Deep Learning

COSC440

Andy Ming / Quelldateien

Table of contents

Details

Science of Arrays

Don't loop over elements in a array. Use numpy functions to do elementwise operations:

```
# Elementwise sum; both produce an array
z = x + y
z = np.add(x, y)
```

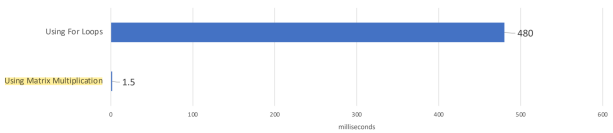
Use **Boradcasting** to work with arrays of different sizes. In Hardware data is take from the same memory space multiple times.

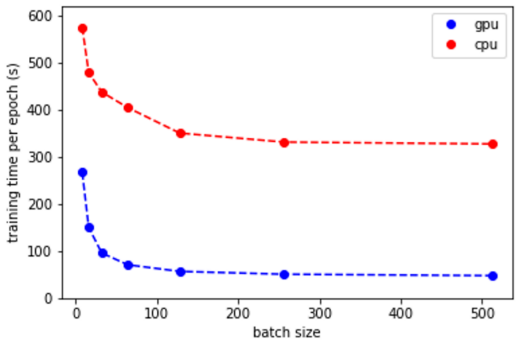
```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11,
↪  12]])
v = np.array([1, 0, 2])
y = x + v.T  # Add v to each row of x using
↪  broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Do **Matrix Multiplications**, remember that matrices of shape $100x20 \times 20x40$ equal a output shape of $100x40$:

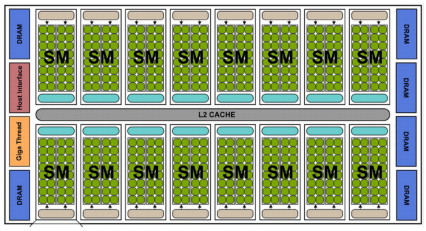
```
C = np.dot(A,B)
F = np.matmul(D,E)
```

The **Reason** is that this code is optimised for fast computation. Manly due to the utilisation of GPUs which offer high parallelism. *Don't bother trying to implement a faster version.*

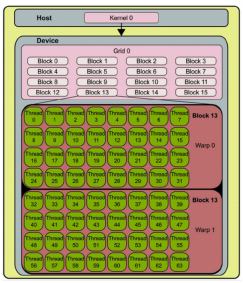




Architecture of a CUDA-capable GPU



CUDA compute model



Morals of AI

Gender Shades

Gender detecting algorithm is trained on a highly biased dataset, which leads to different results, depending on the target. Read More

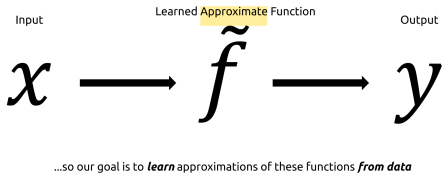
Gender Classifier	Darker Male	Darker Female	Lighter Male	Lighter Female	Largest Gap
Microsoft	94.0%	79.2%	100%	98.3%	20.8%
FACE++	99.3%	65.5%	99.2%	94.0%	33.8%
IBM	88.0%	65.3%	99.7%	92.9%	34.4%

Reknognition

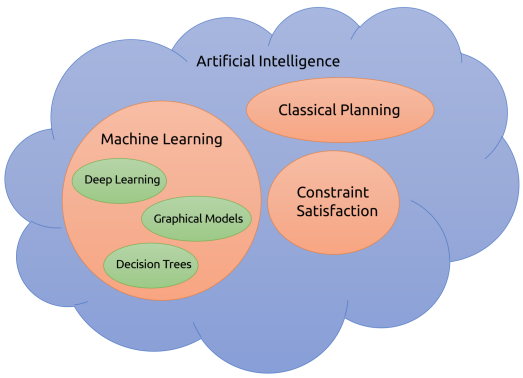
Facial Recognition algorithm wrongly matches government members to mugshots from a criminal database. The algorithm had **5% false positives** which isn't to bad, but when deployed states big issues of wrongly accusing innocent people. Read More

Machine Learning Concepts

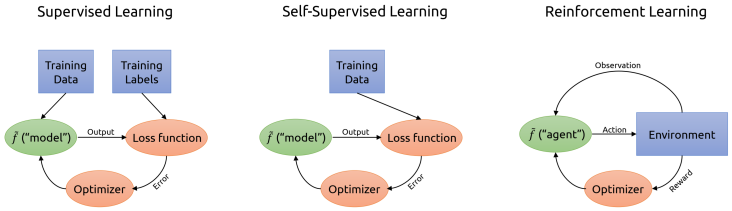
Machine Learning == Function Approximation

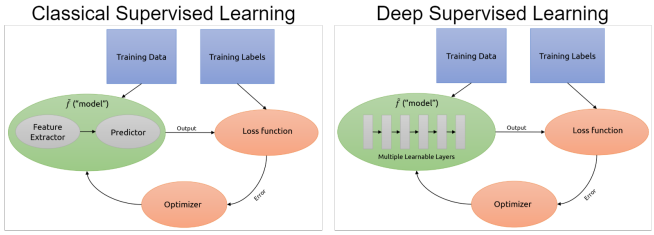


...so our goal is to ***learn*** approximations of these functions ***from data***



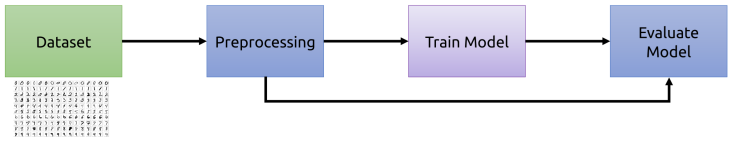
Types of Learning





Types of Problems

Maschine Learning Pipeline



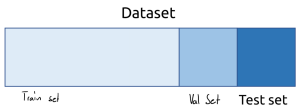
Dataset

Annotated Datasets like MNIST (Handwritten digits).

Preprocessing

Split the dataset into **Train, Validation, and Test sets**

- ***Train set*** — used to adjust the parameters of the model
- ***Validation set*** — used to test how well we're doing as we develop
  - Prevents ***overfitting***, something you will learn later!
- ***Test set*** — used to evaluate the model once the model is done



Train Model

1. **Initialization**: Set all weights w_i to 0.
2. **Iteration Process**:
   - Repeat for N iterations, or until the weights no longer change:
     - For each training example $\mathbf{x}^k$ with label a^k:
       1. Calculate the prediction error:
          * If $a^k - f(\mathbf{x}^k) = 0$, continue (no change to weights).
       2. Otherwise, update each weight w_i using:

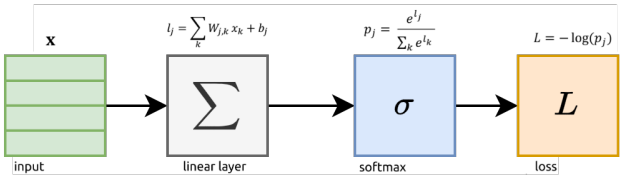
$$w_i = w_i + \lambda \left(a^k - f(\mathbf{x}^k)\right) x_i^k$$

   - where λ is a value between 0 and 1, representing the learning rate.

Optimizing with Gradient Descent

Loss Function

Function L which measures how "wrong" a network is. We want our network to answer right with **high probability**.

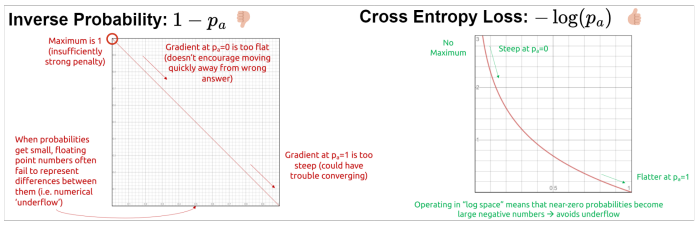


To get a probability for **binary classification**, we introduce a **probability layer**. One of the possible function is **Softmax**

$$p_j = \frac{e^{l_j}}{\sum_k e^{l_k}}$$

For every output j it takes every logit (output of network before activation/probability is applied) l_j in the exponent to ensure positivity. Dividing it by the sum of all logits ensures that $\sum_k p_k = 1$.

To get the loss L we apply a loss-function, *low probability* $\rightarrow$ *high loss*. We use **Cross Entropy Loss**

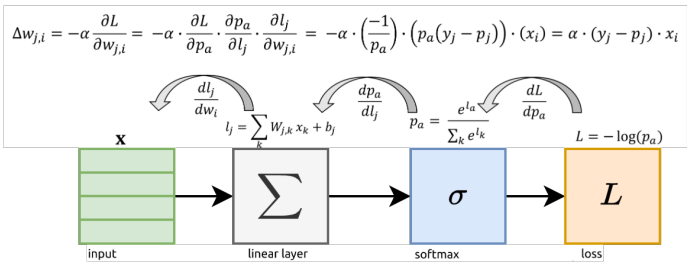


Gradient Descent

$\Delta w_{j,i} = -\alpha \frac{\partial L}{\partial w_{j,i}}$

α: learning rate *(typically 0.1-0.001)*
L: loss function
$w_{j,i}$: one single weight

To compute $-\alpha \frac{\partial L}{\partial w_{j,i}}$ use the chain rule




```
## Backpropagation on batch learning
# y = expected - (f(x)>0)
labels_OH = np.zeros((labels.size, self.num_classes),
↪  dtype=int)
labels_OH[np.arange(labels.size),labels] = 1  #
↪  One-Hot encoding
predictions = np.argmax(outputs, axis=1)
predictions_OH = np.zeros_like(outputs)
predictions_OH[np.arange(outputs.shape[0]),
↪  predictions] = 1
y = labels_OH - predictions_OH
# db = y*1
gradB = np.mean(y, axis=0)   # average over batch
# dW = y*x
y = y.reshape((outputs.shape[0],1,self.num_classes))
inputs =
↪  inputs.reshape((outputs.shape[0],self.input_size[0]*self
dW = inputs*y
gradW = np.mean(dW, axis=0)  # average over batch
```

Stochastic Gradient Descent (SGD)

Train a network on **batches**, small subsets of training data.

```
# Stochastic Gradient Descent
for start in range(0, len(train_inputs),
↪  model.batch_size):
    inputs =
↪  train_inputs[start:start+model.batch_size]
    labels =
↪  train_labels[start:start+model.batch_size]
    # For every batch, compute then descend the
    ↪  gradients for the model's weights
    outputs = model.call(inputs)
    gradientsW, gradientsB =
↪  model.back_propagation(inputs, outputs, labels)
    model.gradient_descent(gradientsW, gradientsB)
```

- Training process is *stochastic* / *non-deterministic*: batches are a random subsample.
- The gradient of a random-sampled batch is a unbiased estimator of the overall gradient of the dataset.
- Pick a large enough batch size for *stable updates*, but small enough to *fit your GPU*

Optimization

Automatic Differentiation

To avoid having to recalculate the whole chain every time a new layer is added, we use *automatic derivation*. There are several options:

Numeric differentiation

- $\frac{df}{dx} \approx \frac{f(x+\Delta x)-f(x)}{\Delta x}$
- Called *finite differences*
- Easy to implement
- Arbitrarily inaccurate/unstable

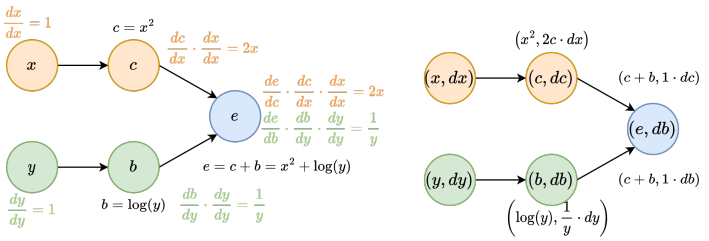
Symbolic differentiation

- $\frac{dx^2}{dx} = 2x$
- Computer does algebra and simplifies expressions
- Very exact
- Complex to implement
- Only handles static expressions

Automatic differentiation

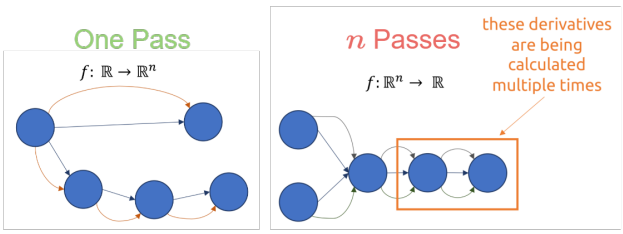
- Use the chain rule at runtime
- Gives exact results
- Handles dynamics
- Easier to implement
- Can't simplify expressions

Forward Mode Autodiff Every node stores its `(value, derivative)` in a tuple, called **dual numbers**. To compute the overall derivative, each derivative can be chained up. This is implemented via **Overloading**, every function / operator has multiple definitions based on the types of the arguments. ML-Framwork functions work on these tuples.

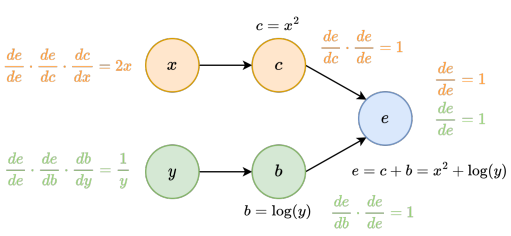


Time Effect: $O(N * M)$ time, $O(1)$ memory, with N = number of inputs, with M = number of nodes

Issue w/ forward mode

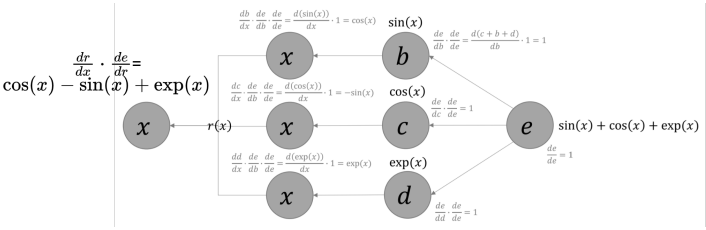


Reverse Mode Autodiff First, run the function to produce the graph, then compute the **derivatives backward**.



- Analog to the forward mode: overload math functions/operators
- Overloaded function return *Node* objects
- Overloaded functions build compute graph while executing
- After forward pass, the operations are recorded
- The backwards pass walks along the graph and computes the derivatives
- **Time Effect**: $O(M)$ time, $O(M)$ memory, with M = number of nodes

Fan-Outs (Reverse) The way to handle fan-out is to **add** the derivatives of the fanned-out nodes through replication $r(x)$.



Diagnosis Problems

Deep Learning Concepts

Common Misconception

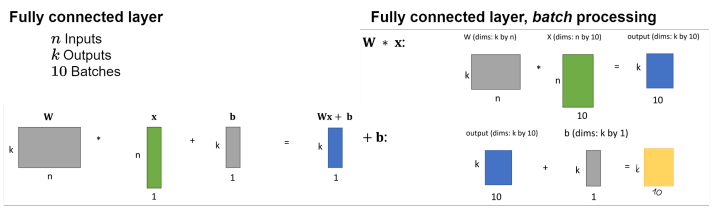
Deep Learning != AI, Just because deep learning algorithms are used doesn't mean there is any intelligence involved.
Deep Learning != Brain, Modern deep nets don't depend solely on *biologically mimiced neural nets* any more. A fully connected layer represents such a neural net the closest.
Deep Learning ==:

1. *Differentiable functions*, composed to more complex diff. func.
2. A deep net is a differentiable function, some inputs are *optimizable parameters*
3. Differentiable functions produce a computiation graph, which can be traversed backwards for *gradient-based optimization*

Multi-Dimensional Arrays & Memory Models

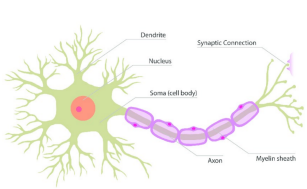
Vectorized Operations

For efficient operation, use **Matrices**.

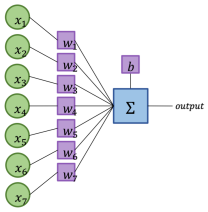


Neural Networks

Perceptron



Biological Neuron



Artificial Neuron (Perceptron)

Predicting with a Perceptron:

1. Multiply the inputs x_i by their corresponding weight w_i
2. Add the bias b
3. **Binary Classifier**, greater than 0, return 1, else return 0

$$f_\Phi(\mathbf{x}) = \begin{cases} 1, & \text{if } b + \mathbf{w} \cdot \mathbf{x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

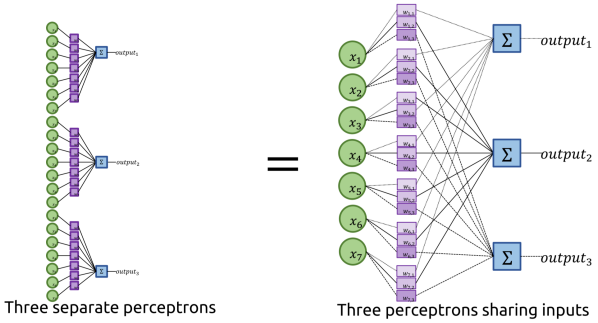
Parameters

Weights: "importance of the input to the output"
- Weight near 0: Input has little meaning to the output
- Negative weight: Increasing input → decreasing output

Bias: "a priori likelihood of positive class"
- Ensures that even if all inputs are 0, there is some result
- Can also be written as a weight for a constant 1 input

$$[x_0, x_1, x_2, \ldots, x_n] \cdot [w_0, w_1, w_2, \ldots, w_n] + b$$
$$= [x_0, x_1, x_2, \ldots, x_n, 1] \cdot [w_0, w_1, w_2, \ldots, w_n, b]$$

Multi-Class Perceptron



Three separate perceptrons

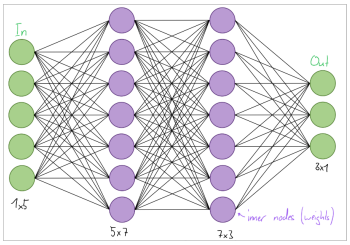
Three perceptrons sharing inputs

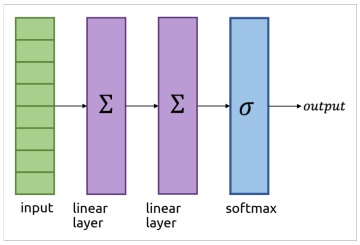
Biary Classifier: Only one output can be active $\hat{y} = \text{argmax}\left(f\left(x^k\right)\right)$, thus the update terms are

$$\Delta w_i = \begin{cases} 0, & \text{for } a^k = \hat{y} \\ -x_i^k, & \text{for } \hat{y} = 1, a^k = 0 \\ x_i^k, & \text{for } \hat{y} = 0, a^k = 1 \end{cases}$$

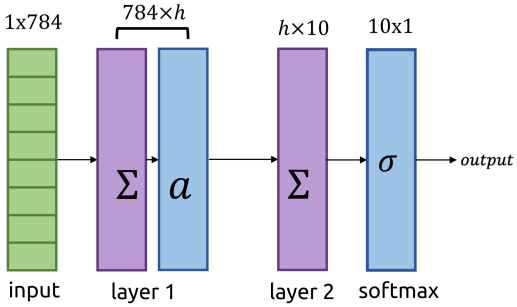
Multi-Layer

Through adding hidden layers we can make bigger networks and add more states to the algorithm.





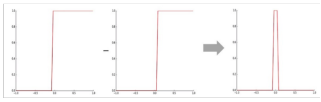
The size of these **hidden layers** are defined by the **hyperparameter**. These define the configuration of a model and are set before training begins. *Rule of Thumb*: Make hidden layers the same size as the input, then start to tweak to see the effect. If you have more time and money, check this.

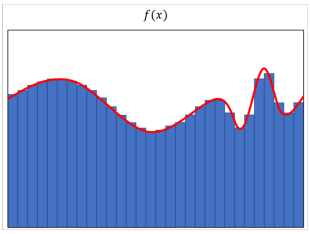


Universal Approximation Theorem

Remarkably, a one-hidden-layer network can actually represent any function (under the following assumptions):
- Function is continuous
- We are modeling the function over a closed, bounded subset of $\mathbb{R}^n$
- Activation function is sigmoidal (i.e. bounded and monotonic)

Proof: Any function can be approximated by boxes (Riemann Sums).
A box is just the difference of two sigmoids.





Stacking Linear Layers isn't Enough

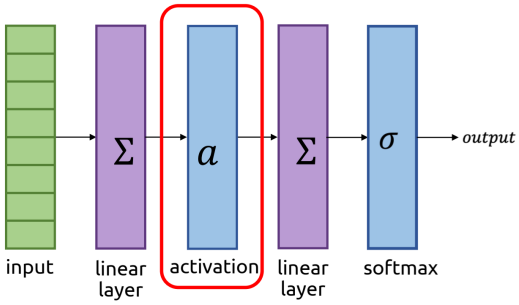
When simplifying the linear equation we get

$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\left(\begin{bmatrix} w_1 & b_1 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)\right) = \sigma\left(\begin{bmatrix} w_{12} & b_{12} \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Which is exactly the same as just one layer again, we need **activation**.

Activation Functions

We introduce a **nonlinear** layer



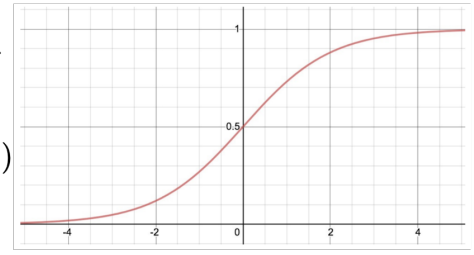
A activation function binds network outputs to a particular range. In the last layer this can be used to restrict the range, for example *age is strictly positive*.

Futher PyTorch activation functions can be found here.

Sigmoid .

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

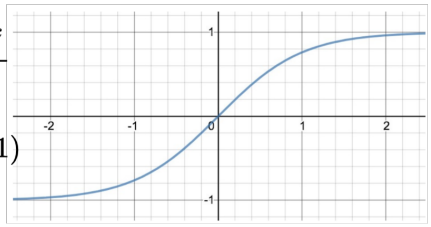
$$\sigma(x) : \mathbb{R} \to (0,1)$$



Tanh .

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{x}}$$

$$\tanh(x) : \mathbb{R} \to (-1,1)$$



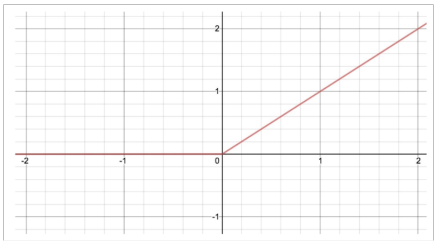
Vanishing Gradient

The problem with **Sigmoid** and **Tanh** is that the further away the parameters get from zero, the smaller is the gradient. Thus the network stops learning at these points. When **stacking layers** the issue gets even more severe.

ReLU .

Rectifies Linear Unit

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & \text{else} \end{cases}$$

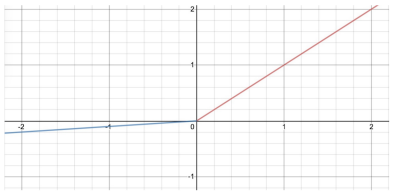


Dead ReLU

Because the negative part fed into the activation will result in a 0 output. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again.

Leaky ReLU To tackle a possible *dead ReLU* issue, we use a tiny positive slope for negative inputs.

$$f(x) = \begin{cases} x, & x > 0 \\ ax, & else \end{cases}$$



Convolution

Convolution is like a *"partially connected"* layer. Only certain inputs are connected to certain output pixels.

To introduce **translational invariance** $f(T(x)) = f(x)$ we apply convolutions. These are "Filters" which highlight different structures, the following network makes sense from the structures, not the pixels itself. Main application: **Computer Vision**.

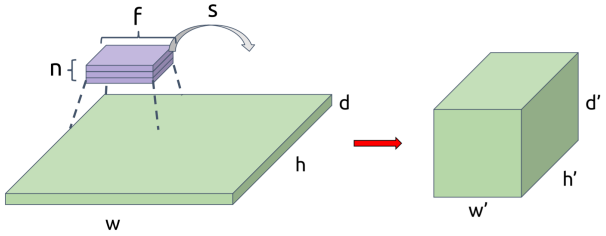
$$V(x,y) = (I \otimes K)(x,y) = \sum_m \sum_n I(x+m, y+n) K(m,n)$$

The output at pixel (x,y) — "Image I convolved with kernel K" — Sum over kernel rows — Sum over kernel columns — Multiply kernel value with corresponding image pixel value

Hyperparameters

There are 4 hyperparameters for the convolution

- Number of filters, n
- Size of these filters, n
- The Stride, s
- Amount of padding, p



We can calculate the output size through

$$w' = \frac{w - f + 2p}{s} + 1, h' = \frac{h - f + 2p}{s} + 1, d' = n$$

For **VALID** padding $p = 0$, for **SAME** padding p is chosen so output is same

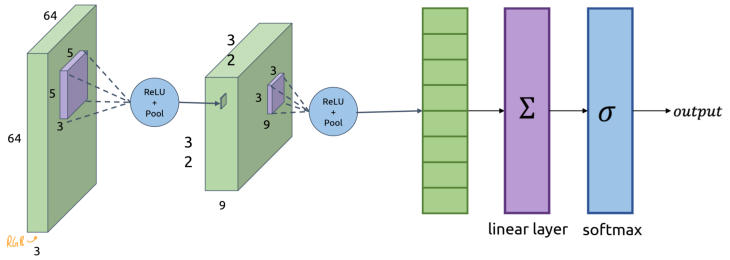
```
# Execute manual Convolution
```

```
# Should be of shape (batch_sz, 32, 32, 3) for CIFAR10
↪
inputs = CIFAR_image_batch
# Sets up a 5x5 filter with 3 input channels and 16
↪  output channels
self.filter = tf.Variable(tf.random.normal([5, 5, 3,
↪  16], stddev=0.1))
# Convolves the input batch with our defined filter
conv = tf.nn.conv2d(inputs, self.filter, [1, 2, 2, 1],
↪  padding="SAME")
```

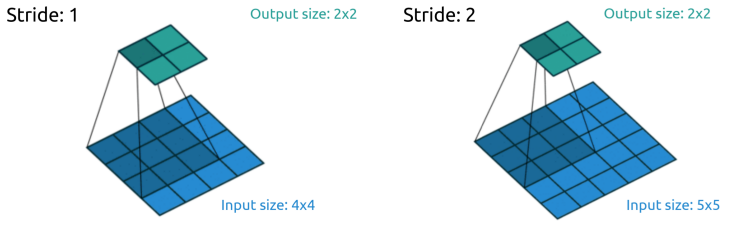
The inputs to `tf.nn.conv2d(...)` are:

- `input` = [batchSz, input_height, input_width, input_channels]
- `filter` = [f_height, f_width, in_channels, out_channels]
- `strides` = [batch_stride, stride_along_height, stride_along_width, stride_along_input_channels]
- `padding` = either `'SAME'` or `'VALID'`

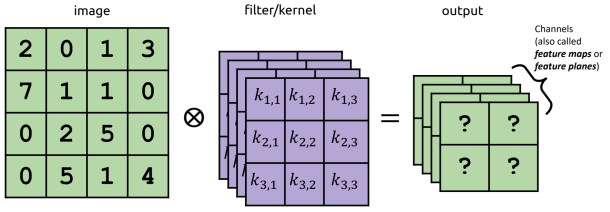
Tipically there are several convolutional layers and then a fully connected layer. This can be achieved through flattening a layer `flat = tf.reshape(conv, [conv.shape[0],-1])`.



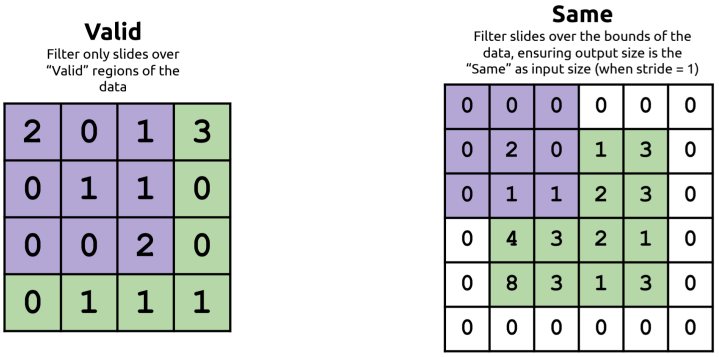
Stride The distance we slide a filter on each iteration is called **stride**. With a bigger stride, you compress a same size input into a smaller output. This decreases the image resolution controlled, **Downsampling**. The filters are **Kernels** and are made of **learnable parameters**.



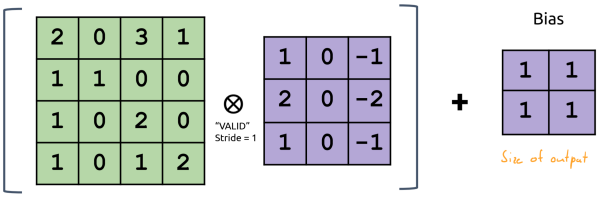
Futhermore, use several kernels per image, this block of kernels is a **filter bank**. The output is then a **mutli-channel** image. Multiple filters are able to extract *different features* of the image.



Padding To not loose resolution through a convolution, the original image has to be extended, **padded**. There are two convolution options, **VALID**, which is without padding, or **SAME** which is padding so that the output size is same as the input size.

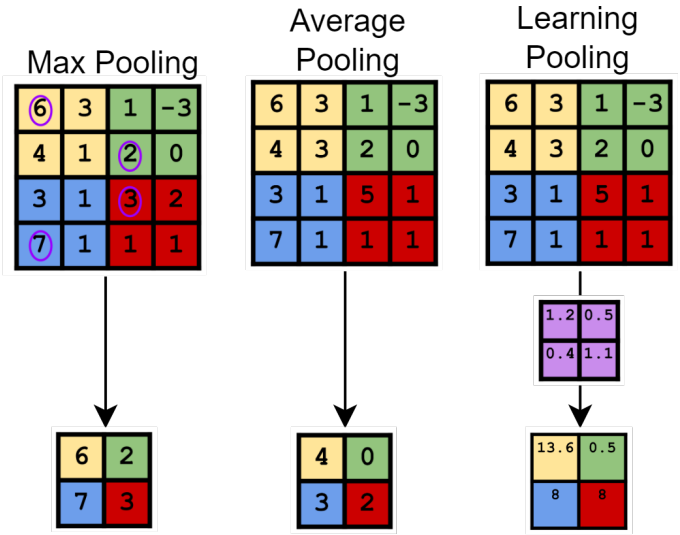


Bias As with other layers, a **Bias** can be added to the convolutional layer



This can be done through `tf.nn.bias_add(value, bias)`. When using keras layers, a bias is included by default `tf.keras.layers.Conv2D(filters, kernel_sz, strides, padding, use_bias = True)`.

Pooling Pooling keeps track of the regions with the highest activation, indicating object presence, also lowers the resolution in a controllable way.



Invariances

The translational variance is largely eliminated with the introduction of convolutions, this can be further improved by introduction of antialising.

There are further invariances, which can hurt a CNNs performance. CNNs don't do to good on these, for good performance, lots of training is needed.

Sequential and Recurrent Networks

Latent Space

Transfer Learning

Training Methods and Tricks

Deep Learning Problems, Models & Research

Computer Graphics and Vision

Natural Language

Audio and Video Synthesis

Search using Deep Reinforcment Learning

Anomaly Detection

Irregular Networks