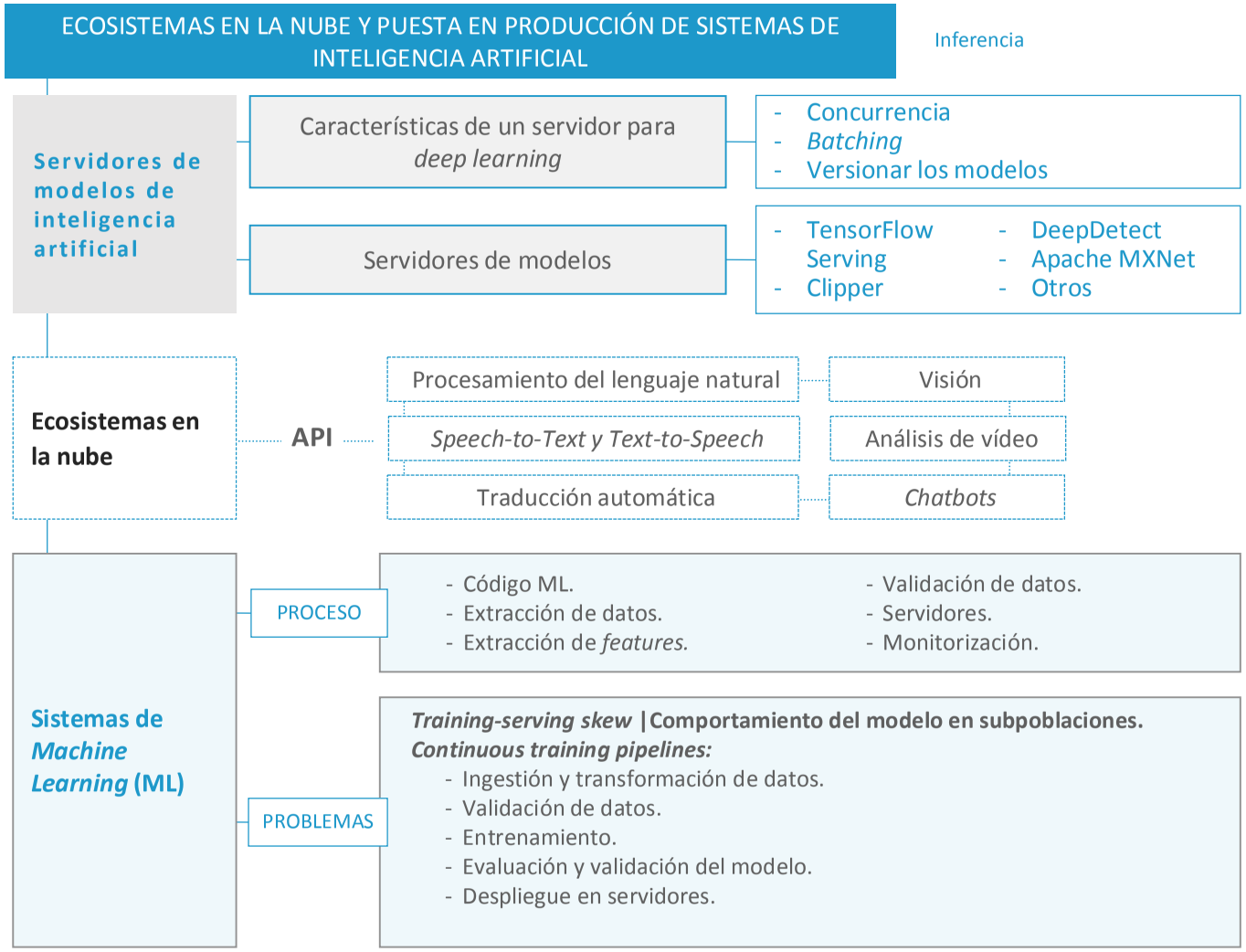
Sistemas Cognitivos Artificiales

Ecosistemas en la nube y puesta en producción de sistemas de inteligencia artificial

Índice

Esquema



Ideas clave

10.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás leer las **Ideas clave** que se desarrollan a continuación.

En este tema veremos una serie de aspectos importantes a la hora de poner en uso un sistema de *machine learning*. Esto es más complicado de lo que podría parecer y requiere la utilización de una infraestructura para servir modelos.

Es importante comprender por qué es necesario tener servidores para poner en producción un sistema de *machine learning* y qué características suelen tener estos. Asimismo, también es esencial entender qué es un ecosistema en la nube y el papel de los proveedores de *cloud computing*. Finalmente, hay que conocer los aspectos de la puesta en producción de un sistema de aprendizaje automático y cuáles son algunos de los problemas más comunes a los que hay que estar atento.

10.2. Servidores de modelos de inteligencia artificial

Hasta ahora nos hemos centrado en el entrenamiento de redes neuronales profundas. Sin embargo, otro aspecto fundamental es utilizar estos modelos para **inferencia**, esto es, utilizar los modelos ya entrenados para extraer conocimiento a partir de nuevos datos. El proceso de utilizar datos no vistos durante el entrenamiento en un modelo es comúnmente llamado inferencia o

inference (a diferencia del entrenamiento/aprendizaje), *serving time* (a diferencia de *training time*) y, en ocasiones, *test time* o *prediction time*.

Durante el proceso de inferencia, alimentamos la red neuronal con datos y recogemos la salida de la red para extraer la información deseada. Por ejemplo, en una red convolucional que clasifique imágenes, podemos obtener la clase resultante a partir de las probabilidades de la última capa.

Utilizar una red para inferencia es una tarea computacionalmente menos costosa que el aprendizaje. Al fin y al cabo, no tenemos que calcular *backpropagation* ni pasar varias veces por los mismos datos hasta converger. Basta con utilizar el *data point* a partir del cual queremos inferir y realizar un *forward pass* para obtener la salida de la red. Frameworks como TensorFlow o librerías como Keras permiten obtener los valores de salida directamente a partir de un modelo entrenado con una sola llamada.

Sin embargo, esto no significa que el proceso de inferencia no tenga sus desafíos y complejidades. El modelo se entrena una vez, pero podría ser utilizado miles o millones de veces con cientos o miles de accesos cada segundo. Imaginemos una red neuronal en un servicio de música o vídeo, calculando recomendaciones para sus usuarios, o un detector de personas que sugiere a quién etiquetar en una foto en una red social. De este modo, la puesta en producción de un modelo de inteligencia artificial presenta varias complicaciones en el plano ingenieril.

Características de un servidor para *deep learning*

Concurrencia

Una propiedad que nos gustaría satisfacer al servir un modelo de manera escalable es la concurrencia, esto es, poder **atender múltiples *requests*** (llamadas a nuestro modelo) a la vez. Normalmente, y dependiendo de la complejidad y tamaño del

modelo, los tiempos para responder una solicitud no deberían ser mayores de medio segundo e, idealmente, deberían bajar de 100ms.

Los sistemas de *machine learning* son, en muchas ocasiones, complementarios a otras partes de un servicio (por ejemplo, las recomendaciones de artículos similares que podemos ver en una tienda *online* son una pequeña parte de toda la información que se nos presenta en pantalla) y, por tanto, no deberían de suponer un cuello de botella que retrasase la respuesta al usuario. De este modo, si nuestro servidor solo permite atender una llamada o *request* al mismo tiempo, las llamadas se pueden ir acumulando y el último usuario en llegar tendría que esperar a que todos los anteriores sean respondidos. Esto daría lugar a un problema de **tiempos de respuesta**, en el que el *round-trip time* de una solicitud se alarga demasiado.

Un servidor concurrente permite tratar varias respuestas a la vez. Esto se consigue habitualmente mediante el uso de varios procesos o hilos, los cuales están a la espera de recibir y tratar *requests* según van llegando. Al llegar una, si un proceso está ocupado, otro proceso libre puede encargarse de la nueva solicitud, de modo que esta no tiene que esperar a la siguiente.

Batching

Otra característica deseable en un servidor para modelos de *deep learning* es el *batching*. Durante el curso, hemos visto el entrenamiento de redes neuronales utilizando *batches* de *training points* para el algoritmo SGD. De hecho, hacer un *batch* o lote de ejemplos no solo tiene un sentido matemático como aproximación al gradiente real, sino que suele **acelerar el entrenamiento de las redes neuronales** si se trata de manera adecuada.

Como sabemos, una red neuronal es calculada a través de multiplicaciones de matrices y operaciones optimizadas en forma vectorial. De este modo, si en vez de calcular las operaciones de la red, elemento a elemento, calculamos a la vez todos los elementos de la *batch* (pongamos el caso de 128), vamos a ganar velocidad al

aprovechar operaciones optimizadas como la multiplicación de matrices. Esto es especialmente importante combinado con el uso de GPU que, como vimos en el tema anterior, aceleran en gran medida estas operaciones. Sin embargo, en inferencia las *requests* que recibimos, suelen ser elemento a elemento. ¿Cómo podemos aprovechar las ventajas computacionales del *batching*?

La solución consiste en ir guardando las *requests* que recibimos en un *buffer* y, cuando tengamos un número suficiente de ellas, crear una *batch* y pasarla por nuestro modelo, obteniendo una salida con todos los valores deseados. Si, además, utilizamos una GPU en nuestro servidor, la aceleración será aún mayor.

Por supuesto, el *batching* solo es útil durante *serving time* y si recibimos un gran número de *requests* por segundo, ya que si solo llega una cada varios segundos, no tiene sentido dejarla esperando.

Si es necesario escalar aún más el servicio, se puede recurrir a utilizar varios servidores a la vez. En este caso, todos los servidores sirven el mismo modelo, con un ***load balancer* o balanceador de carga** encargándose de dirigir las *requests* a cada uno de ellos según su carga de trabajo.

Versionar los modelos

Finalmente, otra característica importante en un servidor es la posibilidad de versionar los modelos disponibles. De este modo, si lo mejoramos entrenándolo con nuevos datos o con una arquitectura distinta, es posible subir el modelo con una nueva versión permitiendo, además, hacer un *rollback* al modelo anterior si el nuevo no está funcionando como esperábamos.

Servidores de modelos

El uso generalizado del *deep learning* en los últimos años ha hecho surgir una serie de servidores para modelos de *machine learning* que incorporan las características

comentadas anteriormente, así como otros aspectos que facilitan la puesta en producción de modelos entrenados con frameworks como TensorFlow, Caffe, Keras, etc. La idea de estos «servidores de modelos» es sencilla: el desarrollador aporta el modelo salvado en disco (por ejemplo, un fichero conteniendo un modelo en formato de TensorFlow) y el servidor se encarga de responder las *requests* de una manera escalable y eficiente.

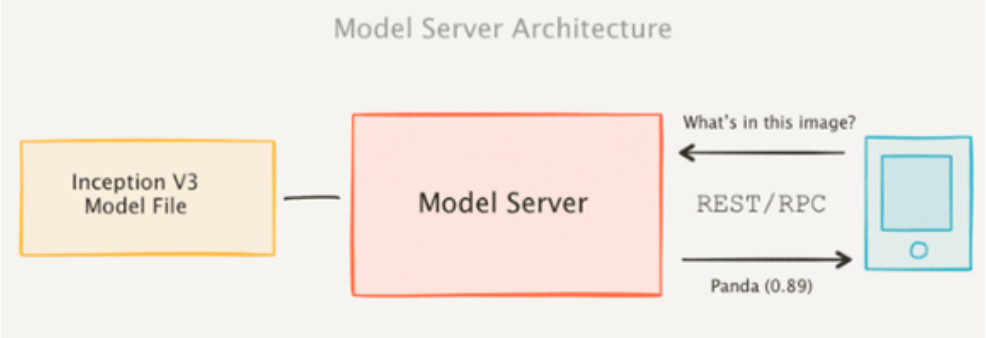


Figura 1. Servidor de modelos.
Fuente: https://medium.com/@vikati/the-rise-of-the-model-servers-9395522b6c58

TensorFlow Serving es el servidor oficial de TensorFlow. Forma parte del ecosistema de este y, si bien se puede generalizar a otros formatos, aporta grandes facilidades para poner en producción modelos de este framework. Incorpora interfaces gRPC y REST para responder las *requests* y es utilizado en Google para muchos sistemas en producción.

Clipper es otro servidor para modelos de *machine learning*, desarrollado en este caso en la universidad de Berkeley. No impone el uso de ningún framework y favorece el desarrollo en *containers*, facilitando la puesta en producción de modelos. La comunicación con el servidor es a partir de una interfaz REST.

Otros servidores disponibles son **DeepDetect** y **Model Server for Apache MXNet**.

10.3. Ecosistemas en la nube

Antes de la irrupción de los servicios en la nube, cuando una compañía quería poner en marcha un servicio en la red, tenía que adquirir normalmente el hardware necesario para ello y tener una su propia sala de servidores. Las empresas de *hosting* se dedicaban básicamente a servir páginas webs sencillas.

Todo eso cambió en 2006 cuando Amazon presentó **Amazon Web Services (AWS)**. AWS es una plataforma de servicios en la nube que admite el desarrollo de soluciones informáticas. En particular, AWS ofrece capacidad de cómputo (servidores), bases de datos, almacenamiento masivo de datos y una larga lista de herramientas y recursos IT que permiten a una empresa poner en funcionamiento complejos sistemas informáticos sin tener que preocuparse del hardware para llevarlo a cabo.

AWS y otros ecosistemas en la nube han revolucionado el desarrollo de soluciones informáticas. Con la computación en la nube, ya no es necesario instalar y mantener una costosa infraestructura informática, sino que basta con pagar por los recursos que se necesitan bajo demanda, lo que facilita también escalar los servicios informáticos a miles o millones de usuarios de una manera más sencilla. Si necesitamos más servidores para ofrecer un producto, no es necesario tener que instalar un *datacenter* mayor: es suficiente con pagar el uso de nuevas máquinas a nuestro proveedor en la nube.

Los mayores proveedores de computación en la nube en la actualidad son AWS, Google Cloud Platform (GCP) y Microsoft Azure. Como no podía ser de otra manera, ante el ascenso de la inteligencia artificial, estos están empezando a ofrecer un amplio abanico de soluciones de computación para esta área. Una muestra es la posibilidad de contratar máquinas específicas con GPU para el entrenamiento y predicción de modelos. Algunos proveedores como Google ofrecen incluso hardware

específico para el entrenamiento de modelos de *deep learning*, como las TPU (*Tensor Processing Unit*).

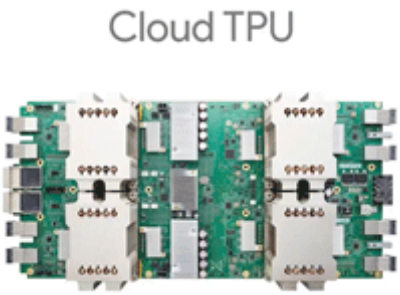


Figura 2. TPU.
Fuente: https://youtu.be/78P0pBj-i4c

Del mismo modo, los ecosistemas en la nube ofrecen una serie de **API para resolver tareas comunes** en el mundo del *machine learning*:

- Procesamiento del lenguaje natural: problemas sobre textos como análisis de sentimiento, extracción de entidades, *POS tagging*…
- *Speech-to-Text* y *Text-to-Speech*: convertir audio a texto y texto a audio, respectivamente.
- Traducción automática: traducción de textos.
- Visión: clasificación de imágenes, detección de objetos y caras en imágenes o extracción de texto desde imágenes, entre otros.
- Análisis de vídeo: tareas como extracción de contenidos en vídeos y etiquetado.
- Chatbots: frameworks de *machine learning* para el desarrollo de asistentes conversacionales, especialmente alrededor de la detección de la intención del usuario.

Este tipo de API resuelven problemas muy comunes y para los cuales una solución general puede dar buen resultado. Por ejemplo, la traducción automática es un problema definido de manera muy clara, para el cual se necesita una gran cantidad de datos y una labor compleja de refinamiento de los modelos. Recopilar todos estos datos y entrenar un modelo propio sería demasiado caro y complicado para algunas

de estas tareas, por lo que en ocasiones merece la pena recurrir a estas soluciones «*off-the-shelf*».

10.4. Aspectos prácticos de la puesta en producción de sistemas de *machine learning*

La puesta en producción de un sistema de aprendizaje automático es un proceso muy complejo. De hecho, la parte del sistema que corresponde puramente al código de ML es una pequeña parte de todo el proceso, como se ve en la siguiente imagen:

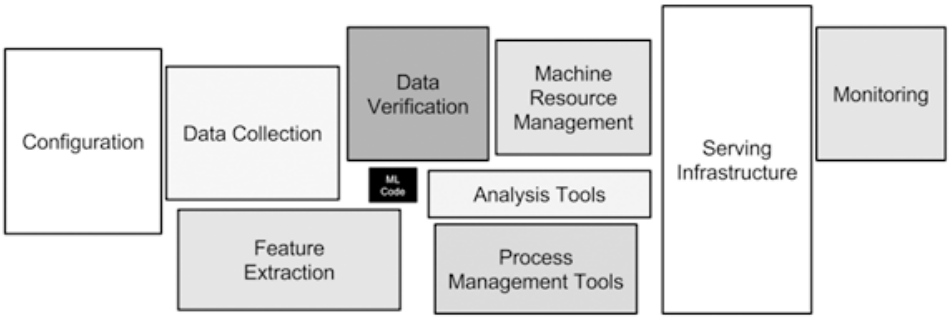


Figura 3. Sistema de aprendizaje automático.
Fuente: https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

En la imagen podemos ver una serie de aspectos necesarios para implementar un sistema de *machine learning*:

- Una parte importante es la **recolecta** de datos, así como la **extracción** de *features* de estos datos y una posterior **validación** de los mismos.

Antes de implementar nuestros modelos y entrenarlos, es necesario saber qué datos necesitamos y cuáles de ellos tienen sentido para la tarea que queremos resolver. Este proceso puede volverse extraordinariamente complejo.

Por ejemplo, podríamos necesitar interacciones de usuarios de una red social: qué *likes* se han dado, quién es amigo de quién, qué tipos de vídeos y páginas ha visto un usuario, etc. Estos datos pueden estar guardados en formato de *logs* o en distintas bases de datos, de modo que se hace necesario escribir código para extraerlos y tratarlos de modo que sean útiles para nuestro algoritmo. Igualmente, es posible que los datos que necesitamos no estén siendo recopilados hasta el momento, por lo que sería necesario trabajar en conseguir que esos datos se capturen en nuestro sistema.

- Asimismo, como hemos visto en los apartados anteriores de este tema, es necesario contar con una **infraestructura para entrenar y servir nuestros modelos**, para lo cual podemos utilizar un proveedor de *cloud computing* y un servidor de modelos, como hemos visto en los apartados anteriores.

- Del mismo modo, es importante contar con una serie de elementos de **monitorización y análisis** del funcionamiento de nuestro sistema. Estos guardan datos acerca de cuántas *requests* se reciben, cuál es el tiempo de respuesta de estas, etc. El sistema de monitorización debería de alertar cuando los servidores están caídos o cuando el tiempo de respuesta es demasiado alto, lo cual nos permite actuar a tiempo para evitar que nuestro sistema deje de funcionar o que los usuarios reciban un servicio lento y con errores.

Finalmente, hay una serie adicional de aspectos prácticos más propios del aprendizaje automático que conviene tener en cuenta y que pueden hacer que nuestro sistema en producción no funcione como esperábamos. Veremos dos de ellos: el *training-serving skew* y el comportamiento del modelo en subpoblaciones.

Training-serving skew

Training-serving skew es el problema que viene dado por una **degradación del rendimiento** entre el entrenamiento del modelo y su uso para inferencia. Es muy típico entrenar un modelo y ver que todas nuestras métricas de entrenamiento (métricas *offline* como *accuracy*, *precision*, *recall*, AUC, etc.) son muy buenas, por lo

que ponemos a nuestro modelo a funcionar en entornos reales con tranquilidad. Sin embargo, al comprobar con detalle lo que el modelo está haciendo en el sistema en producción, nos damos cuenta de que este no está funcionando tan bien como nos parecía. Hay varias **razones** por las que esto ocurre:

- Una de ellas es la existencia de una discrepancia entre **cómo se extraen los datos** para entrenar y los datos que el modelo recibe durante *serving*. Es común tener caminos de código distintos para generar estos datos, lo que puede dar lugar a que ciertas *features* sean calculadas de manera un poco distinta o que, en ocasiones, directamente no aparezcan. ¿Se han normalizado los valores de una *feature* de la misma manera a la hora de entrenar que a la hora de servir? ¿Estamos seguros de que la edad del usuario está disponible a la hora de servir, tal y como estaba durante el entrenamiento? El hecho de que nuestro algoritmo reciba una distribución de datos muy diferente con la que fue entrenado, o que incluso algunos datos presentes durante el entrenamiento desaparezcan, puede provocar que el modelo se vuelva inestable y su rendimiento disminuya en gran medida, haciendo incluso que los resultados sean totalmente erróneos.
  - Una solución a este problema consiste en asegurarse de que el código utilizado es el mismo o es totalmente equivalente a la hora de extraer los datos para entrenamiento e inferencia.
  - Otra forma de hacer esto es, en el caso de que sea posible, recopilar los datos de entrenamiento para futuros modelos a la hora de servir. Si nuestro sistema guarda los datos que ve durante *serving*, esos valores se pueden utilizar luego para entrenar el modelo y podemos, por tanto, estar seguros de que los datos son los adecuados.

- Otra razón por la que aparece el *training-serving skew* es por utilizar directamente **distribuciones de datos distintas** al entrenar y al servir. Por ejemplo, podemos entrenar un modelo con datos de 2017, pero si lo utilizamos en 2020 es muy posible que los datos que vea este modelo sean de una naturaleza distinta: un sistema de recomendación de películas entrenado en 2017 no sabrá nada acerca de los nuevos estrenos de 2020. En este caso, estamos ante distribuciones de

datos distintas, así, nuestro modelo no será capaz de realizar predicciones razonables. De manera similar, si entrenamos un modelo con una subpoblación de usuarios (por ejemplo, usuarios españoles), podríamos encontrarnos con un *training-serving skew* si lo utilizamos con usuarios alemanes, cuya distribución de datos puede presentar diferencias en nuestra aplicación.

Comportamiento del modelo en subpoblaciones

Otro problema que podemos encontrarnos en la práctica es que un modelo se comporte de manera **muy inestable** para distintos subconjuntos de los datos. Esto puede ser especialmente crítico en el caso de que nuestro modelo trate con usuarios.

> **Ejemplo ilustrativo**
>
> Imaginemos el siguiente caso. Ponemos en producción un nuevo sistema de recomendación de canciones con buenas métricas de entrenamiento y que también está consiguiendo buenas métricas *online* para nuestros usuarios, esto es, las métricas que tenemos para medir el rendimiento del modelo en producción están siendo buenas, por ejemplo, que el tiempo de escucha está aumentando. Sin embargo, pronto empezamos a recibir quejas de usuarios acerca de que sus recomendaciones son mucho peores repentinamente.
>
> Puede que las métricas hayan mejorado en general y la mayoría de usuarios estén recibiendo una mejor experiencia, pero a costa de que un pequeño subconjunto de usuarios haya empeorado en gran medida. Estos podrían ser usuarios de cierta zona geográfica, de una edad o género en particular…, incluso podrían ser los usuarios que pagan por el servicio, lo cual sería desastroso para el producto.

Este problema es complicado de tratar y requiere contar con herramientas que nos permitan detectarlo, idealmente, antes de poner el sistema en funcionamiento.

Continuous training pipelines

Otra complicación que surge al poner sistemas de *machine learning* en producción es la periodicidad con la que necesitamos actualizar nuestro modelo. En ocasiones, es suficiente con entrenar un modelo y comprobar que funciona correctamente. Por ejemplo, un clasificador de imágenes de razas de perro puede ser implementado una vez y no necesitar grandes mejoras en el futuro, ya que lo más probable es que las razas de perro no cambien mucho en apariencia en el futuro cercano.

Sin embargo, y relacionado con lo que hemos visto con *training-serving skew*, ciertos datos tienen una **importante componente temporal**. Por ejemplo, un sistema de recomendación de vídeos y canciones tiene que mantenerse actualizado con los hábitos actuales de los usuarios. O, de manera similar, un clasificador de documentos que detecta *spam* tiene que estar al corriente de los nuevos tipos de *spam* que van apareciendo.

Esto implica que, en muchas ocasiones, no vale con entrenar el modelo una vez y olvidarse del asunto, sino que se necesita recopilar datos actualizados y reentrenar el modelo cada cierto tiempo para que se mantenga actualizado. Llega un momento en el que la periodicidad necesaria para hacer esto (podría llegar a ser semanal o incluso diaria) implica que este sistema tenga que automatizarse. Esto es lo que se conoce como una *continuous training pipeline* y conlleva un nuevo nivel de complejidad técnica a tener en cuenta, ya que se hace **necesario automatizar** todos el proceso de entrenamiento, evaluación, *deployment* a un servidor…

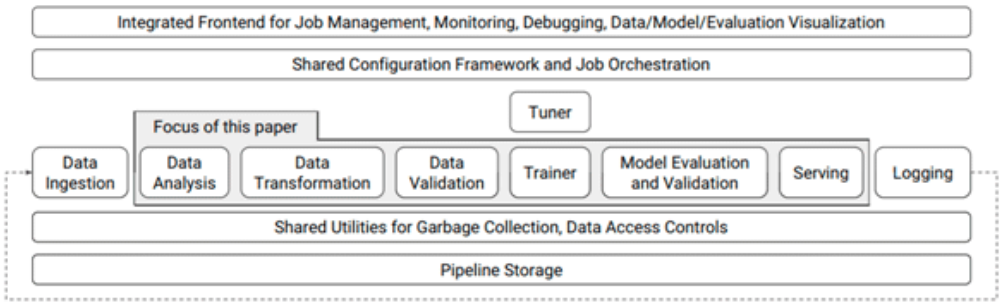


Figura 4. *Continuous training pipeline*.
Fuente: https://dl.acm.org/citation.cfm?id=3098021

Una *continuous training pipeline* consta de varias partes, entre las cuales se puede encontrar, dependiendo del caso en particular:

Ingestión y transformación de datos. Componente del sistema que se encarga de obtener los datos de diversas procedencias y hacer las transformaciones necesarias para su uso en modelos de *machine learning*.

Validación de datos. Se encarga de, una vez tenemos los datos extraídos y listos para usar, realizar comprobaciones para asegurarnos de que esos datos son correctos. Por ejemplo, comprobar que no faltan *features* o asegurarnos de que los datos siguen cierta distribución en consonancia con ejecuciones anteriores de la *training pipeline*. Este componente puede parar la ejecución de la *pipeline* si se detectan anomalías.

Entrenamiento. Proceso de entrenamiento del modelo, potencialmente con *hyperparameter tuning* según métricas *offline*.

Evaluación y validación del modelo. Componente que se encarga de evaluar y validar el correcto funcionamiento del modelo. Primero, asegurándose de que este puede ser servido y no da lugar a errores de ejecución y, segundo, validando ciertas métricas *offline*. Por ejemplo, asegurándose de que la *accuracy* se mantiene en unos márgenes similares a los de ejecuciones anteriores. En esta parte podría medirse también el comportamiento del modelo en varias

subpoblaciones de importancia, deteniendo la ejecución si encontramos anomalías.

Despliegue del modelo en servidores. Envío del modelo a los servidores de inferencia y puesta final en producción.

Para una mayor seguridad, una vez el nuevo modelo está en funcionamiento, se podría tener un componente midiendo factores como el *training-serving skew*.

Lo + recomendado

Lecciones magistrales

Puesta en producción de un sistema de inteligencia artificial

Como indica el título, en esta sesión veremos la puesta en producción de un sistema de inteligencia artificial y se hará hincapié en cómo evaluar que este sistema está funcionando correctamente.

Accede a la lección magistral a través del aula virtual

No dejes de leer

Reglas de aprendizaje automático: recomendaciones para la ingeniería de aprendizaje automático

Zinkevich, M. (Actualizado 16 de mayo de 2018). *Reglas de aprendizaje automático: recomendaciones para la ingeniería de aprendizaje automático* [En línea].

Documento muy interesante de Google con recomendaciones y buenas prácticas a la hora de implementar un sistema de *machine learning*.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:
https://developers.google.com/machine-learning/rules-of-ml/

TFX: A TensorFlow-Based Production-Scale Machine Learning Platform

Baylor, D. (2017). TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. En *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1387-1395). New York, Estados Unidos: ACM.

Explicación de TFX, una plataforma para poner en producción modelos en TensorFlow que puede funcionar como *continuous training pipeline*.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:
https://dl.acm.org/citation.cfm?id=3098021

+ Información

Webgrafía

TensorFlow Serving

Página oficial de TensorFlow Serving, utilizado en Google para muchos sistemas en producción. En esta página podrás encontrar documentación oficial, guías, tutoriales, etc.

Accede a la página web a través del aula virtual o desde la siguiente dirección:
https://www.tensorflow.org/serving/

Clipper

Clipper, como hemos visto en este tema, es otro servidor para modelos de *machine learning*, desarrollado por la universidad de Berkeley. En esta página encontrarás documentación oficial, API, Github, guías, etc.

Accede a la página web a través del aula virtual o desde la siguiente dirección:
http://clipper.ai/

Bibliografía

Sculley et al. (2015). Hidden Technical Debt in Machine Learning Systems. En C. Cortes, N.D. Lawrence, D.D. Lee, M. Sugiyama y R. Garnett (Eds.). *Advances in Neural Information Processing Systems 28*. Recuperado de https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

Test

1. Hacer *batching* en *serving time* es deseable porque (marca la respuesta correcta):

A. Permite ejecutar varias *requests* a la vez.

B. Permite la utilización de varios servidores concurrentes para ejecutar una *batch* más rápido.

C. En sistemas con un gran número de *requests* por segundo permite combinar varias a la vez y calcular la salida más rápido que si se fuera una a una.

2. Durante el proceso de inferencia se realiza (marca la respuesta correcta):

A. Solo el *forward pass* de una red.

B. Solo el *backward pass* de una red.

C. Ambos, *forward* y *backward passes*.

D. Ninguno de ellos.

3. Un servidor con concurrencia (marca las respuestas correctas):

A. Explota las capacidades multihilo de los procesadores y permite tratar varias *requests* a la vez.

B. Permite reducir el tiempo de respuesta ante la presencia de solicitudes concurrentes.

C. Permite aumentar el número total de *requests* por segundo que pueden ser respondidas.

D. No aporta grandes ventajas si el servidor recibe una *request* cada minuto.

4. Un balanceador de carga (marca la respuesta correcta):

A. Balancea el versionado de modelos en un conjunto de servidores distribuidos.

B. Almacena *requests* y las envía en forma de *batch* a un servidor.

C. Mejora el tiempo de respuesta mediante la aplicación de concurrencia en un servidor.

D. Ninguna de las anteriores.

5. Un ecosistema en la nube tipo AWS o GCP (marca la respuesta correcta):

A. Facilita el desarrollo de soluciones informáticas permitiendo que los desarrolladores no tengan que centrarse en el código.

B. Facilita el desarrollo de soluciones informáticas, abstrayendo a los desarrolladores de la complejidad de instalar y mantener complejos sistemas de hardware.

C. Suele ser más caro que instalar y mantener tu propio hardware.

D. Es un servicio de *hosting* de páginas web.

6. Una API de inteligencia artificial de un proveedor en la nube (marca la respuesta correcta):

A. Puede ser útil cuando necesitamos resolver un problema específico. Por ejemplo, queremos convertir audio a texto, para lo cual existen muchas soluciones disponibles.

B. Es útil cuando tenemos una necesidad particular y compleja. Por ejemplo, queremos entrenar un modelo que clasifique imágenes de distintas series de televisión.

C. Ninguna de las anteriores.

7. ¿Cuáles de los siguientes puntos pueden hacer la extracción de datos y *features* en un proceso complejo? (marca las respuestas correctas):

A. Normalmente ninguno, la extracción de datos suele ser siempre la parte más sencilla a implementar.

B. Los datos necesarios pueden provenir de varias fuentes distintas (bases de datos, *logs*...).

C. Los datos en bruto pueden necesitar ser tratados para que sean útiles para nuestro modelo.

D. Es posible que los datos no estén siendo almacenados hasta el momento, por lo que es necesario crear esa lógica.

8. ¿Cuáles de los siguientes casos puede dar lugar a *training-serving skew*? (marca las respuestas correctas):

A. Entrenar un modelo de reconocimiento de caras solo con personas de pelo moreno y sin gafas.

B. Olvidar que la *feature* «edad del usuario» no está disponible a la hora de servir, pero se ha usado para entrenar.

C. Poner, sin querer, en producción un modelo con malas métricas de entrenamiento.

D. Entrenar un modelo de recomendación solo con los usuarios que están suscritos al servicio.

9. Una *continuous training pipeline* (marca la respuesta correcta):

A. Permite entrenar modelos de manera óptima a partir de cualquier fuente de datos.

B. Es un conjunto de componentes que evitan el *training-serving skew* y otros problemas en *machine learning*.

C. Es un sistema donde un modelo de *machine learning* es entrenado y puesto en producción de manera automática de forma periódica.

10. Una forma de evitar el problema del comportamiento inestable del modelo en subpoblaciones sería (marca la respuesta correcta):

A. A la hora de evaluar el modelo entrenado, tener datos de test separados según las subpoblaciones que nos interesan más y comprobar las métricas del modelo en cada una de estas subpoblaciones.

B. Asegurarnos de que el código que extrae los datos para el modelo es el mismo en *training* y *serving time*.

C. Comprobar que las distribuciones de los datos a la hora de entrenar y servir son las mismas.