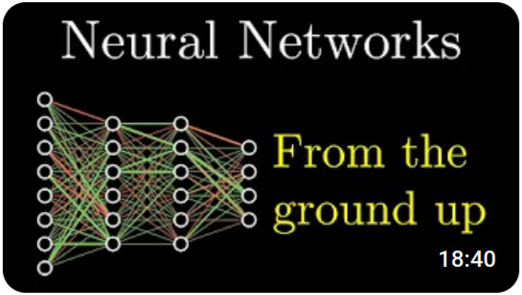
Artificial intelligence

Artificial Neural Networks (ANN)

Carl McBride Ellis

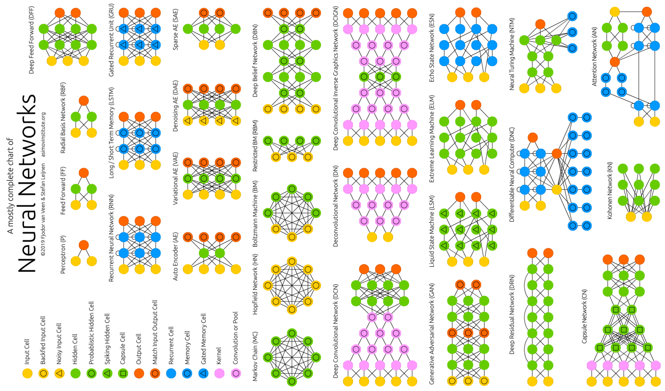
✉ carl.mcbride@u-tad.com

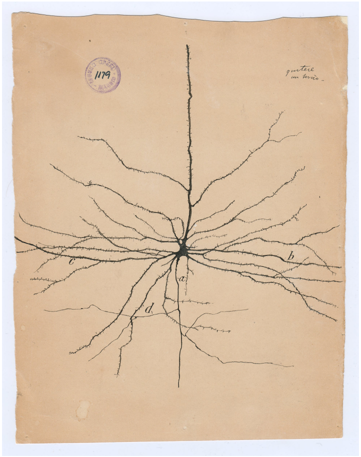
A video by Grant Sanderson of 3Blue1Brown:



"But what is a neural network?"

The Neural Network Zoo (computational graphs)

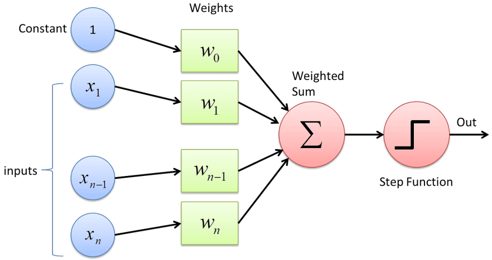


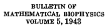


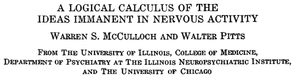
Santiago Ramón y Cajal (Premio Nobel de Medicina 1906)

(*¿Aberrante o necesario? La historia del feo sello azul que salvó los dibujos de Ramón y Cajal*)

The McCulloch-Pitts (MP) artificial neuron (Threshold Logic Unit (TLU)) takes binary inputs




BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

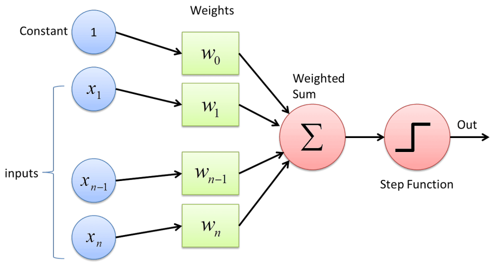

A LOGICAL CALCULUS OF THE
IDEAS IMMANENT IN NERVOUS ACTIVITY

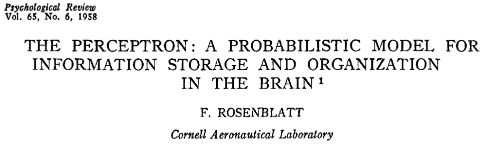
WARREN S. MCCULLOCH AND WALTER PITTS

FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

(Paper: "A logical calculus of the ideas immanent in nervous activity" Warren S. McCulloch and Walter Pitts (1943)

The perceptron: now takes real valued inputs.
Graph:




Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

(Paper: "The perceptron: A probabilistic model for information storage and organization in the brain" Frank Rosenblatt (1958)

For both the MP and the perceptron the 'activation' function is the Heaviside function (H):

$$\begin{aligned}\hat{y} &= H(w_1x_1 + w_2x_2 + ... + w_nx_n + b) \\ &= H(\mathbf{w} \cdot \mathbf{x} + b)\end{aligned}$$

where

$$H(x) := \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

in python + numpy

```python
def perceptron(x, w, b):
    """ Perceptron function with weight w and bias b """
    v = np.dot(w, x) + b
    y = np.heaviside(v, 0.5)
    return y
```

Logic gates ‘truth’ tables:

1-input logic gate

Input	Output
A	NOT
0	1
1	0

2-input logic gates

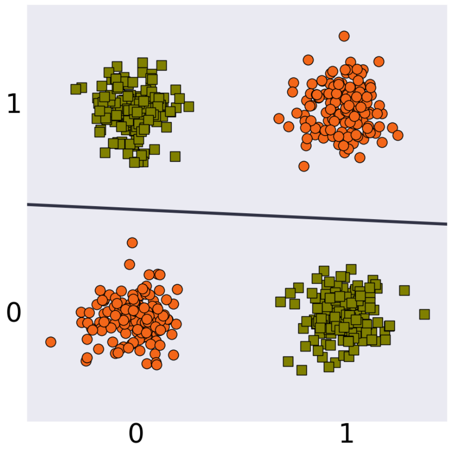
A	B	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```python
# NOT using a perceptron
def perceptron_NOT(x):
    return perceptron(x, w = -1, b = 0.5)
```

```python
# AND using a perceptron
def perceptron_AND(x):
    w = np.array([1, 1])
    b = -1.5
    return perceptron(x, w, b)
```

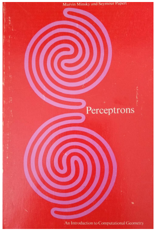
```python
# OR using a perceptron
def perceptron_OR(x):
    w = np.array([1, 1])
    b = -0.5
    return perceptron(x, w, b)
```

The ‘XOR problem’



The first ‘AI winter’

However, the book “Perceptrons: an introduction to computational geometry” by Marvin Minsky and Seymour Papert (1969)


Marvin Minsky and Seymour Papert
Perceptrons
An Introduction to Computational Geometry

mathematically proved that the XOR is not linearly separable using a single perceptron. This ‘killed’ ANN for many years.

Neural networks: Universal approximation theorem

> "...any continuous function on a compact set in $\mathbb{R}^n$ can be approximated by a multi-layer feed-forward network with only one hidden layer and non-polynomial activation function (like the sigmoid function)."

(Note the word **continuous**. Categorical features are not continuous, hence NN do not work so well for tabular data)

Papers (all from 1989):

- *"Multilayer feedforward networks are universal approximators"*
- *"Approximation by superpositions of a sigmoidal function"*
- *"On the approximate realization of continuous mappings by neural networks"*

Keras (“Deep learning for humans”)

(3.0 Release: 28 November 2023)

Written by François Chollet to make TensorFlow *much* easier to use!

DL frameworks that can be used as the Keras ‘backend’

TensorFlow

PyTorch

JAX

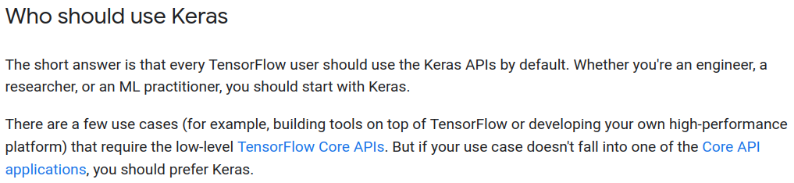
Keras comes as part of TensorFlow
To install TensorFlow 2 as well as Keras

```
pip install tensorflow

pip freeze | grep tensor
pip freeze | grep keras
```

To install PyTorch

```
pip install torch

pip freeze | grep torch
```

BTW: Even TensorFlow does not recommend using TensorFlow!


Who should use Keras

The short answer is that every TensorFlow user should use the Keras APIs by default. Whether you're an engineer, a researcher, or an ML practitioner, you should start with Keras.

There are a few use cases (for example, building tools on top of TensorFlow or developing your own high-performance platform) that require the low-level TensorFlow Core APIs. But if your use case doesn't fall into one of the Core API applications, you should prefer Keras.

(Source: `www.tensorflow.org/guide/keras`)

(See also: *"A Comparative Survey of PyTorch vs TensorFlow for Deep Learning: Usability, Performance, and Deployment Trade-offs"* (2025))

Linear regression uses the identity activation function (*i.e.* `activation=None`) and the $L2$ loss function (*i.e.* MSE):

```
model = Sequential()
model.add(Dense(n_neurons, activation=None))

optimizer = tensorflow.keras.optimizers.SGD(learning_rate=0.3)
model.compile(loss='mean_squared_error', optimizer=optimizer)
```

$$\hat{y} = 1(\mathbf{w} \cdot \mathbf{x} + b)$$

Practical session

Use Keras to perform linear regression

`notebook_ANN_Keras_linear.ipynb`

Logistic regression using the sigmoid activation function and the log-loss (binary_crossentropy) function:

```
model = Sequential()
model.add(Dense(n_neurons, activation='sigmoid'))

optimizer = tensorflow.keras.optimizers.SGD(learning_rate=0.3)
model.compile(loss='binary_crossentropy', optimizer=optimizer)
```

$$\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

Note: If one had more than two classes then use the categorical_crossentropy function, such as the case for the MNIST problem which has 10 classes.

Practical session

Use Keras to perform logistic regression

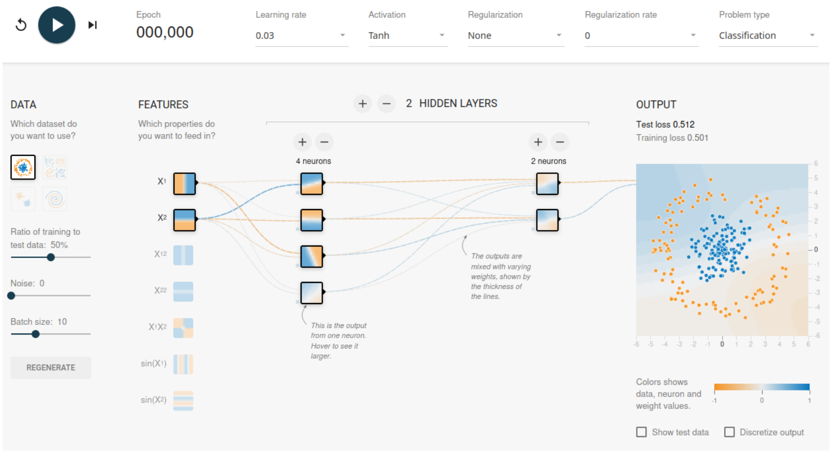
`notebook_ANN_Keras_logistic.ipynb`

Remark: most of the time we use neural networks to perform either image classification, or image segmentation.

We have now seen the overall Keras structure:

```
model = keras.Sequential([layers......])

model.compile(...)

model.fit(...)

model.evaluate(...)

model.predict(...)
```

Fun: We can play with an interactive NN classifier in our browser


Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons

2 neurons

This is the output from one neuron. Hover to see it larger.

The outputs are mixed with varying weights, shown by the thickness of the lines.

OUTPUT

Test loss 0.512

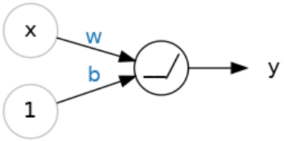
Training loss 0.501

Colors shows data, neuron and weight values.

☐ Show test data ☐ Discretize output

Activation functions

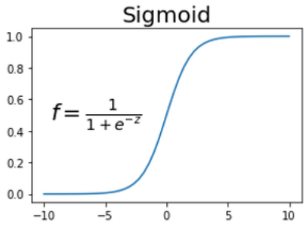
We have seen the linear (1) and the sigmoid (σ) activation functions. The ramp, or rectified linear unit activation (ReLU) function is cheaper to calculate than a sigmiod (having expensive exponentials)

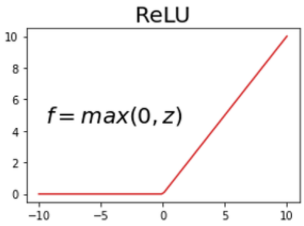


A rectified linear unit.

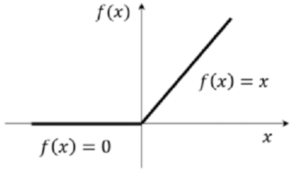
```python
def relu(x):
    return max(0, x)
```

Sigmoid vs ReLU

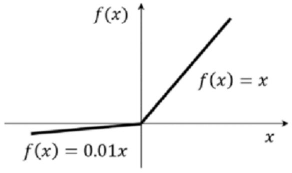




However, there is a problem with the logistic (`sigmoid`) and the hyperbolic tangent (`tanh`) activation functions: vanishing gradients. ReLU neurons can also 'die' due to having a zero gradient.
Solution: LeakyReLU



ReLU activation function

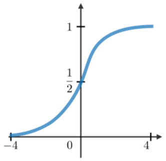
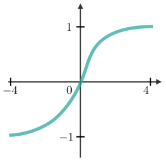
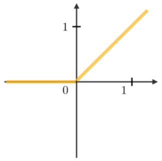
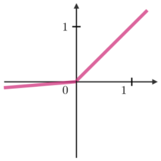


LeakyReLU activation function

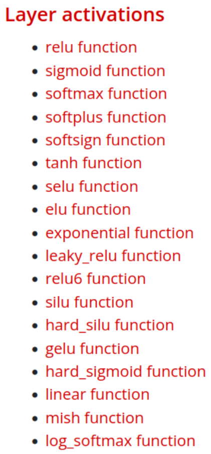
```python
def leakyrelu(x, slope):
    return (max(0, x) + slope * min(0, x))
```

usually with a very small slope of around 0.01

Summary:

Sigmoid	**Tanh**	**ReLU**	**Leaky ReLU**
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0,z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

Keras has many more layer activation functions:



Layer activations

- relu function
- sigmoid function
- softmax function
- softplus function
- softsign function
- tanh function
- selu function
- elu function
- exponential function
- leaky_relu function
- relu6 function
- silu function
- hard_silu function
- gelu function
- hard_sigmoid function
- linear function
- mish function
- log_softmax function

(See also: "Three Decades of Activations: A Comprehensive Survey of 400 Activation Functions for Neural Networks")

Training a NN

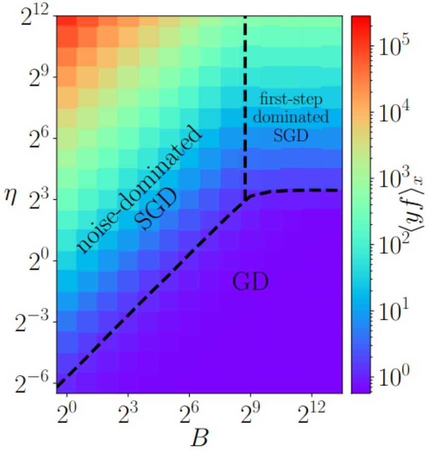
Epochs ($\approx$ iterations) and batches ($=$ subsamples)

- When all training samples are used to create one batch, the learning algorithm is called **batch gradient descent**.
- When the batch is the size of one sample, the learning algorithm is called **stochastic gradient descent**.
- When the batch size is more than one sample and less than the size of the training dataset, the learning algorithm is called **mini-batch gradient descent**.

In the case of mini-batch gradient descent, popular batch sizes include 32, 64, and 128 samples.

(Source: "Difference Between a Batch and an Epoch in a Neural Network")

Batch size vs. learning rate



(Paper: "*On the different regimes of stochastic gradient descent*")

We have seen a single neuron, but the magic happens when we have many connected neurons, forming a feed-forward neural network:

$$x_j^{(l)} = \sigma\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right)$$

where

$$w_{ij}^{(l)} \begin{cases} 1 \leq l \leq L, & \text{layers} \\ 0 \leq i \leq d^{(l-1)}, & \text{inputs} \\ 1 \leq j \leq d^{(l)}, & \text{outputs} \end{cases}$$

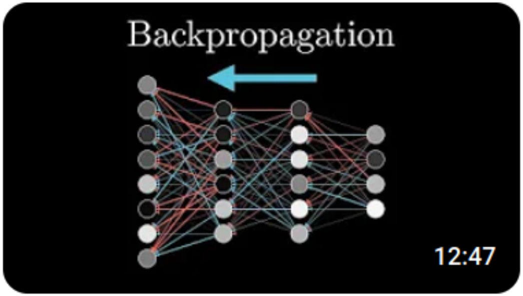
where σ is the activation function

But how does an ANN arrive at (aka learn) the best weights and biases to encode the training dataset?
We adjust the weights via

$$\frac{\partial \ \mathrm{error}(w)}{\partial w_{ij}^{(l)}} \ \ \forall \ \ i, j, l$$

and this can be done efficiently using **backpropagation**.
You will work through backpropagation next year in MLII

but in the meantime here is a video

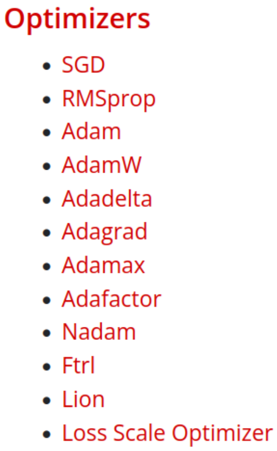


“What is backpropagation really doing?”

Foundational works on the backpropagation technique:

- Henry J Kelley "*Gradient Theory of Optimal Flight Paths*" (1960)
- Seppo Linnainmaa "*The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*" MSc Thesis (1970)

Keras optimizers:



Optimizers

- SGD
- RMSprop
- Adam
- AdamW
- Adadelta
- Adagrad
- Adamax
- Adafactor
- Nadam
- Ftrl
- Lion
- Loss Scale Optimizer

Note that the Keras SGD optimizer is really a GD optimizer

- if `batch_size = 1` we perform SGD
- if `batch_size` is between 1 and `n_train` we perfrom MBGD
- if `batch_size = n_train` we perform BGD

Learning rate (η) and the exploding gradient problem

> "...as the gradient is backpropagated through the network, it may grow exponentially from layer to layer. This can, for example, make the application of vanilla SGD impossible. Either the step size is too large for updates to lower layers to be useful or it is too small for updates to higher layers to be useful."

(paper: "The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions" (2018))

Starting weights

If all of the initial weights in a layer are constant then all of the gradients are the same and the NN does not start learning.

Solution: by default Keras uses `glorot_uniform` to get going.

Starting biases

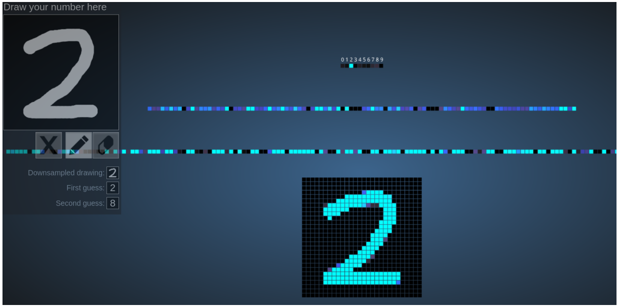
By default all of the initial biases are zero.

(For ReLU the He initialization is suggested)

MNIST

The MNIST ‘handwritten digits’ dataset (1994)

Interactive: 2D fully-connected network visualization



(This ANN model used 784 input neurons, 300 neurons in the first hidden layer, 100 neurons in the second hidden layer, and 10 neurons in the output layer.)

Keras datasets:

MNIST digits classification dataset

```
keras.datasets.mnist.load_data()
```

This is a dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

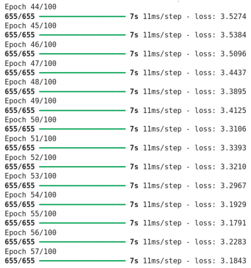
Practical session

We are going to do MNIST

`notebook_ANN_Keras_MNIST.ipynb`

Also: compare the results with using the `RMSprop` and `Adam` optimizers.

Note: we can turn off the Keras progress bar for a cleaner saved notebook



```
Epoch 44/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.5274
Epoch 45/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.5384
Epoch 46/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.5096
Epoch 47/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.4437
Epoch 48/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.3895
Epoch 49/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.4125
Epoch 50/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.3106
Epoch 51/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.3393
Epoch 52/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.3210
Epoch 53/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.2967
Epoch 54/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.1929
Epoch 55/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.1791
Epoch 56/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.2283
Epoch 57/100
655/655 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - loss: 3.1843
```

set

```
model.fit(..., verbose=0)
```

and

```
model.predict(..., verbose=0)
```

Notice the `flatten` layer

```
layers.Flatten()
```

which reshapes a 2 dimensional 28×28 image array
into a one dimensional vector of size 784.

note: this is similar to a `numpy` `reshape`

```
flat_array = array.reshape(-1)
```

also notice that this is now multi-class classification;
we now have 10 classes

```python
from keras.utils import to_categorical

# convert class vectors to binary class matrices
y_train_encoded = to_categorical(y_train)
y_test_encoded = to_categorical(y_test)
```

This will one-hot encode `y` (since this is **nominal** categorical data)
and use the `categorical_crossentropy` loss

(use `sparse_categorical_crossentropy` for ordinal categorical data)

When performing bianry classification we have two classes so the very last layer in our model will be:

```
model.add(Dense(2, activation='softmax')
```

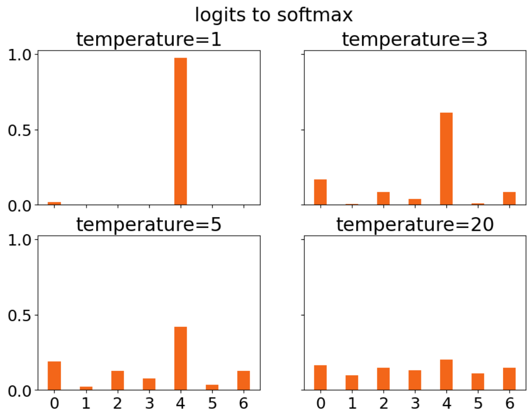
in conjunction with

```
loss = 'categorical_crossentropy'
```

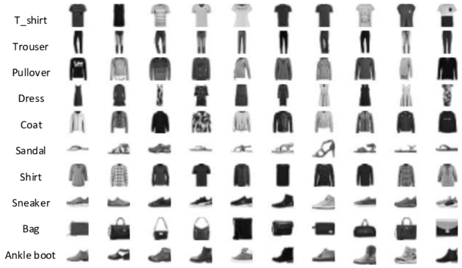
The `softmax` function will convert our logit scores into probabilities. We can then dichotomize this output via

```
label = np.argmax(prediction)
```

This is the multiclass extension of logistic regression.


logits to softmax
temperature=1
temperature=3
temperature=5
temperature=20
1.0
0.5
0.0
0
1
2
3
4
5
6

Let us also explore swapping out the MNIST dataset for the Fashion MNIST dataset:



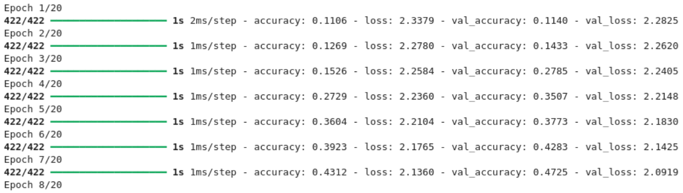
```
keras.datasets.fashion_mnist.load_data()
```

This is a dataset of 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images.

Loss function and metrics

```python
model.compile(loss="categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

By default whilst training Keras will report the loss (which is actually the cost) in terms of the log-loss, which is a proper scoring rule so we are OK. However, we may also like to know about other metrics, such as the (*enfant terrible*) accuracy score



```
Epoch 1/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.1106 - loss: 2.3379 - val_accuracy: 0.1140 - val_loss: 2.2825
Epoch 2/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.1269 - loss: 2.2780 - val_accuracy: 0.1433 - val_loss: 2.2620
Epoch 3/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.1526 - loss: 2.2584 - val_accuracy: 0.2785 - val_loss: 2.2405
Epoch 4/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.2729 - loss: 2.2360 - val_accuracy: 0.3507 - val_loss: 2.2148
Epoch 5/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.3604 - loss: 2.2104 - val_accuracy: 0.3773 - val_loss: 2.1830
Epoch 6/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.3923 - loss: 2.1765 - val_accuracy: 0.4283 - val_loss: 2.1425
Epoch 7/20
422/422 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.4312 - loss: 2.1360 - val_accuracy: 0.4725 - val_loss: 2.0919
Epoch 8/20
```

we can evaluate the preformance of our model on a test dataset using

```
model.evaluate(test_dataset,
               return_dict=True,
               verbose=1)
```

Layers

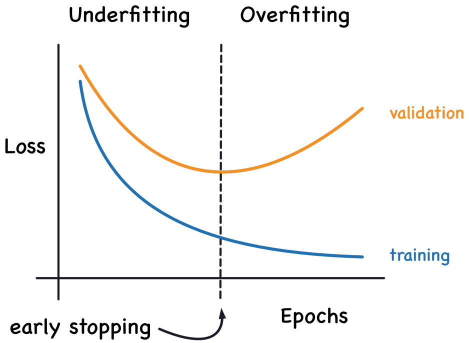
> “Most of deep learning consists of chaining together simple layers that will implement a form of progressive data distillation. A deep learning model is like a sieve for data processing, made of a succession of increasingly refined data filters—the layers.

from François Chollet “*Deep Learning with Python*” 2nd Edition (2021) (p. 28)

Keras makes creating your own neural network architecture easy by sequentially stacking layers in a modular fashion

- `Flatten` - a reshaping layer (*i.e.* from 28×28 to 1×784)
- `Dense` - a fully connected layer
- `BatchNormalization` - a normalization layer
- `Dropout` - a regularization layer to reduce over-fitting

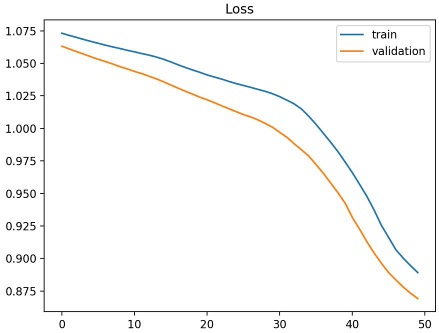
Training curves



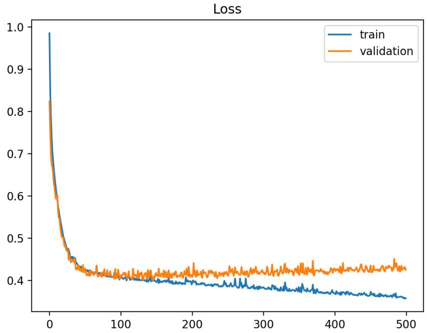
see: "How to use Learning Curves to Diagnose Machine Learning Model Performance"

- The training curve shows us how well the network is learning the data
- The validation curve indicates how well the model can generalize to new data

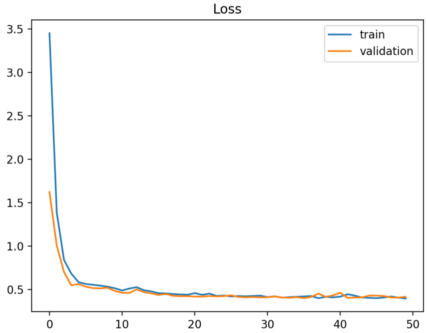
Remember, the whole point of machine learning is to create a useful model for **new** data, thus it is the validation curve interests us the most.



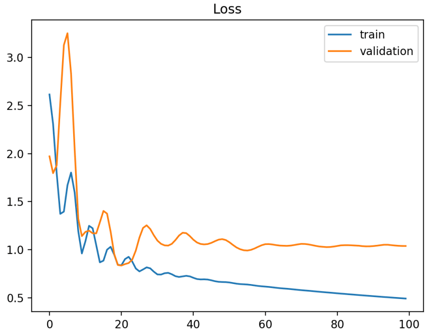
Under fitting: the training curve is decreases very slowly or even becomes almost flat: we are learing *very* slowly



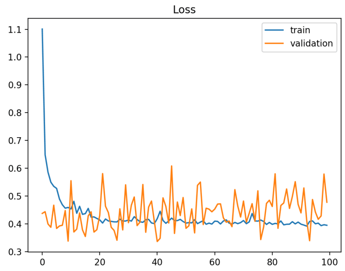
Over fitting: The model learns the training data quickly, and the validation gets slowly worse

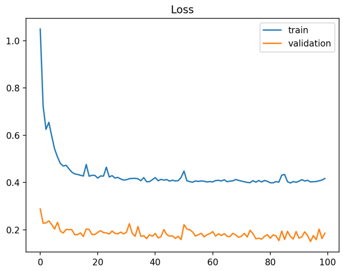


Good fitting: Both the training and validation are in harmony, and the validation keeps slowly decreasing



Poor training dataset: A large gap between performance, and the validation does not improve





Poor validation dataset: The validation is really noisy and does not decrease

Callbacks

EarlyStopping: Stop training when a monitored metric has stopped improving. By using a callback we can activate early stopping to avoid 'over-training' (learning the training data too well):

```
# This callback will stop the training when there is no improvement in
# the loss for 20 consecutive epochs.
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss', # default
    # minimum amount of change to count as an improvement
    min_delta=0.001,
    # how many epochs to wait before stopping
    patience=20,
    restore_best_weights=True,)

history = model.fit(callbacks=[early_stopping],)
```

Note that one should monitor the validation loss.

LearningRateScheduler This function keeps the initial learning rate for the first ten epochs and decreases it exponentially after that.

```python
def scheduler(epoch, lr):
    if epoch < 10:
        return lr
    else:
        return lr * ops.exp(-0.1)
```

```python
callback_LRS = keras.callbacks.LearningRateScheduler(scheduler)
history = model.fit(callbacks=[callback_LRS])
```

ReduceLROnPlateau: Reduce learning rate when a metric has stopped improving.

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                              factor=0.2,
                              patience=5,
                              min_lr=0.001)
```

Hyperparameters

Hyperparameters are parameters that are not learnt during trainning:

- architecture/topology tuning
    - number of hidden layers
    - neurons per layer
    - dropout rate
    - activation functions
- learning tuning

Tuning the learning process (optimizer)

- `optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']`
- learning rate
- batch size
- momentum
- weight decay

Papers

- *"A disciplined approach to neural network hyper-parameters: Part 1"*
- *"Hyper-Parameter Optimization: A Review of Algorithms and Applications"*

- Random Search
- Grid Search
- Bayesian Optimization - tuning with Gaussian process

Getting started with KerasTuner

Saving/loading a trained model

Save the model and the weights:

```
model.save("model.keras")
```

load the model and the weights

```
model = keras.saving.load_model("model.keras")
```

Save just the weights:

```
model.save_weights("model.weights.h5")
```

load the weights

```
model.load_weights("model.weights.h5")
```

Note: the architecture should be the same as when the weights were saved.

ModelCheckpoint: to save the Keras model or model weights at some frequency.

```python
# Model is saved at the end of every epoch if is the best seen so far
model_checkpoint_callback = keras.callbacks.ModelCheckpoint(
    filepath='./models',
    monitor='val_loss',
    mode='min',
    save_best_only=True)

history = model.fit(callbacks=[model_checkpoint_callback])
```