Escuela AI HUB de verano
Sobre IA y Educación

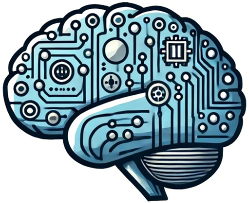
Deep Learning: Fundamentos y Aplicaciones

Steven Van Vaerenbergh (Universidad de Cantabria)

3 de julio de 2024

CSIC Consejo Superior de Investigaciones Científicas | CSIC HUB Inteligencia Artificial
Fundación "la Caixa"
IFIC Institut de Física Corpuscular | Vniversitat de València

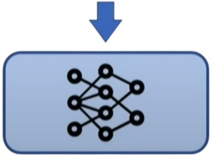
Contents

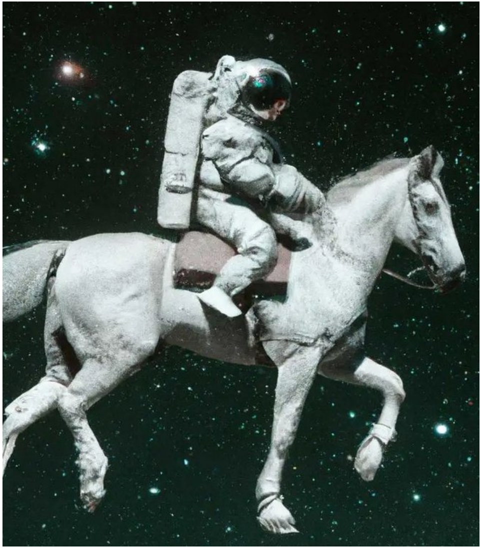


Introduction to Deep Learning

Deep learning for generating content

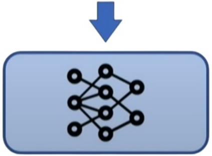
"A photo of an astronaut riding a horse."

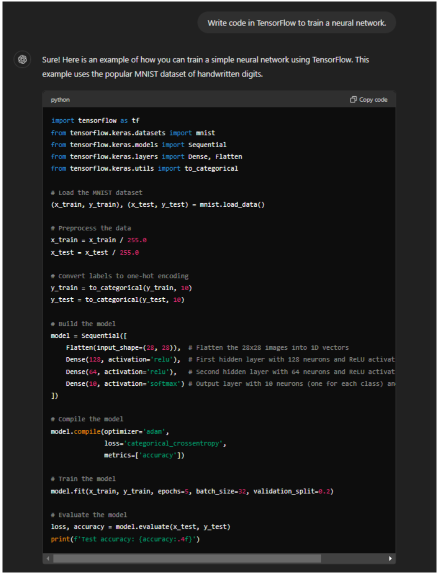




Deep learning for generating content

"Write code in TensorFlow to train a neural network."





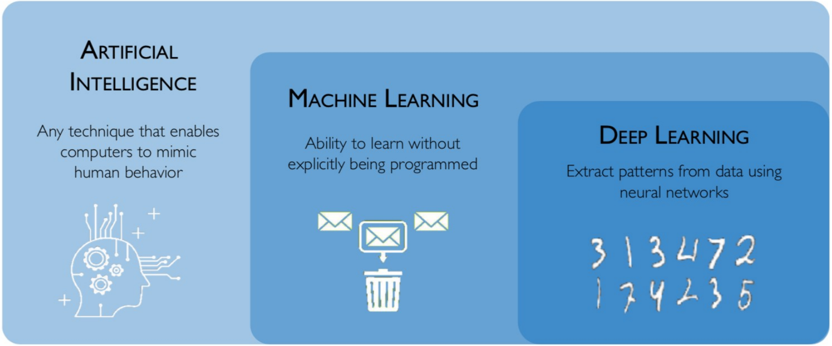
Write code in TensorFlow to train a neural network.

Sure! Here is an example of how you can train a simple neural network using TensorFlow. This example uses the popular MNIST dataset of handwritten digits.

python Copy code

```python
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess the data
x_train = x_train / 255.0
x_test = x_test / 255.0

# Convert labels to one-hot encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build the model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten the 28x28 images into 1D vectors
    Dense(128, activation='relu'),  # First hidden layer with 128 neurons and ReLU activat
    Dense(64, activation='relu'),   # Second hidden layer with 64 neurons and ReLU activat
    Dense(10, activation='softmax') # Output layer with 10 neurons (one for each class) an
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)

# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {accuracy:.4f}')
```

What is Deep Learning?



Teaching computers how to **learn a task** directly from **raw data**.

Image from [1].

Why Deep Learning?

Hand-engineered features are time consuming, brittle, and not scalable in practice.
Can we learn the underlying features directly from the data?

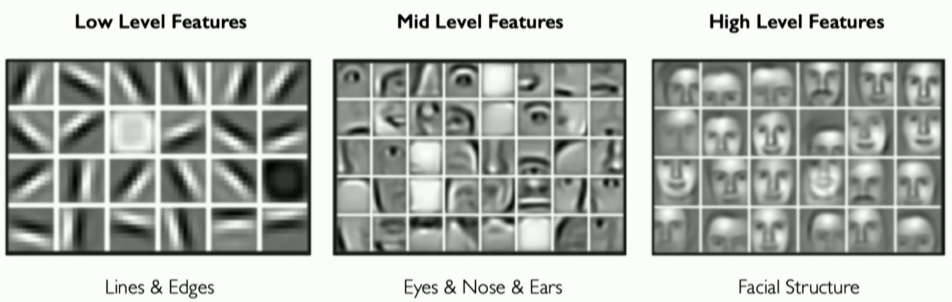


Image from [1].

Why now?

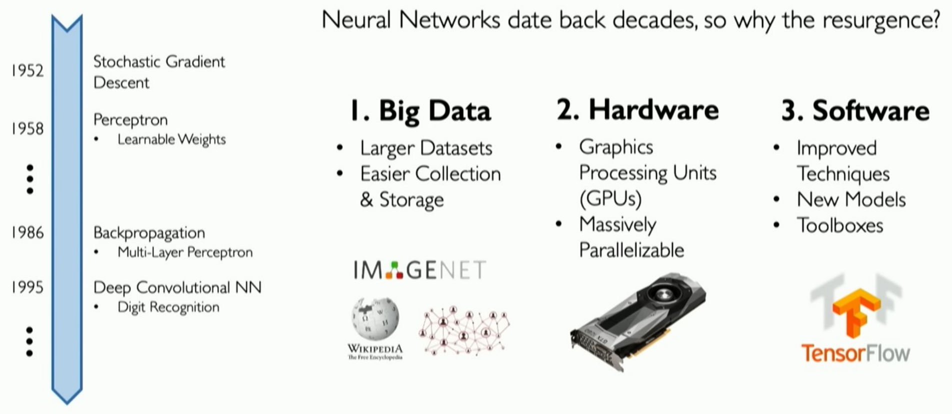
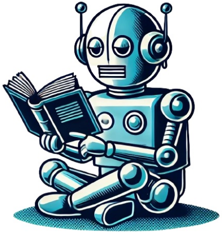
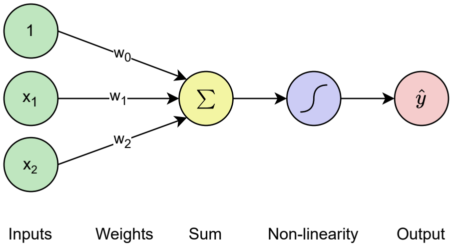


Diagram from [1].



Fundamentals of Deep Learning

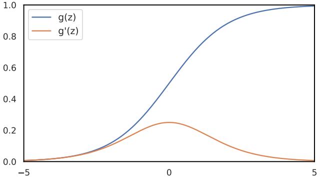
A single neuron: the Perceptron



- $\hat{y} = g(w_0 + x_1 w_1 + x_2 w_2)$
- w_0: bias term
- $g(\cdot)$: nonlinear activation function, e.g. sigmoid $g(z) = \frac{1}{1+e^{-z}}$

Nonlinear activation functions

Sigmoid function

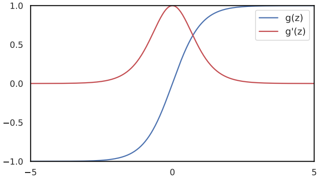


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

```
tf.math.sigmoid(z)
```

Hyperbolic Tangent

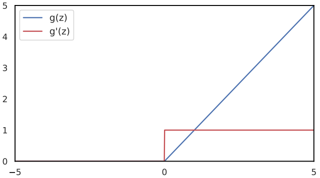


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

```
tf.math.tanh(z)
```

Rectified Linear Unit (ReLU)

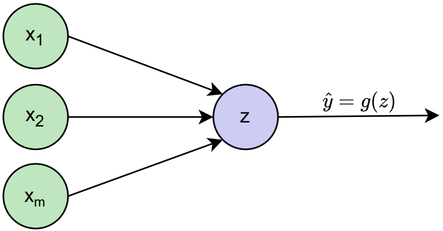


$$g(z) = max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0. \\ 0, & \text{otherwise.} \end{cases}$$

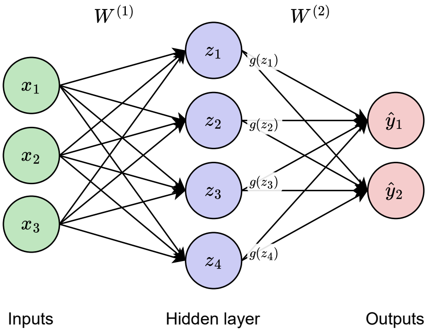
```
tf.math.relu(z)
```

Simplified neuron representation



$$z = w_0 + \sum_{j=1}^{m} x_j w_j \qquad \hat{y} = g(z)$$

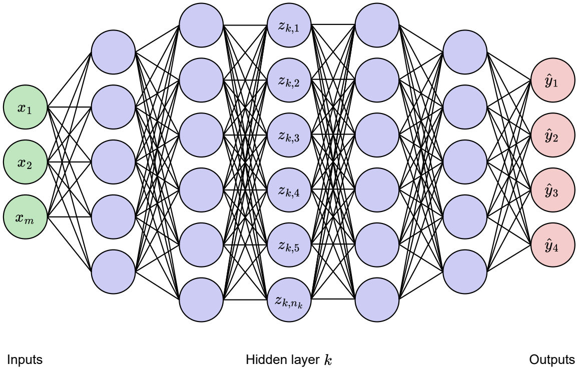
Hidden layer with multiple neurons, multiple outputs




```python
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation='relu',
     input_shape=(3,)),
    tf.keras.layers.Dense(2, activation='softmax')
])
```

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)}\right)$$

Deep Neural Network (DNN)



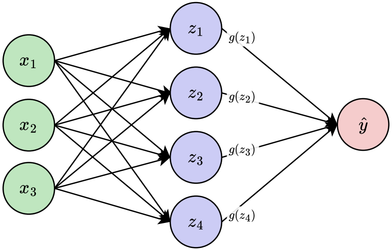

```python
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(5,
     input_shape=(3,)),
    tf.keras.layers.Dense(6),
    tf.keras.layers.Dense(6),
    tf.keras.layers.Dense(6),
    tf.keras.layers.Dense(5),
    tf.keras.layers.Dense(4)
])
```

$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

Loss functions

- The loss of our network measures the cost incurred from incorrect predictions.
- The empirical loss measures the total loss over our entire dataset.

$$X = \begin{bmatrix} 4.1 & 1.3 & 2.0 \\ 1.2 & 1.2 & 3.9 \\ 4.4 & -1.1 & 3.1 \\ & \vdots & \end{bmatrix}$$



$$\hat{Y} = \begin{bmatrix} 0.3 \\ 0.1 \\ 0.9 \\ \vdots \end{bmatrix} ; Y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

Empirical loss: $J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

Loss functions

Binary cross entropy loss:

- For classification models that output 0 or 1.
- $J(W) = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)}\log\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{y}^{(i)}\right)\right] \quad$ con $\hat{y} = f(x^{(i)}; W)$

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted))
```

Loss functions

Binary cross entropy loss:

- For classification models that output 0 or 1.
- $J(W) = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)}\log\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\log\left(1-\hat{y}^{(i)}\right)\right] \quad \text{con } \hat{y} = f(x^{(i)}; W)$

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted))
```

Mean Squared Error loss:

- For regression models that output continuous variables.
- $J(W) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 \quad \text{con } \hat{y} = f(x^{(i)}; W)$

```
loss = tf.reduce_mean(tf.square(tf.subtract(y, predicted)))
loss = tf.keras.losses.MSE(y, predicted)
```

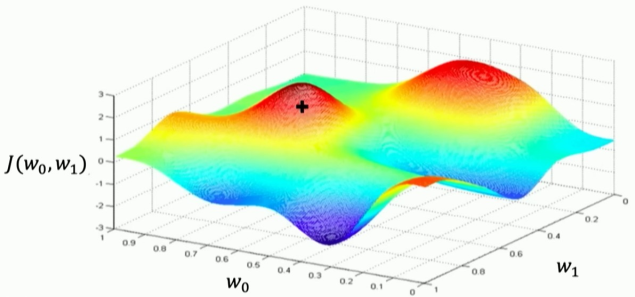
Loss minimization

Find the network weights that achieve the lowest loss.

$$W^* = \arg\min_{W} J\left(\{W^{(0)}, W^{(1)}, \ldots\}\right)$$

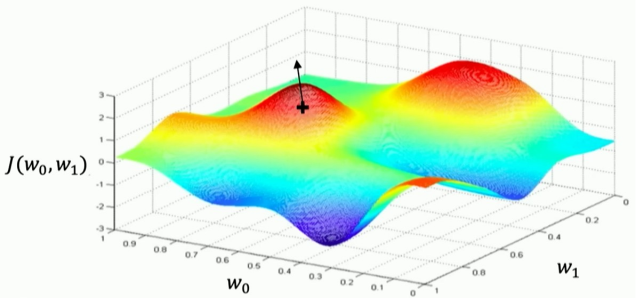
Loss minimization

Randomly pick an initial value for (w_0, w_1).



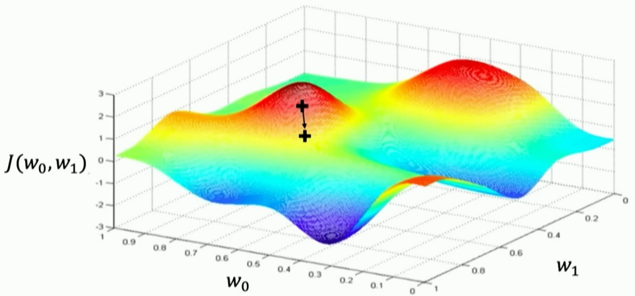
Loss minimization

Compute the gradient $\partial J(W)/\partial W$.



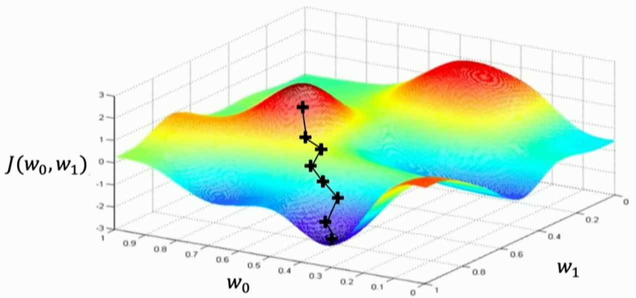
Loss minimization

Take a small step in the opposite direction of the gradient.



Loss minimization

Repeat until convergence.



Gradient descent algorithm

Algorithm:

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient $\frac{\partial J(W)}{\partial W}$
4. Update weights: $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

- How do we find the gradient $\frac{\partial J(W)}{\partial W}$?
- How do we find the learning rate η?

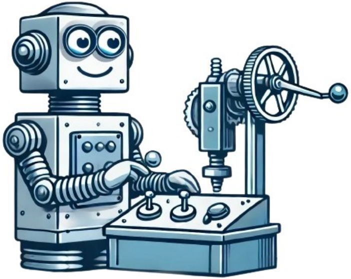
Computing the gradients: Backpropagation

How does a small change in one weight (e.g. w_1) affect the final loss $J(W)$?

$$x_1 \xrightarrow{w_1} z_1 \xrightarrow{w_2} \hat{y} \longrightarrow J(W)$$

Apply the chain rule: $\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$

Traditionally: calculated by hand. Nowadays: **Autograd**.

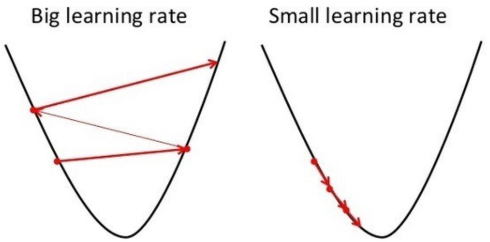


Deep Learning in Practice

How to choose the learning rate?

A correct choice of the learning rate is critical.

- Large learning rates overshoot, become unstable and diverge.
- Small learning rates converge slowly.



What about an **adaptive learning rate**?

Different gradient descent algorithms

W: parameters, η: learning rate, g_t: gradient $\frac{\partial J(W)}{\partial W}$
v and m: used for moving averages of gradients.

SGD (Stochastic Gradient Descent)

Classic optimization method.

$$W_{t+1} = W_t - \eta \cdot g_t$$

RMSprop

Adapts the learning rate for each parameter:

$$v_t = \beta v_{t-1} + (1-\beta) g_t^2$$

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot g_t$$

Momentum

Incorporates velocity to accelerate SGD:

$$v_t = \gamma v_{t-1} + \eta g_t$$

$$W_{t+1} = W_t - v_t$$

Adam (Adaptive Moment Estimation)

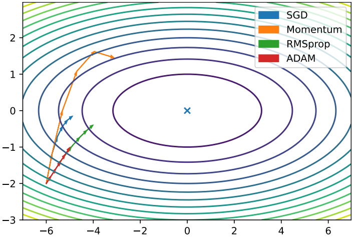
Combines Momentum and RMSprop:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

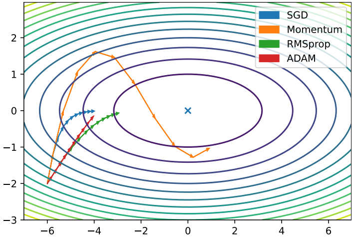
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot m_t$$

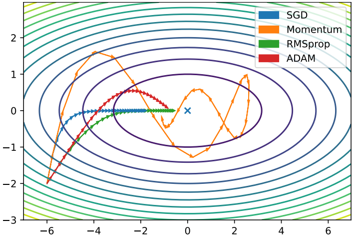
Different gradient descent algorithms



5 steps



10 steps



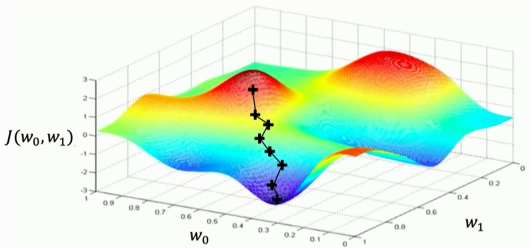
30 steps

Source: `https://gist.github.com/MaverickMeerkat/792628c18f42140d76a33cc2ccf153af`

Practical gradient descent: mini-batches

Algorithm:

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient $\frac{\partial J(W)}{\partial W}$
4. Update weights: $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights



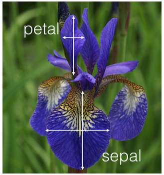
- In practice, calculating the gradient can be computationally intensive.
- Mini-batches: compromise between full calculation (expensive) and calculation for a single point (noisy).
- Pick a batch of B data points. $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$

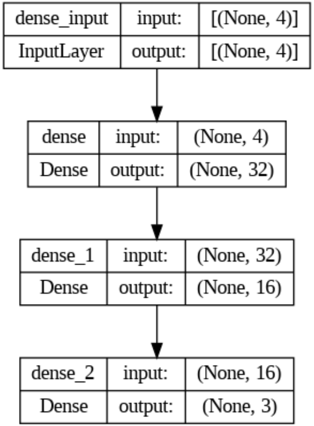
Example 1: Multilayer Perceptron (MLP)

MLP for classification:
`example01_mlp.ipynb`

Iris data set:

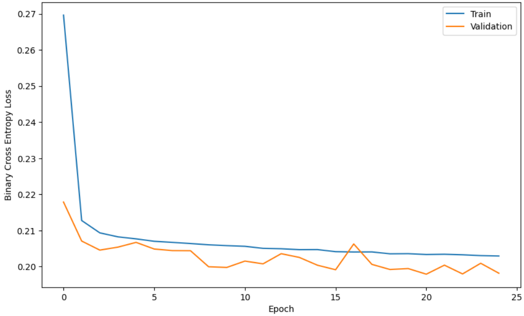
- 3 classes
- 150 data
- 4 features



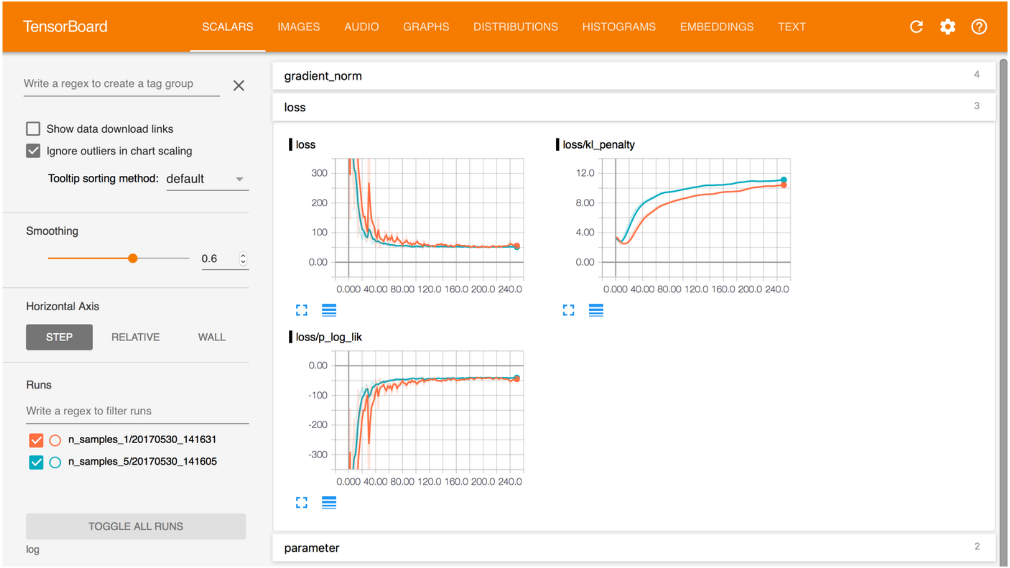


Learning curves

- Visual representations of model training progress over time.
- Display training and validation loss or accuracy metrics.
- Assist in hyperparameter tuning and determining optimal training duration.

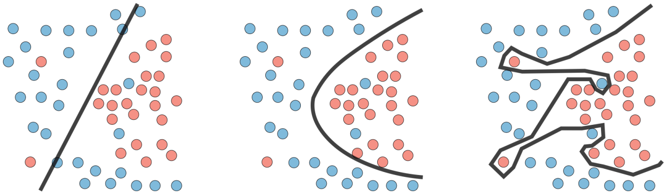


TensorBoard enables real-time monitoring and analysis of learning curves.


TensorBoard
SCALARS
IMAGES
AUDIO
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
EMBEDDINGS
TEXT
Write a regex to create a tag group
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
n_samples_1/20170530_141631
n_samples_5/20170530_141605
TOGGLE ALL RUNS
log
gradient_norm
4
loss
3
loss
loss/kl_penalty
loss/p_log_lik
parameter
2

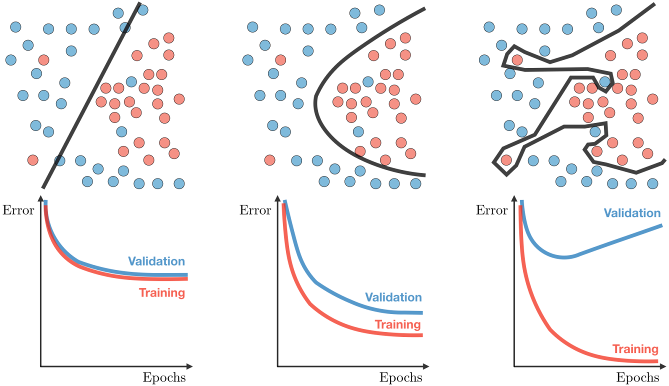
The problem of overfitting

3 classifiers have been trained over the dataset shown in the figure



Which one will perform better over a different **test** dataset? Tradeoff between:

- Training error / test error (generalization error, a.k.a. out-of-sample error)
- Bias/variance of the model/classifier



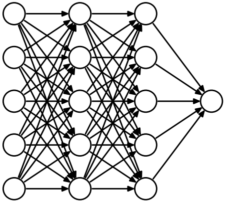
- **Underfitting** refers to a model that cannot learn the training dataset.
- **Overfitting** refers to a model that has learned the training dataset too well, including the statistical noise or random fluctuations in the training dataset.

The complexity of the learned model can be restricted by **regularizing** the optimization problem.

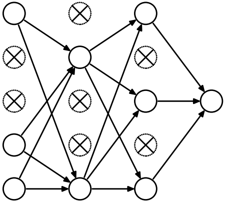
Dropout

- Randomly drop units (along with their connections) from the neural network during training[a].
- Reduces overfitting, improves generalization.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(10, activation='softmax')
])
```



a) Standard neural network

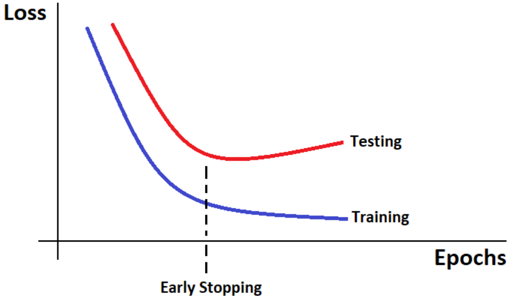


b) After applying dropout

[a] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

Early stopping

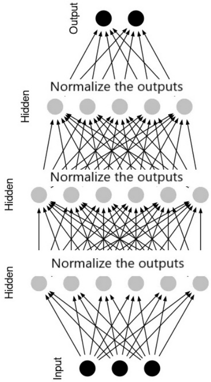
Use the model obtained before overfitting happened.



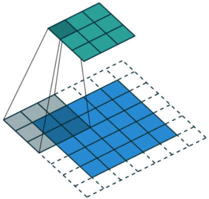
Batch Normalization

- Normalizes layer inputs to have zero mean and unit variance across mini-batches[a].
- Stabilizes the learning process, allows for higher learning rates, and reduces the sensitivity to initial weights.
- Enabling faster and more reliable training of deep networks.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

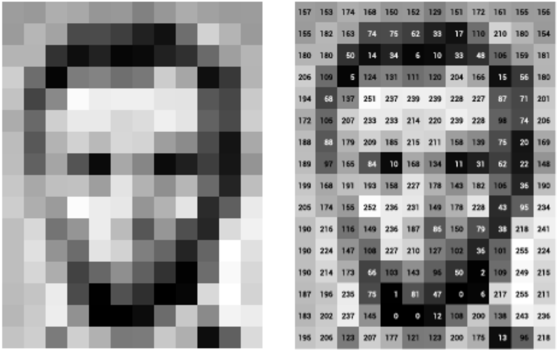


[a] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.



Convolutional Neural Networks

How a computer sees an image

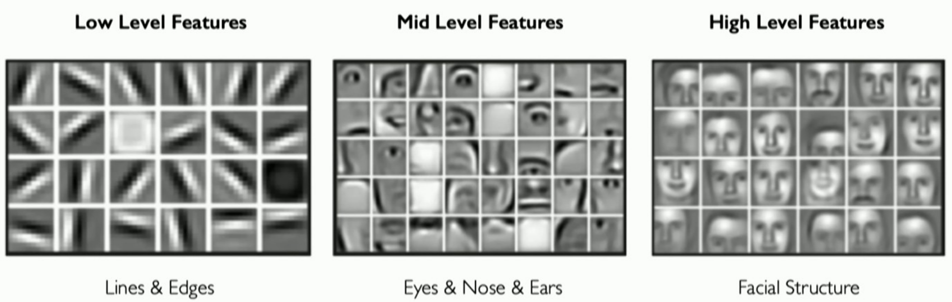


157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Image from [3].

Learning feature representation

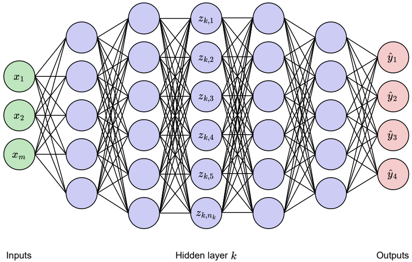
Can we learn a **hierarchy of features** directly from the data instead of hand-engineering them?



Fully-connected neural network

Input:

- 2D image
- Flatten into a 1D array.

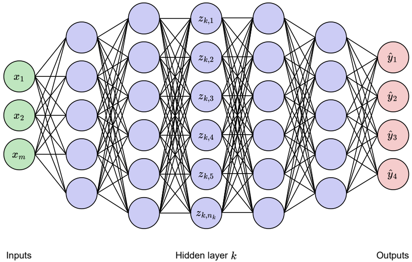


- Connect neuron in hidden layer to all neurons in previous layer.
- No spatial information!
- Many parameters!

Fully-connected neural network

Input:

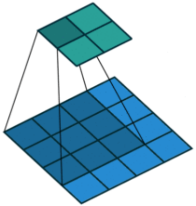
- 2D image
- Flatten into a 1D array.



- Connect neuron in hidden layer to all neurons in previous layer.
- No spatial information!
- Many parameters!

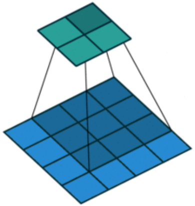
How can we maintain some of that spatial structure?

Filter extraction with convolution



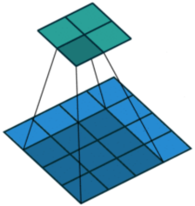
- Filter of size 3×3: 9 different weights.
- Apply this same filter to 3×3 patches in input.
- Shift by 1 pixel (stride=1).

Filter extraction with convolution



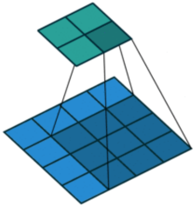
- Filter of size 3×3: 9 different weights.
- Apply this same filter to 3×3 patches in input.
- Shift by 1 pixel (stride=1).

Filter extraction with convolution



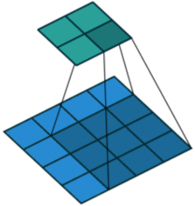
- Filter of size 3×3: 9 different weights.
- Apply this same filter to 3×3 patches in input.
- Shift by 1 pixel (stride=1).

Filter extraction with convolution



- Filter of size 3×3: 9 different weights.
- Apply this same filter to 3×3 patches in input.
- Shift by 1 pixel (stride=1).

Filter extraction with convolution



- Filter of size 3×3: 9 different weights.
- Apply this same filter to 3×3 patches in input.
- Shift by 1 pixel (stride=1).

Images from Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285* (2016)

Interactive demo of a convolutional neural network

https://adamharley.com/nn_vis/cnn/2d.html

Producing feature maps

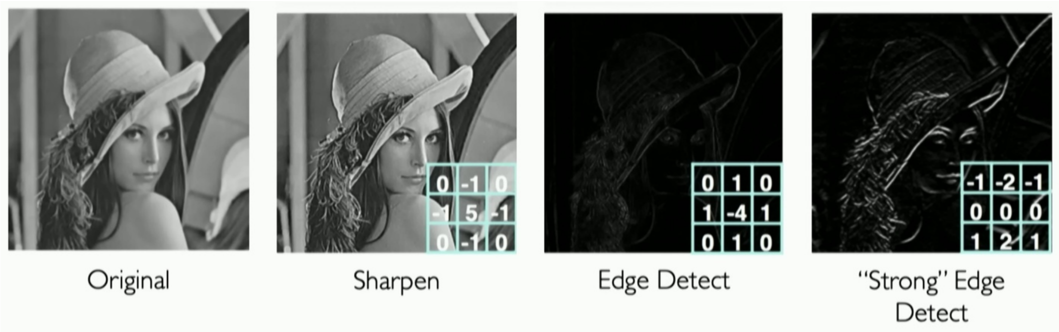


Image from [1].

Convolution and pooling

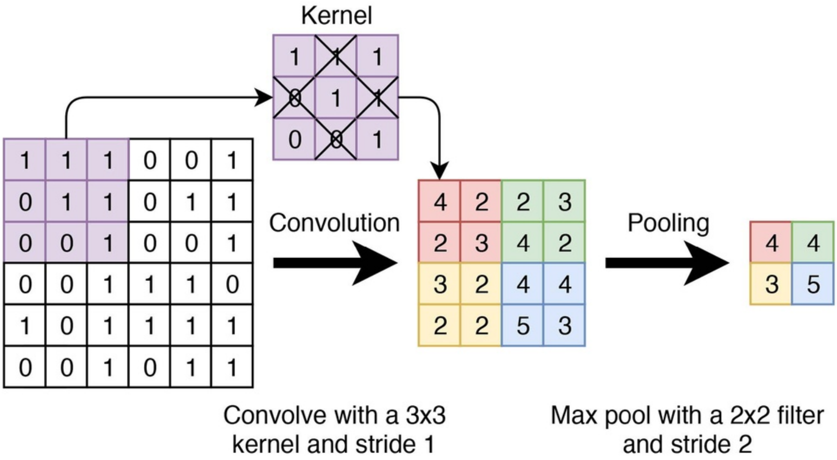
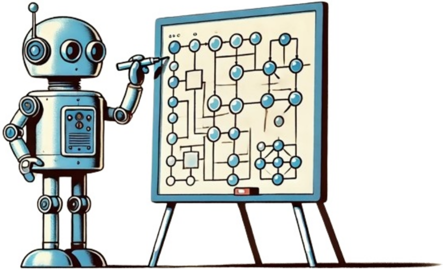


Figure from Verschoof and Lambers, 2019.

Example 2: Convnets for MNIST

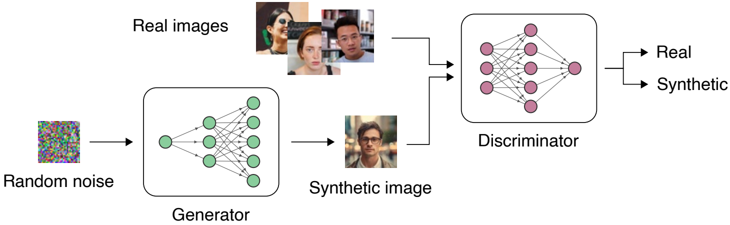
example02_convnet_orig.ipynb



Architectures

Generative Adversarial Networks (GANs)

- GANs consist of two neural networks, a Generator and a Discriminator, that are trained simultaneously through adversarial competition[1].
- The generator's role is to generate realistic data samples from random noise.
- The discriminator's role is to distinguish between real data samples and those generated by the generator.

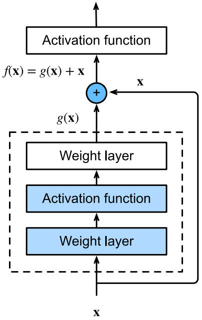


[1] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014).

Residual Networks

- Introduce **skip connections**, which bypass one or more layers[a].
- Addresses the degradation problem where adding more layers to a deep network can lead to higher training error.
- ResNets can train very deep networks (over 150 layers).

```
from official.vision.keras_layers import ResidualBlock
model = tf.keras.Sequential([
    tf.keras.Input(shape=(224, 224, 3)),
    ResidualBlock(filters=64, strides=1),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

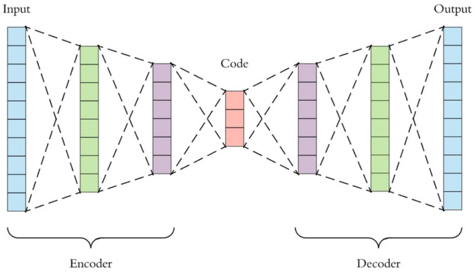


[a] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

Autoencoders

- Bottleneck architecture: An encoder that reduces the data dimensions and a decoder that reconstructs the data from the reduced dimensions[2].
- Unsupervised method that learns efficient encodings of the input data.
- Equivalent to PCA if encoder and decoder are linear models.

```
> example03_autoencoder.ipynb
```



[2] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *Science* 313.5786 (2006), pp. 504–507.

Recurrent Neural Networks (RNNs)

- Designed to process sequences. They remember information from previous inputs.
- Training uses "Backpropagation Through Time", which unfolds the network through time and then propagating errors back through these unfolded steps.
- Best-known RNN: Long short-term memory (LSTM)[3].

```
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(50,
     return_sequences=True,
     input_shape=(100, 1)),
    tf.keras.layers.LSTM(50),
    tf.keras.layers.Dense(1)
])
```

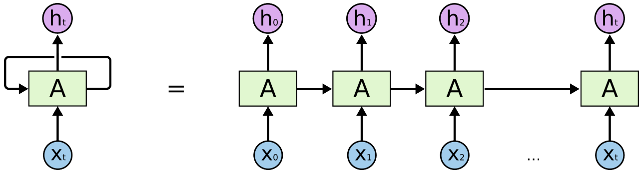
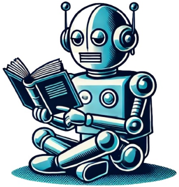


Image from https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[3] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

Key takeaways

1. Neurons: building blocks with nonlinear activation functions
2. Neural networks: optimization through back-propagation
3. DNN in practice: adaptive learning, mini-batches, regularization
4. CNN for computer vision: classification, object detection, segmentation, . . .
5. A diverse range of deep learning architectures can be assembled using these fundamental building blocks.



Web references

[1] MIT 6.S191 Introduction to Deep Learning
http://introtodeeplearning.com/

[2] Shervine Amidi's Machine Learning tips and tricks cheatsheet
https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks

[3] Openframeworks https://openframeworks.cc/ofBook/chapters/image_processing_computer_vision.html

References I

- Dumoulin, Vincent and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285* (2016).
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.
- Goodfellow, Ian et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014).
- He, Kaiming et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- Hinton, Geoffrey E and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *Science* 313.5786 (2006), pp. 504–507.
- Hochreiter, Sepp and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.
- Ioffe, Sergey and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.
- James, Gareth et al. *An introduction to statistical learning: With applications in python*. Springer Nature, 2023.

References II

- Kingma, Diederik P and Jimmy Ba. "Adam: A method for stochastic optimization". In: *3rd International Conference for Learning Representations*. San Diego, 2015.
- Ruder, Sebastian. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).
- Srivastava, Nitish et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.
- Verschoof-Van der Vaart, Wouter Baernd and Karsten Lambers. "Learning to look at LiDAR: The use of R-CNN in the automated detection of archaeological objects in LiDAR data from the Netherlands". In: *Journal of Computer Applications in Archaeology* 2.1 (2019), pp. 31–40.