Inteligencia Artificial II

Juan Pablo Restrepo Uribe

Ing. Biomedico - MSc. Automatización y Control Industrial

jprestrepo@correo.iue.edu.co

2023

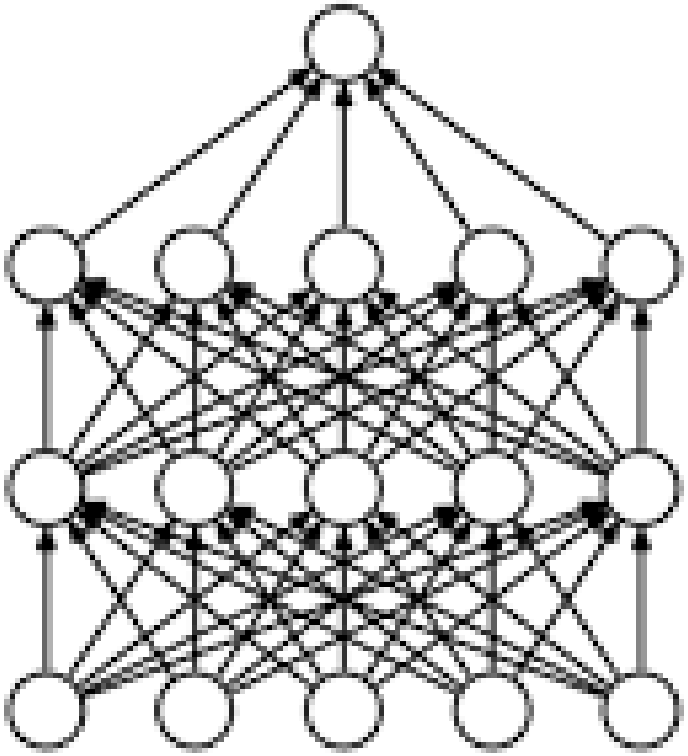
Institución Universitaria de Envigado

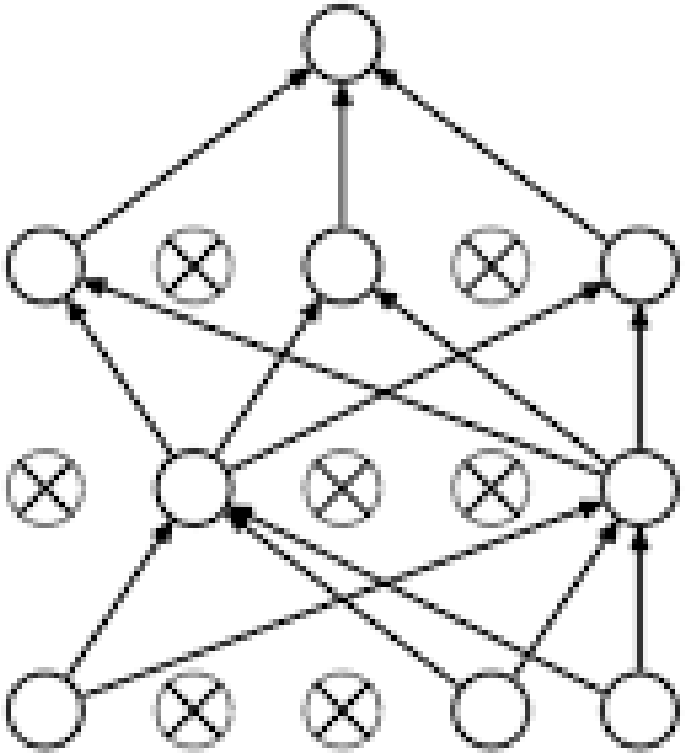
Capas

Las capas están definidas como grupos de neuronas, separadas de forma lógica en una red jerárquica. Para simplificar el modelo Keras provee muchos tipos de capas y varias formas de conectarlas. Entre las principales se encuentran

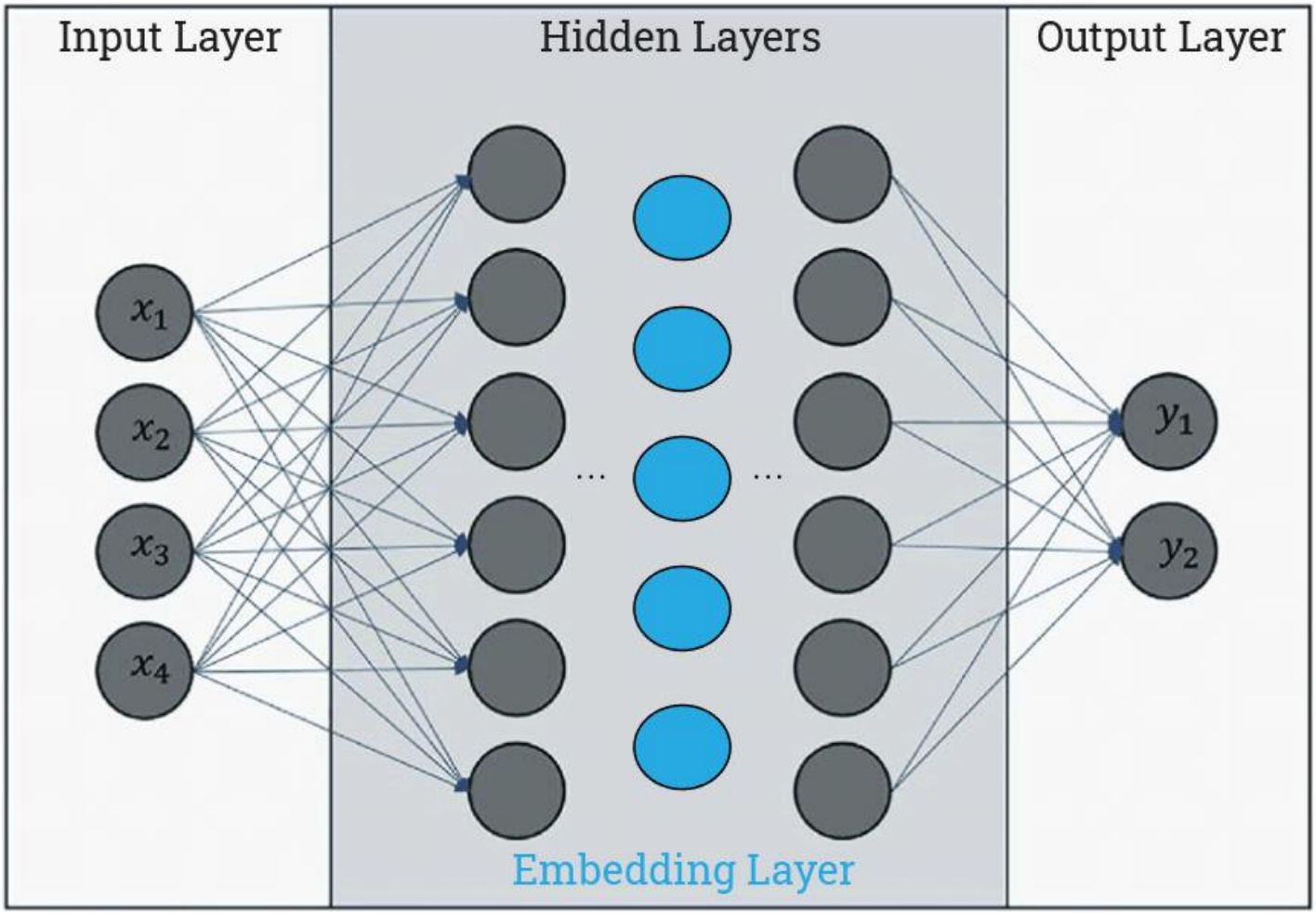
- Dense Layer: Una capa densa es una capa regular que conecta cada neurona definida en la capa, con cada neurona en la capa previa.
- Dropout Layer: Ayuda a reducir overfitting. Esta capa elimina unas pocas neuronas y reduce el calculo en el entrenamiento.
- Embedding layers / Convolutional layers / Pooling layers / etc.
- Uno puede incluso escribir su propia capa en Keras para un caso particular

Capas

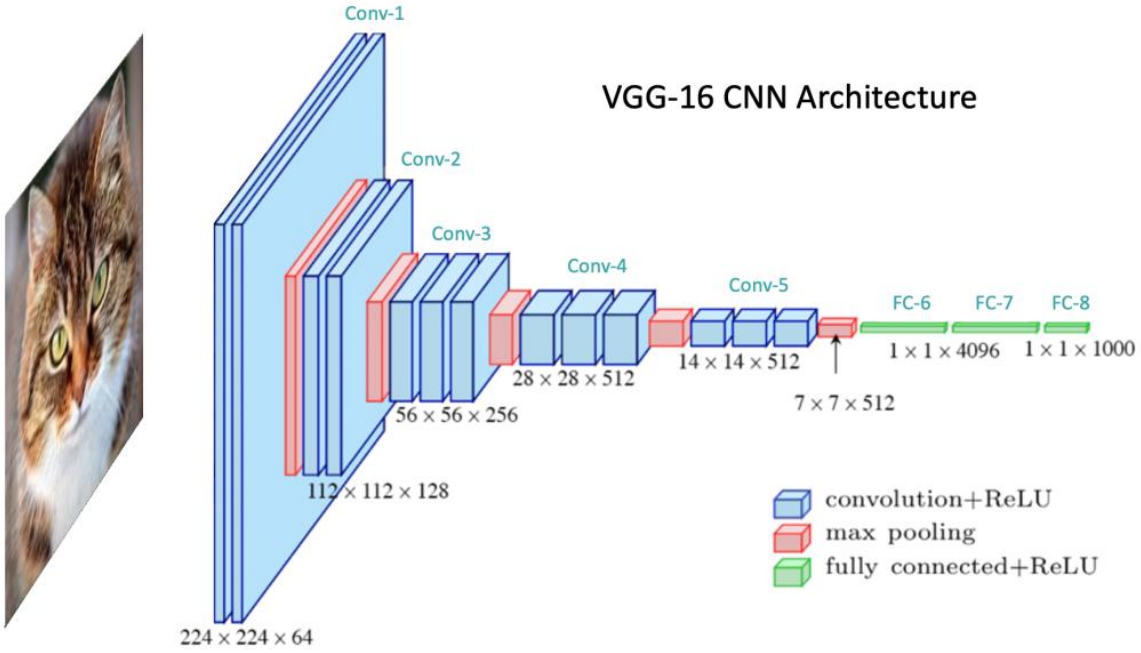




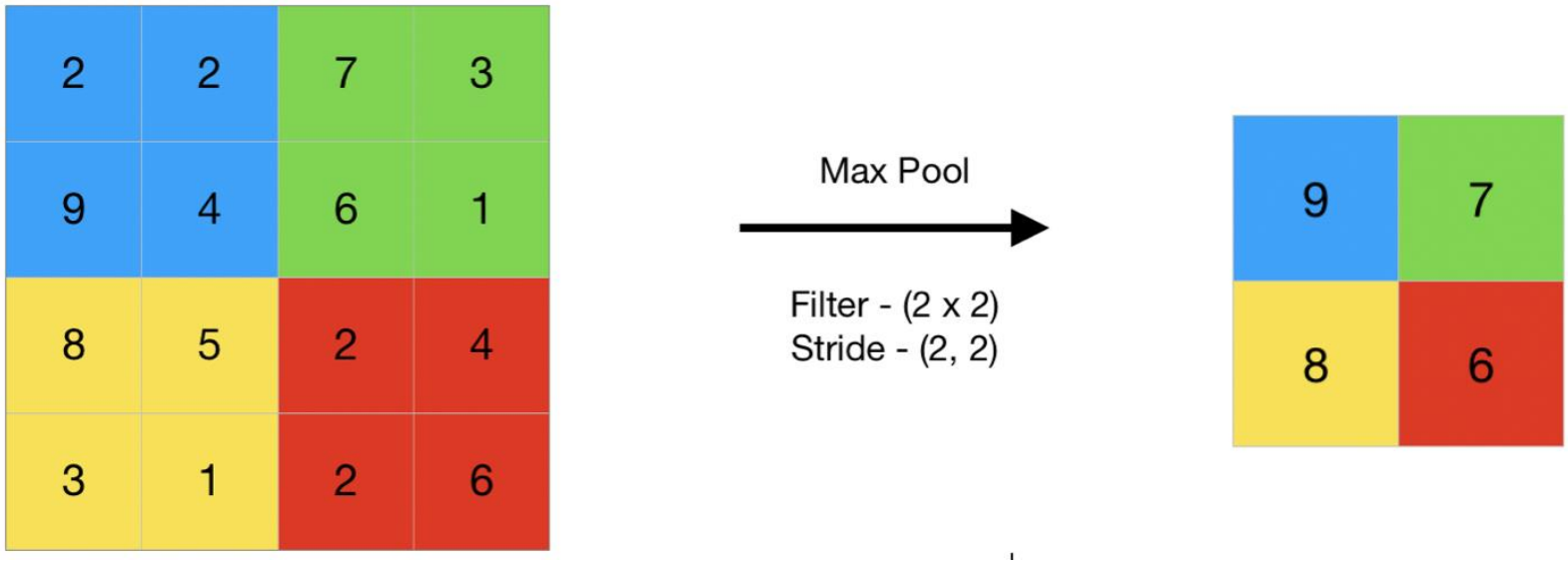
Capas


Input Layer
Hidden Layers
Output Layer
x_1
x_2
x_3
x_4
y_1
y_2
Embedding Layer

Capas


VGG-16 CNN Architecture
Conv-1
Conv-2
Conv-3
Conv-4
Conv-5
FC-6
FC-7
FC-8
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU

Capas



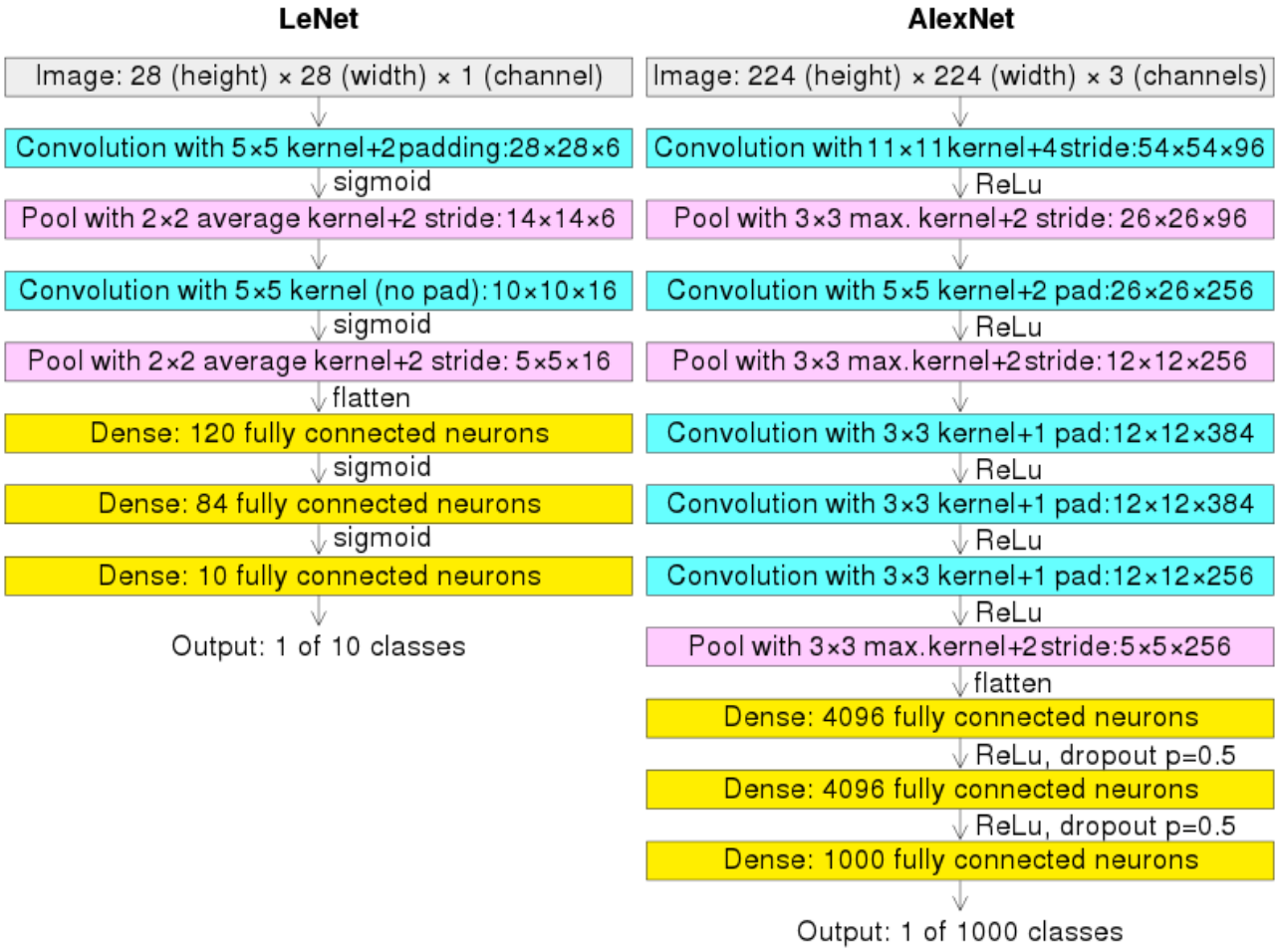
Capas

AlexNet es el nombre de una arquitectura de red neuronal convolucional (CNN), diseñada por Alex Krizhevsky en colaboración con Ilya Sutskever y Geoffrey Hinton.

AlexNet compitió en el Desafío de reconocimiento visual a gran escala de ImageNet el 30 de septiembre de 2012. La red logró un error entre los 5 primeros del 15,3%, más de 10,8 puntos porcentuales menos que el del subcampeón. El resultado principal del artículo original fue que la profundidad del modelo era esencial para su alto rendimiento, lo cual era costoso desde el punto de vista computacional, pero factible gracias a la utilización de unidades de procesamiento de gráficos (GPU) durante el entrenamiento.

https://www.image-net.org/

Capas


LeNet
Image: 28 (height) × 28 (width) × 1 (channel)
Convolution with 5×5 kernel+2 padding: 28×28×6
sigmoid
Pool with 2×2 average kernel+2 stride: 14×14×6
Convolution with 5×5 kernel (no pad): 10×10×16
sigmoid
Pool with 2×2 average kernel+2 stride: 5×5×16
flatten
Dense: 120 fully connected neurons
sigmoid
Dense: 84 fully connected neurons
sigmoid
Dense: 10 fully connected neurons
Output: 1 of 10 classes
AlexNet
Image: 224 (height) × 224 (width) × 3 (channels)
Convolution with 11×11 kernel+4 stride: 54×54×96
ReLu
Pool with 3×3 max. kernel+2 stride: 26×26×96
Convolution with 5×5 kernel+2 pad: 26×26×256
ReLu
Pool with 3×3 max. kernel+2 stride: 12×12×256
Convolution with 3×3 kernel+1 pad: 12×12×384
ReLu
Convolution with 3×3 kernel+1 pad: 12×12×384
ReLu
Convolution with 3×3 kernel+1 pad: 12×12×256
ReLu
Pool with 3×3 max. kernel+2 stride: 5×5×256
flatten
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 1000 fully connected neurons
Output: 1 of 1000 classes

La función de pérdida

Esta función es una métrica que ayuda a la red a comprender si está aprendiendo en la dirección correcta.
Supongamos que tenemos las siguientes evaluaciones en cinco pruebas consecutivas:

- 3.8
- 4.7
- 5.5
- 6.0
- 6.5

Uno podría pensar que está mejorando el rendimiento ya que las notas suben. Si las notas bajan hubiésemos pensado lo contrario. De forma similar, la red utiliza una función para evaluar su desempeño.

La función de pérdida

En esencia la función de pérdida mide cuanto nos alejamos de un objetivo.

Por ejemplo, para el caso de las notas y los estudiantes, nuestro modelo predice un puntaje de 0.87 para un alumno (1.0 aprueba / 0.0 reprueba). Si al modificar los pesos en nuestra red la predicción baja a 0.4, entonces el cambio realizado no ayuda a aprender, y debemos deshacer este cambio. En cambio, si sube a 0.9 entonces sí estaríamos aprendiendo en la dirección correcta.

Algunas funciones de pérdida

- Mean Squared Error
- Mean Absolute Error
- Mean Absolute percentage error
- Mean Square logarithmic error
- Binary-cross entropy (por categoría)
- Categorical cross-entropy (por categoría)

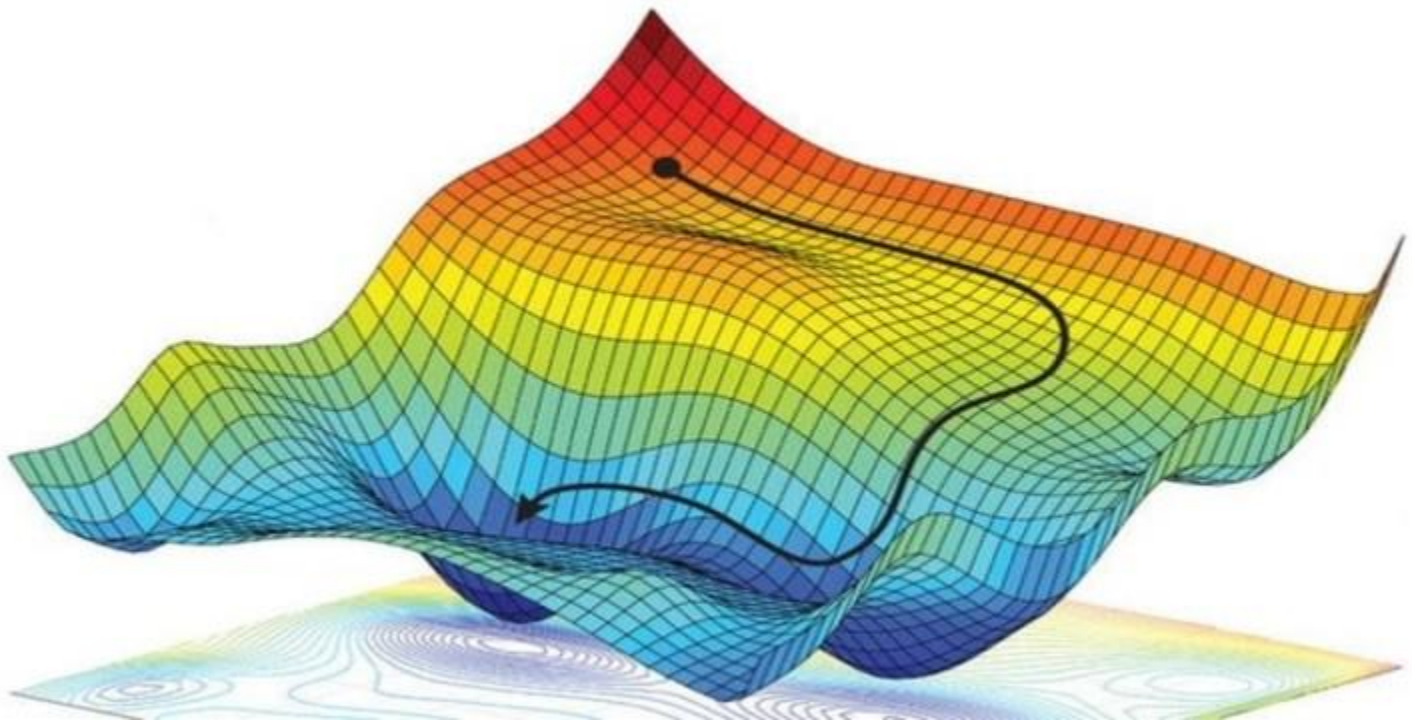
$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}(Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \cdot \log (1 - \hat{Y}_i))$$

https://www.tensorflow.org/api_docs/python/tf/keras/losses

La función de pérdida



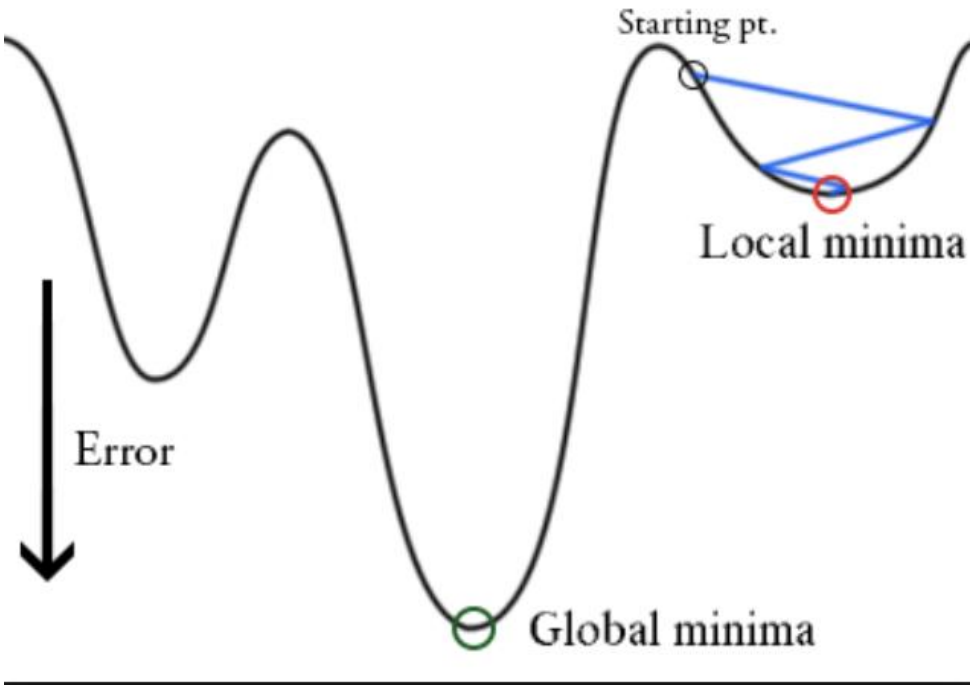
Optimizadores

Los optimizadores son la parte más importante del modelo. Hasta ahora hemos hablado del feedback para el modelo, algoritmo llamado back-propagation. Este es en sí el optimizador.
Volvamos al modelo de estudiantes, e inicialicemos los pesos de las neuronas al azar en un principio. Al salir la señal luego de pasar por todas las capas, la función de pérdida nos dirá que tan bien esta el modelo (inicialmente al azar). Ahora el modelo, debe reducir esta función de perdida. ¿Pero cómo lo puede hacer el modelo? Acá es donde aparece el optimizador.

El optimizador es una función matemática que usa derivadas, derivadas parciales, regla de la cadena y otros para comprender cuanto la red verá en pérdida haciendo pequeños cambios al peso de las neuronas.

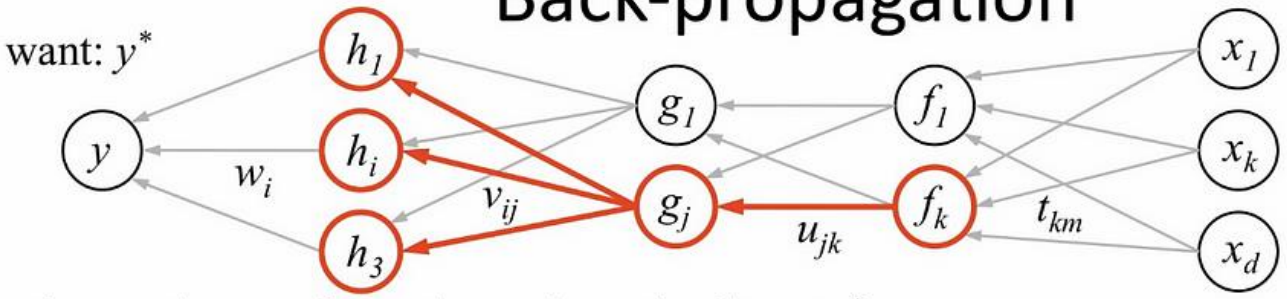
Optimizadores

La retropropagación es un método popular para entrenar redes neuronales artificiales, especialmente redes neuronales profundas.
Se necesita retropropagación para calcular el gradiente, que necesitamos para adaptar los pesos de las matrices de peso. Los pesos de las neuronas (es decir, los nodos) de la red neuronal se ajustan calculando el gradiente de la función de pérdida. Para ello se utiliza un algoritmo de optimización de descenso de gradiente. También se le llama propagación hacia atrás de errores.



Optimizadores

Back-propagation



1. receive new observation $\boldsymbol{x} = [x_1 \ldots x_d]$ and target y^*
2. **feed forward:** for each unit g_j in each layer 1…L
   compute g_j based on units f_k from previous layer: $g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right)$
3. get prediction y and error $(y\text{-}y^*)$
4. **back-propagate error:** for each unit g_j in each layer L…1

(a) compute error on g_j

$$\frac{\partial E}{\partial g_j} = \sum_i \sigma'(h_i) v_{ij} \frac{\partial E}{\partial h_i}$$

should g_j be higher or lower? | how h_i will change as g_j changes | was h_i too high or too low?

(b) for each u_{jk} that affects g_j

(i) compute error on u_{jk}

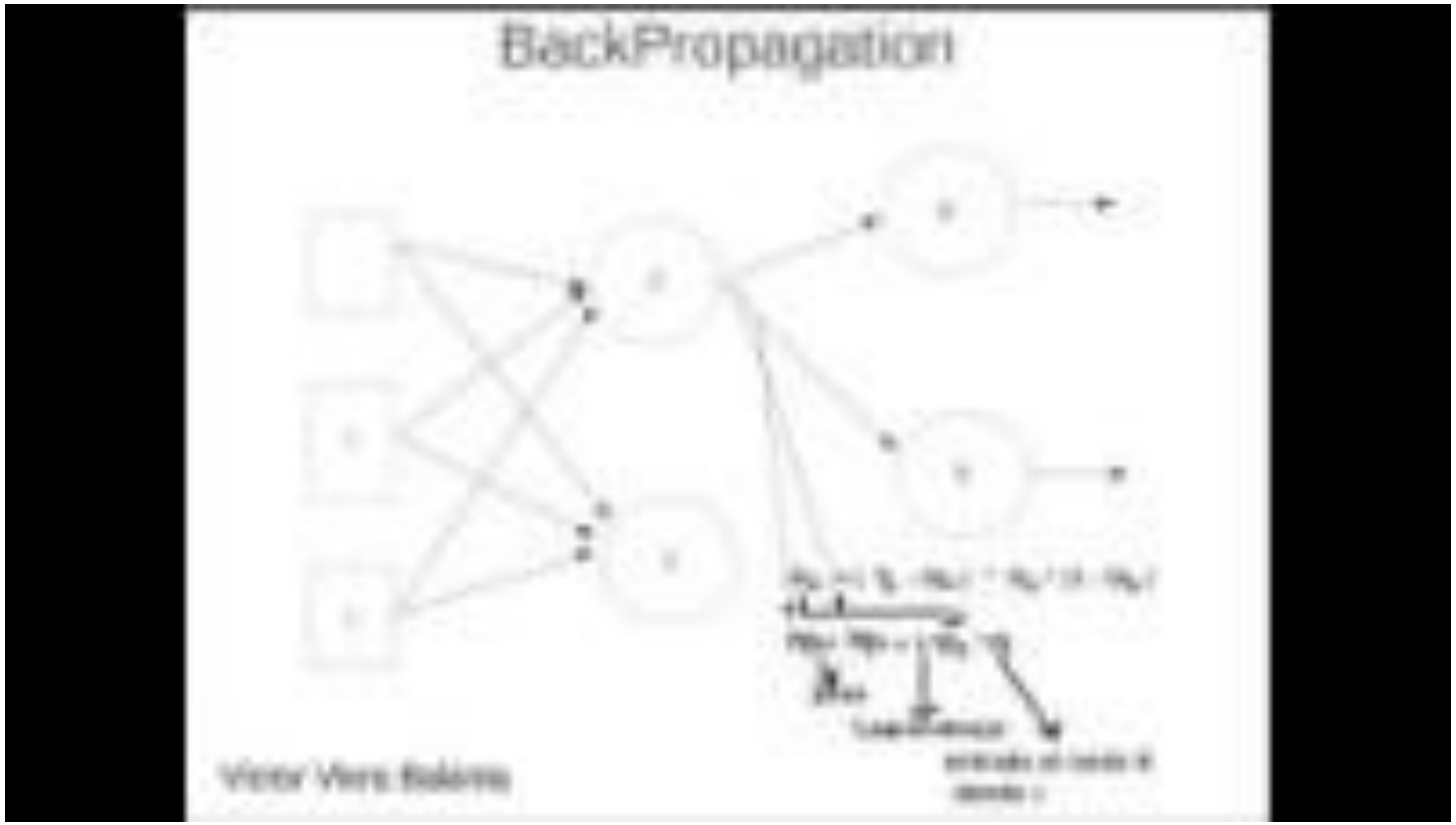
$$\frac{\partial E}{\partial u_{jk}} = \frac{\partial E}{\partial g_j} \sigma'(g_j) f_k$$

do we want g_j to be higher/lower | how g_j will change if u_{jk} is higher/lower

(ii) update the weight

$$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$$

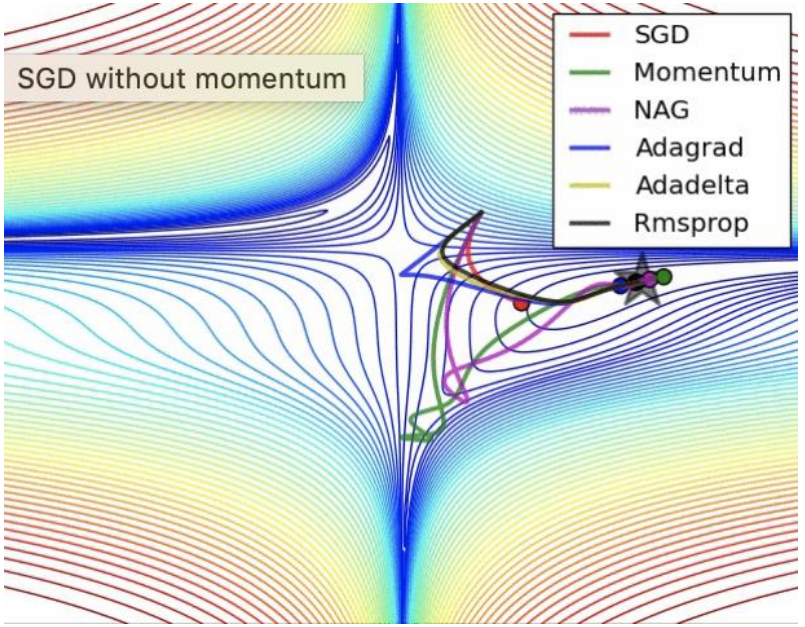
Optimizadores



Optimizadores

Entre los más comunes se encuentran

- Stochastic Gradient Descent (SGD)
- Adaptative Moment Estimation (Adam): Uno de los mejores actualmente.
- Adagrad
- Adadelta
- RMSPRop/Adamax/Nadam



https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Optimizadores

El cálculo de un entrenamiento de muestra base de datos iniciamos desde el input al output este proceso es llamado un paso. Usualmente el entrenamiento se hace en batches, debido a las condiciones de memoria de un computador.

Un batch es un subset de la muestra de entrenamiento completa.

La red actualiza sus pesos luego de procesar un batch completo, esto es llamado una iteración.

Una vez que se han analizado todos los elementos de la muestra de entrenamiento, con su actualización de pesos (batch by batch), tenemos lo que se conocerá como epoch.

Así luego de muchos epochs la red aprenderá a hacer predicciones cada vez más correctas, en base a las muestras de entrenamiento dado.

Métricas

La métrica puede entenderse como la función usada para juzgar la performance del modelo sobre un set de datos no visto, también conocido como set de datos de validación.

https://www.tensorflow.org/api_docs/python/tf/keras/metrics

Ejemplo

En base a un set de datos con información sobre la diabetes en indio/americanos. Los datos se encuentran en este link. Las variables de estos datos son:

- Input Variables (X):
  - Numero de ocasiones que se ha embarazado.
  - Concentración de glucosa a las dos hrs en test de tolerancia
  - Presión diastólica (mm Hg)
  - Doblamiento (ancho) de piel de tríceps (mm)
  - Dos horas serum insulina (mu U/ml)
  - Índice de masa corporal
  - Diabetes pedigree function
  - Edad
- Output Variables
  - (y): Clase (0 o 1)

Ejemplo

¿Y la predicción?
Es fácil predecir con nuestro modelo, simplemente utilizando predict en nuestro código. En nuestro caso actual la salida es una función sigmoide entre 0 y 1, por lo que tenemos que normalizar para que nuestra predicción sea el valor 0 o 1.

```
# make probability predictions with the model
predictions = model.predict(X) # round
predictions rounded = [round(x[0]) for x in
predictions]
```

O bien, podemos hacerlo directamente con:

```
# make probability predictions with the model
predictions = model.predict_classes(X)
```