

# Deep Learning

COSC440

Andy Ming / [Quelldateien](#)

## Table of contents

<b>Details</b>	<b>1</b>
Science of Arrays . . . . .	1
Morals of AI . . . . .	2
Gender Shades . . . . .	2
Reknognition . . . . .	2
<b>Machine Learning Concepts</b>	<b>2</b>
Types of Learning . . . . .	2
Self-Supervised Learning . . . . .	2
Types of Problems . . . . .	3
Maschine Learning Pipeline . . . . .	3
Dataset . . . . .	3
Preprocessing . . . . .	3
Train Model . . . . .	3
Optimizing with Gradient Descent . . . . .	3
Loss Function . . . . .	3
Gradient Descent . . . . .	3
Stochastic Gradient Descent (SGD) . . . . .	3
Adaptive Momentum Estimation (Adam) . . . . .	4
Automatic Differentiation . . . . .	4
Numeric differentiation . . . . .	4
Symbolic differentiation . . . . .	4
Automatic differentiation . . . . .	4
Diagnosis Problems . . . . .	5
Overfitting . . . . .	5
Regularization . . . . .	5
<b>Deep Learning Concepts</b>	<b>5</b>
Multi-Dimensional Arrays & Memory Models . . . . .	5
Vectorized Operations . . . . .	5
Neural Networks . . . . .	5
Perceptron . . . . .	5
Multi-Layer . . . . .	5
Activation Functions . . . . .	6
Convolution . . . . .	6
Pooling . . . . .	9
Invariances . . . . .	9
Sequential and Recurrent Networks . . . . .	9
Latent Space . . . . .	9
Principal Component Analysis (PCA) . . . . .	9
Butterfly-Network (Autoencoder) . . . . .	10
Convolutional Autoencoder . . . . .	10
Autoencoder Applications . . . . .	10
Transfer Learning . . . . .	10
Fine Tuning . . . . .	10
Zero Training . . . . .	11
Few-Shot Learning . . . . .	11
Training Methods and Tricks . . . . .	11
Early Stopping . . . . .	11

Reduce Parameters . . . . .	11
Data Agumentation . . . . .	11
Dropout . . . . .	12
Skip Connections / Residual Blocks . . . . .	12
Batch Normalisation . . . . .	12
Checkpointing . . . . .	12
Xavier Initialisation . . . . .	13
Keras . . . . .	13
Tensorboard (Visualization) . . . . .	13

<b>Deep Learning Problems, Models &amp; Research</b>	<b>13</b>
Computer Graphics and Vision . . . . .	13
Denoising . . . . .	13
Diffusion Models . . . . .	13
Attention . . . . .	13
Natural Language . . . . .	14
Attention . . . . .	14
Self-Attention . . . . .	14
Multi-Head Attention . . . . .	15
Word Vectors . . . . .	15
Transformer . . . . .	15
Audio and Video Synthesis . . . . .	16
Search using Deep Reinforcement Learning . . . . .	16
Anomaly Detection . . . . .	16
Irregular Networks . . . . .	16

## 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)
```

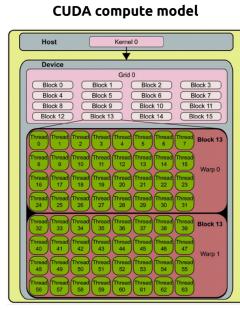
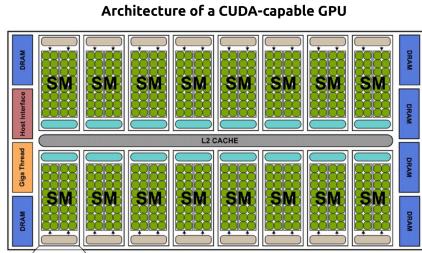
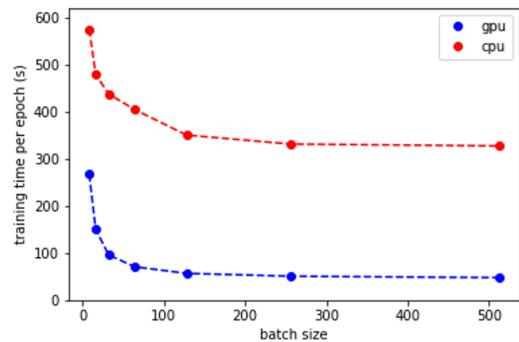
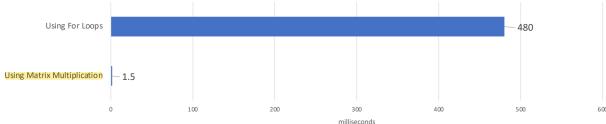
Use **Broadcasting** to work with arrays of different sizes. In Hardware data is take from the same memory space multiple times.

```
# 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  $100 \times 20 \times 20 \times 40$  equal a output shape of  $100 \times 40$ :

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

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



## Morals of AI

### Gender Shades

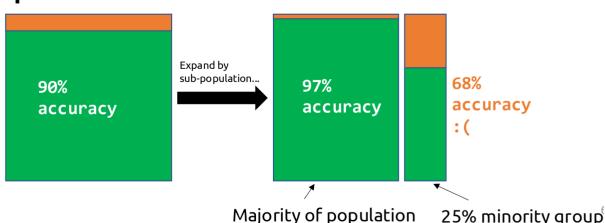
Gender detecting algorithm is trained on a highly biased dataset, which leads to different results, depending on the target. [Read More](#)

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%

### Beyond Average Test-Set Performance

Even if a test-set is a well representation of the real world, which will lead to good average accuracy. The network can still do badly on minority group test sets.

**Good average performance can mask poor performance of specific cases**



## Reknognition

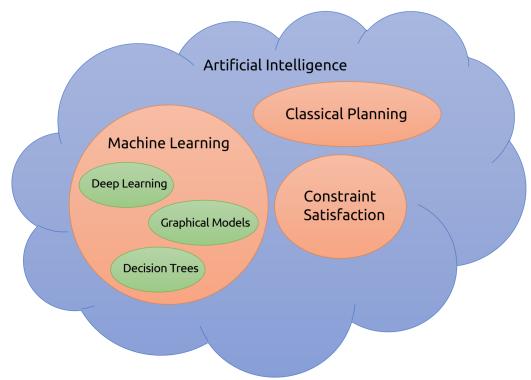
Facial Recognition algorithm wrongly matches government members to mugshots from a criminal database. The algorithm had **5% false positives** which isn't too bad, but when deployed states big issues of wrongly accusing innocent people. [Read More](#)

## Machine Learning Concepts

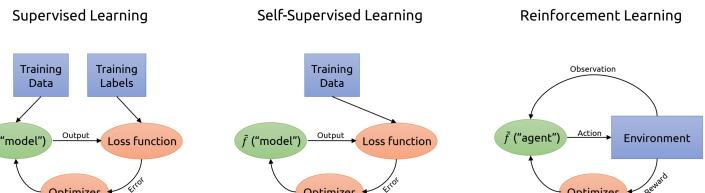
### Machine Learning == Function Approximation

$$\text{Input} \rightarrow \text{Learned Approximate Function} \rightarrow \tilde{f} \rightarrow \text{Output}$$

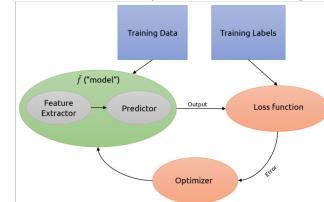
...so our goal is to *learn* approximations of these functions *from data*



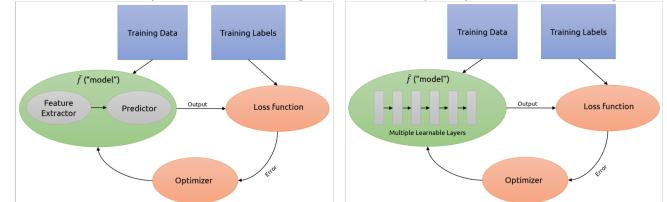
## Types of Learning



### Classical Supervised Learning



### Deep Supervised Learning



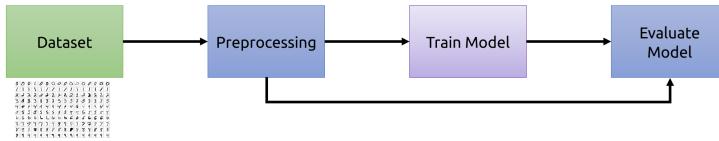
## Self-Supervised Learning

From data **without labels** we can learn the structure of the data itself, there are several approaches, the basic idea is dimensionality reduction:

- K-Means Clustering
- Principal Component Analysis (PCA)
- Butterfly-Network (Autoencoder)
- ...

## 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  $a^k$ :
    - Calculate the prediction error:  
\* If  $a^k - f(\mathbf{x}^k) = 0$ , continue (no change to weights).
    - Otherwise, update each weight  $w_i$  using:

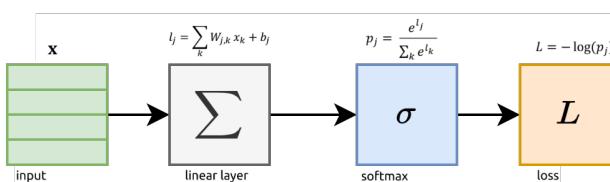
$$w_i = w_i + \lambda (a^k - f(\mathbf{x}^k)) x_i^k$$

- where  $\lambda$  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**.

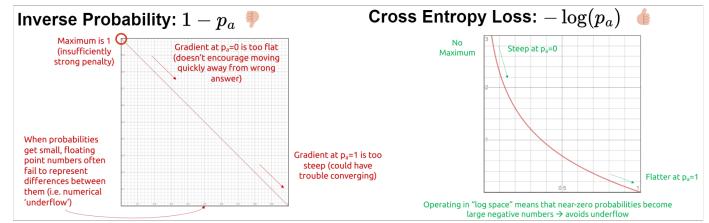


To get a probability for **binary classification**, we introduce a **probability layer**. One of the possible function is **Softmax**

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

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

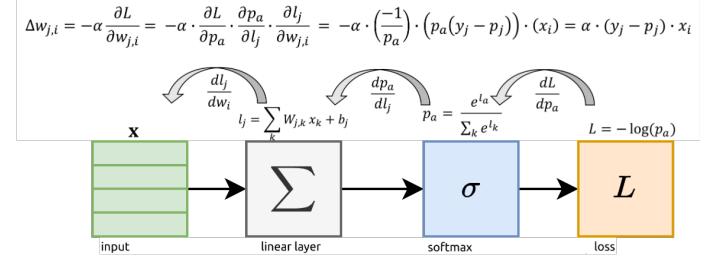
$$\Delta w_{j,i} = -\alpha \frac{\partial L}{\partial w_{j,i}}$$

$\alpha$ : 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_0H = np.zeros((labels.size, self.num_classes),
                     dtype=int)
labels_0H[np.arange(labels.size), labels] = 1 # One-Hot encoding
predictions = np.argmax(outputs, axis=1)
predictions_0H = np.zeros_like(outputs)
predictions_0H[np.arange(outputs.shape[0]),
              predictions] = 1
y = labels_0H - predictions_0H
# 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)

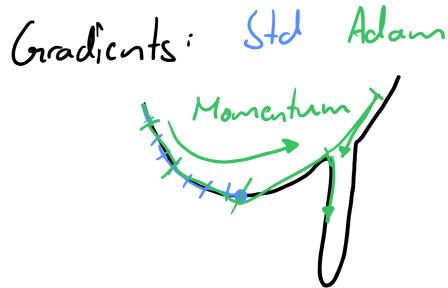
Train a network on **batches**, small subsets of training data.

```
# 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*

### ⚠️ Stuck Gradients

When the gradients get low or there is a local minima, SGD can get **Stuck**.



### Adaptive Momentum Estimation (Adam)

Two moments: SGD momentum and squared gradients. Also uses an exponentially decaying average. Fast and almost always the best, very little need to hyperparameter tune learning rate.

See [Notebook](#)

## 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

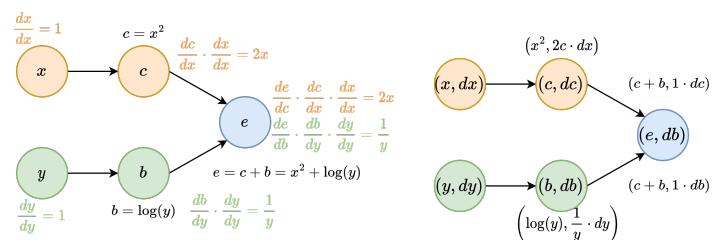
- $\frac{d^2x}{dx^2} = 2x$

- Computer does algebra and simplifies expressions
- Very exact
- Complex to implement
- Only handles static expressions

### Automatic differentiation

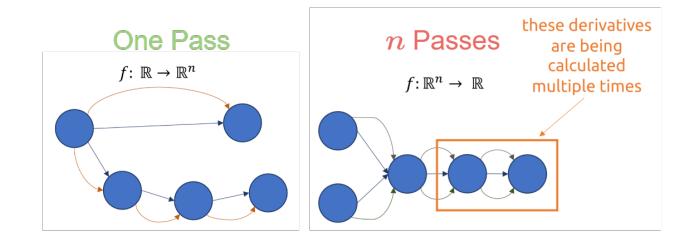
- Use the chain rule at runtime
- Gives exact results
- Handles dynamics
- Easier to implement
- Can't simplify expressions

**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-Framework functions work on these tuples.

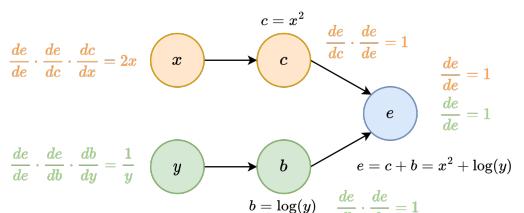


**Time Effect:**  $O(N * M)$  time,  $O(1)$  memory, with  $N$  = number of inputs, with  $M$  = number of nodes

#### 💡 Issue w/ forward mode



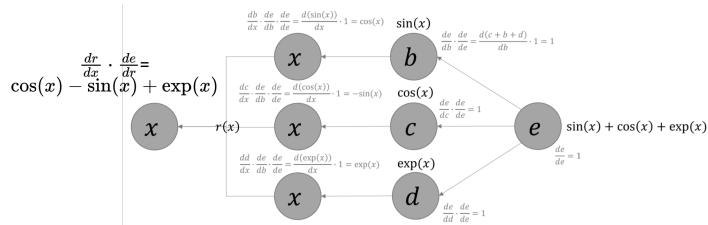
**Reverse Mode Autodiff** First, run the function to produce the graph, then compute the **derivatives backward**.



- 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  $M = \text{number of nodes}$

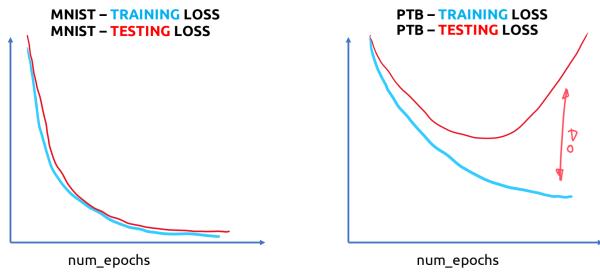
**Fan-Outs (Reverse)** The way to handle fan-out is to **add** the derivatives of the fanned-out nodes through replication  $r(x)$ .



## Diagnosis Problems

### Overfitting

Training on a complex dataset can lead to **overfitting** (PTB is a language dataset).



### Regularization

This approach **modifies the loss** through adding a additional term to our existing loss function.

#### L2 regularization

$$\lambda \sum_{j=1}^n |W_j|^2$$

**Penalize sum of squared weights**

**Effect:** keeps all weights small-ish, i.e. network can't learn to rely too heavily on any single pattern in the data

For both, this is a term added to the existing loss function.

$\lambda$  controls the strength of the penalty

#### L1 regularization

- $\lambda \sum_{j=1}^n |W_j|$
- Penalize absolute value of weights
- Effect:** tends to produce *sparse weights* (i.e. many zero-valued weights) → prevents the network from relying on too many different patterns in the data

Regularization can be applied to certain layers on Keras through `tf.keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l1(lambda), activation='relu')`.

## 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.

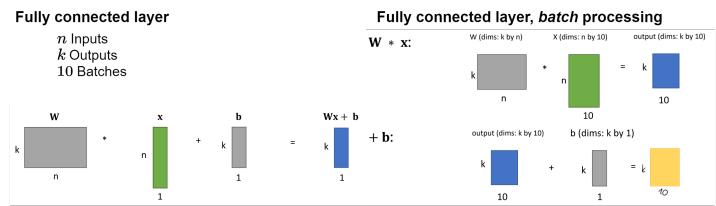
**Deep Learning ==:**

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

## Multi-Dimensional Arrays & Memory Models

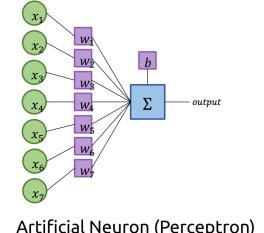
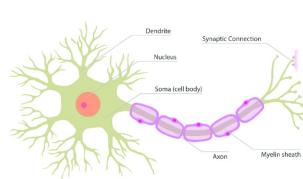
### Vectorized Operations

For efficient operation, use **Matrices**.



## Neural Networks

### Perceptron



### Predicting with a Perceptron:

- Multiply the inputs  $x_i$  by their corresponding weight  $w_i$
- Add the bias  $b$
- 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"

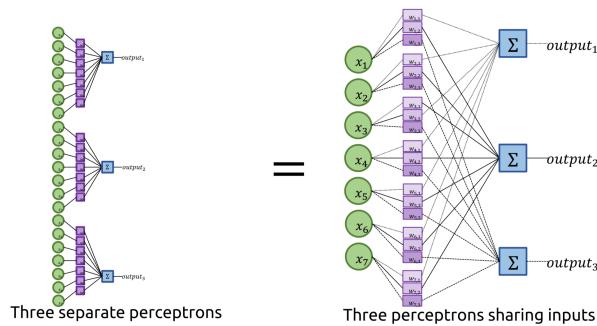
- Weight near 0: Input has little meaning to the output
- Negative weight: Increasing input → decreasing output

**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

$$\begin{aligned} & [x_0, x_1, x_2, \dots, x_n] \cdot [w_0, w_1, w_2, \dots, w_n] + b \\ &= [x_0, x_1, x_2, \dots, x_n, 1] \cdot [w_0, w_1, w_2, \dots, w_n, b] \end{aligned}$$

## Multi-Class Perceptron

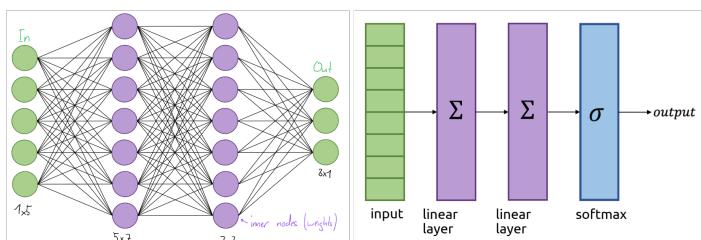


**Binary Classifier:** Only one output can be active  $\hat{y} = \text{argmax}(f(x^k))$ , thus the update terms are

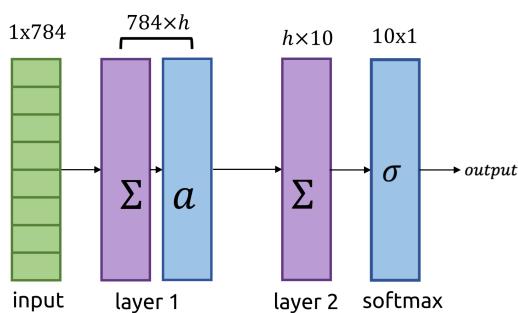
$$\Delta w_i = \begin{cases} 0, & \text{for } a^k = \hat{y} \\ -x_i^k, & \text{for } \hat{y} = 1, a^k = 0 \\ x_i^k, & \text{for } \hat{y} = 0, a^k = 1 \end{cases}$$

## 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](#).

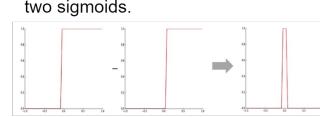


## i Universal Approximation Theorem

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.



$f(x)$

## ⚠ Stacking Linear Layers isn't Enough

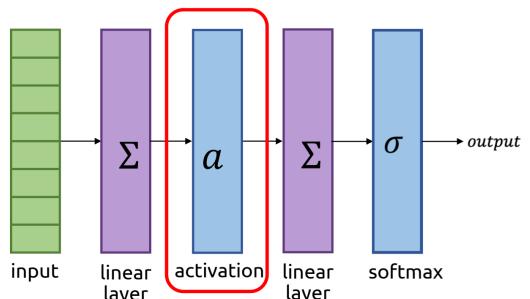
When simplifying the linear equation we get

$$\sigma([w_2 \ b_2]([w_1 \ b_1][\vec{x}])) = \sigma([w_{12} \ b_{12}][\vec{x}])$$

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

## Activation Functions

We introduce a **nonlinear** layer



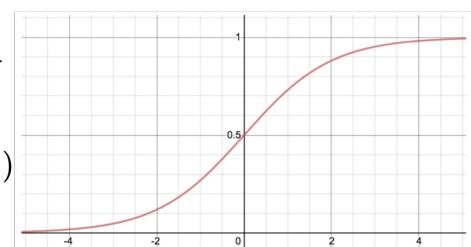
A activation function binds network outputs to a particular range. In the last layer this can be used to restrict the range, for example *age is strictly positive*.

Futher PyTorch activation functions can be found [here](#).

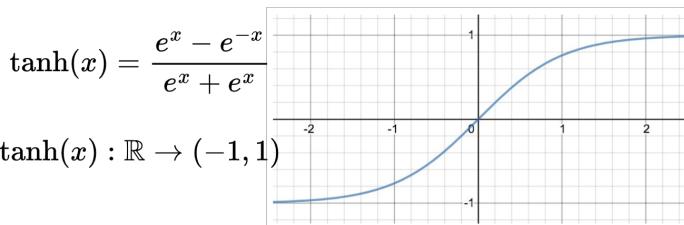
## Sigmoid

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

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



## Tanh



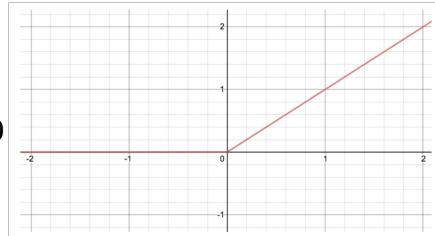
### ⚠ Vanishing Gradient

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

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

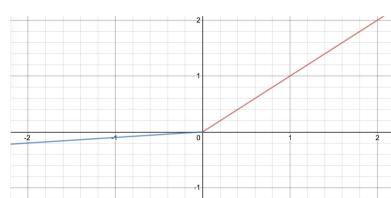


### ⚠ Dead ReLU

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.

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



### 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**.

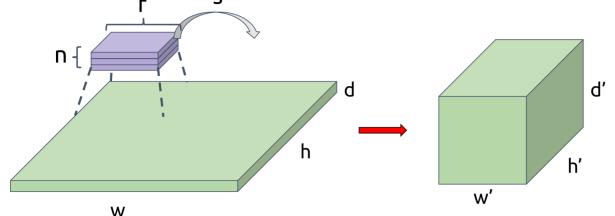
$$V(x, y) = (I \otimes K)(x, y) = \sum_m \sum_n I(x+m, y+n)K(m, n)$$

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

### ℹ 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

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

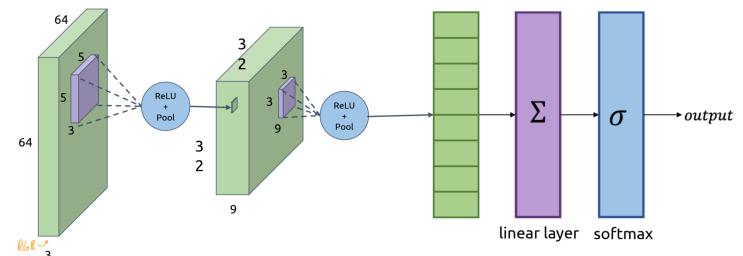
For **VALID** padding  $p = 0$ , for **SAME** padding  $p$  is chosen so output is same

```
# 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:

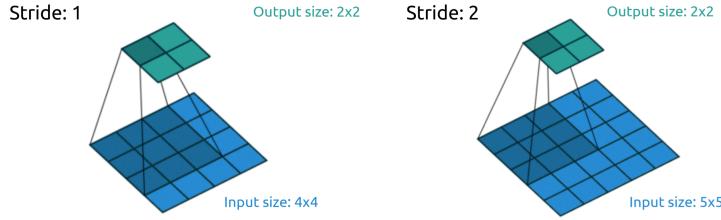
- `input` = [batchSz, input\_height, input\_width, input\_channels]
- `filter` = [f\_height, f\_width, in\_channels, out\_channels]
- `strides` = [batch\_stride, stride\_along\_height, stride\_along\_width, stride\_along\_input\_channels]
- `padding` = either 'SAME' or 'VALID'

Typically 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])`.

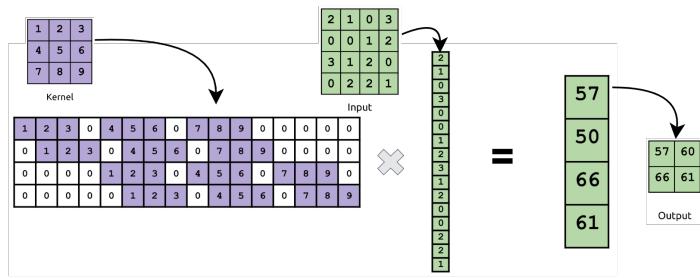


**Stride** The distance we slide a filter on each iteration is called **stride**. With a bigger stride, you compress a same size input into a smaller output. This decreases the image resolution controlled,

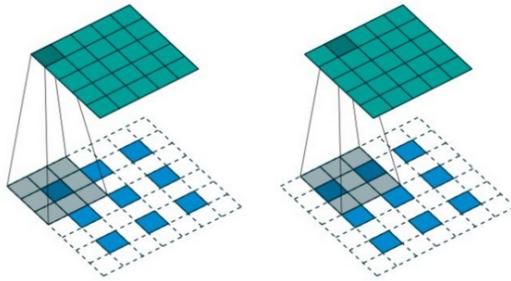
**Downsampling.** The filters are **Kernels** and are made of **learnable parameters**.



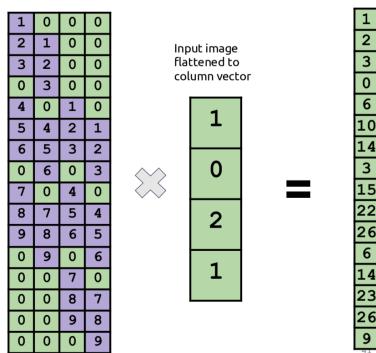
**Computation** The image is flattened and the kernel is unrolled into a bigger matrix. This leads to a normal **matrix/vector multiplication**



**Fractional Stride** For **deconvolutions** you can also use fractionally-strided convolutions (here  $\frac{1}{2}$  stride):



**Computation** Same as with the convolution, we flatten all matrices into vectors/matrices but now we **transpose** the kernel matrix, which gives us the **de-convolution**:



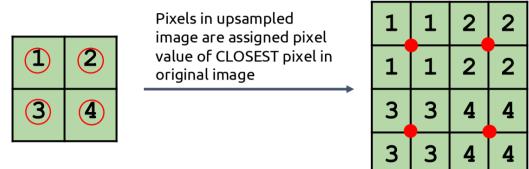
### 🔥 Checkerboard Artifacts

The transpose convolution causes **artifacts in output images** because some pixels get written more often than others (at the overlaps, which occur in a line).

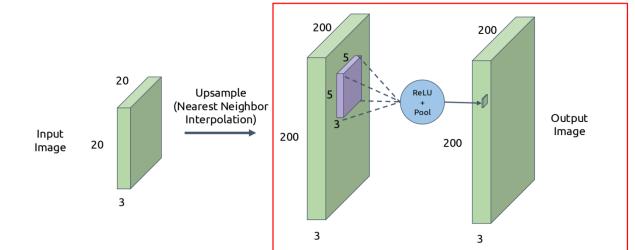


**Prevention:**

1. Upsampling using nearest neighbour interpolation

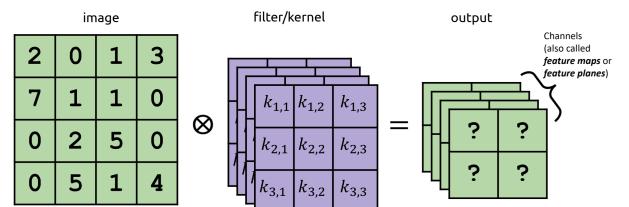


2. Perform a convolution with 'SAME' padding on up-sampled image

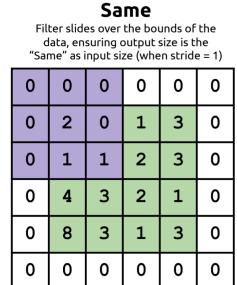
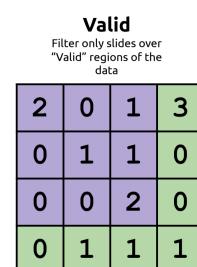


```
# Layer to upsample the image by a factor of 5
→ in x and y using nearest
# neighbor interpolation
tf.keras.layers.UpSampling2D(size=(5, 5),
→ interpolation='nearest')
# Do a convolutional layer on the result
tf.keras.layers.Conv2D(filters = 1, kernel_size
→ = (10,10), padding = "SAME")
```

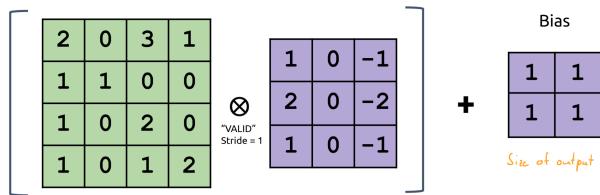
**Filter Banks** Furthermore, use several kernels per image, this block of kernels is a **filter bank**. The output is then a **multi-channel** image. Multiple filters are able to extract *different features* of the image.



**Padding** To not lose 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.



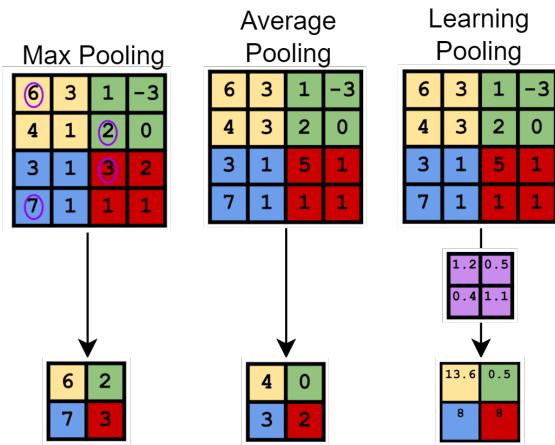
**Bias** As with other layers, a **Bias** can be added to the convolutional layer



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

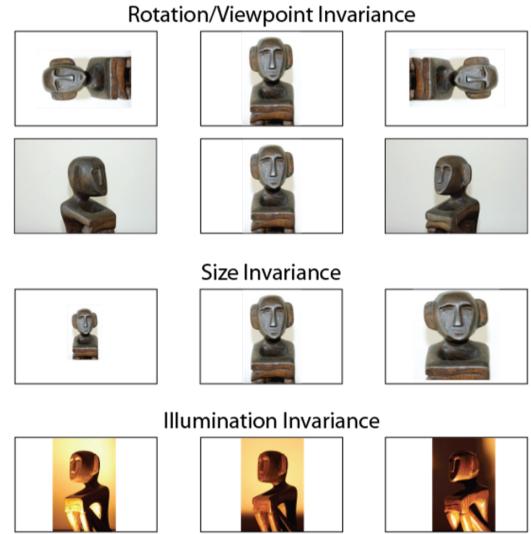
Pooling keeps track of the regions with the highest activation, indicating object presence, also lowers the resolution in a controllable way.



## Invariances

The translational variance is largely eliminated with the introduction of convolutions, this can be further improved by introduction of [antialiasing](#).

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



## Sequential and Recurrent Networks

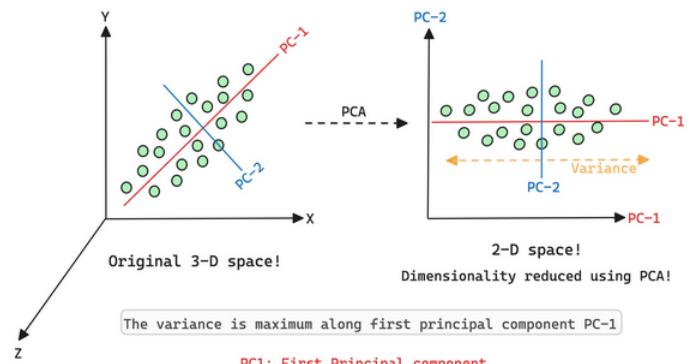
### Latent Space

**Latent Space** is a compact representation, representing the input in a lower dimension. This is used for **Self-Supervised Learning**.

Represent the data with fewer dimensions, although data might exist in high dimensional space, it actually may exist along a lower dimensional subspace (e.g. 2D-Plane in 3D-Space, Line in 2D-Space), **Dimensionality Reduction**.

We do this for **smaller dataset footprint (memory)**, more efficient search through **nearest neighbour algorithms**, many **clustering algorithms behave better** in lower dimensions, Easier to **visualize**.

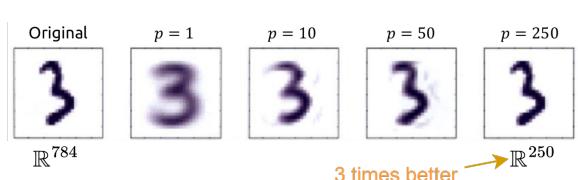
### Principal Component Analysis (PCA)



Given a dataset  $D$  of dimension  $n$  and a target dimension  $m \leq n$ , find  $m$  vectors in  $\mathbb{R}^n$  along which  $D$  has the highest variance.  $m$  are the **principal components**.

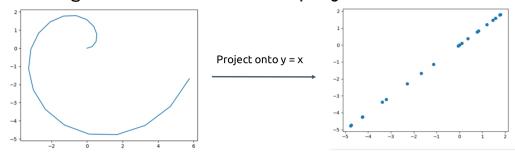
How: Find the direction of maximal variation, project onto this vector, repeat  $m$  times.

*Example with MNIST:*



## Limitations of PCA

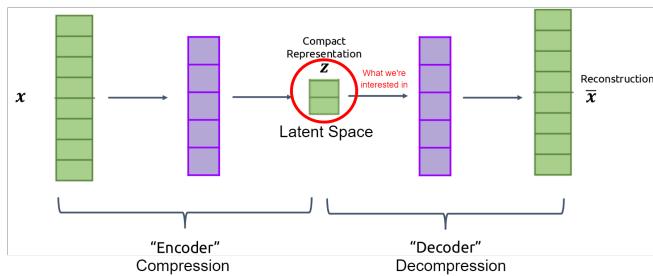
PCA can't figure out **non-linear** projections from  $\mathbb{R}^2$  to  $\mathbb{R}^1$ .



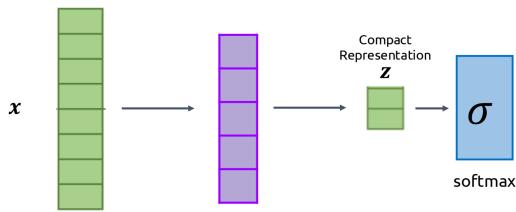
## Butterfly-Network (Autoencoder)

Because we don't have labels, we copy the first part of the network and inverse it. The loss is then calculated through

$$L(x, \bar{x}) = (x - \bar{x})^2 \quad \text{squared error loss}$$

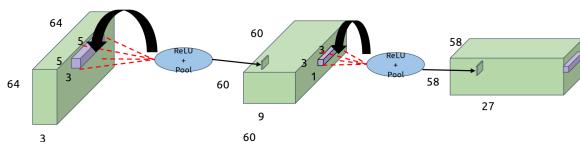


After training the network with the unlabelled data, the second part can be cut off and a binary layer be added. Now fine tune the last layer on **labelled data through supervised learning**.



## Convolutional Autoencoder

The same approach can be applied to convolutional networks. But **convolutional matrices are expensive**, thus we just swap the forward and backwards pass code for the **deconvolution**.



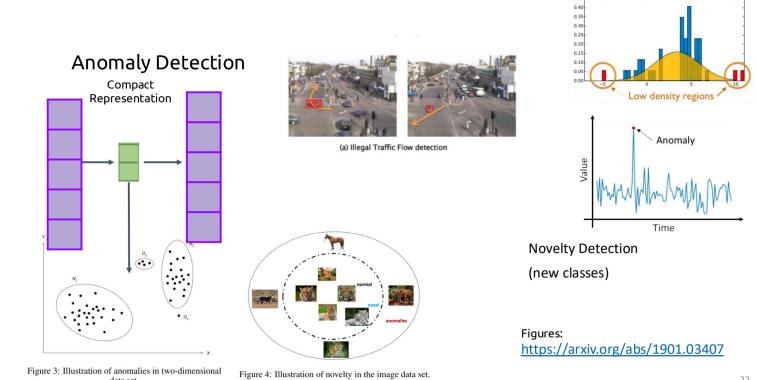
This can be done in [Tensorflow](#)

```
tf.nn.conv2d_transpose(input, filters, output_shape, strides, padding='SAME')

4D tensor of shape [batch, height, width, in_channels]
4-D Tensor with shape [height, width, output_channels, in_channels]
length 4 1D tensor representing the output shape.
Strides along each dimension (list of integers)
String representing type of padding
```

## Autoencoder Applications

### Anomaly / Novelty Detection



Figures:  
<https://arxiv.org/abs/1901.03407>

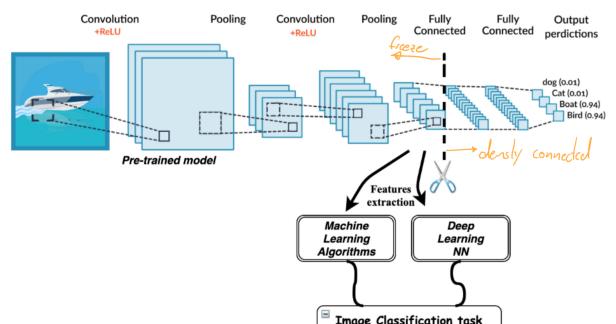
22

## Others

- Transformer
- Density-based techniques (k-nearest neighbor, local outlier factor, isolation forests, and many more variations of this concept)
- Subspace, correlation-based, and tensor-based outlier detection for highdimensional data
- One-class support vector machines
- Replicator neural networks, autoencoders, and long short-term memory neural networks
- Bayesian Networks
- Hidden Markov models (HMMs)
- Cluster analysis-based outlier detection
- Deviations from association rules and frequent itemsets
- Fuzzy logic-based outlier detection;

## Transfer Learning

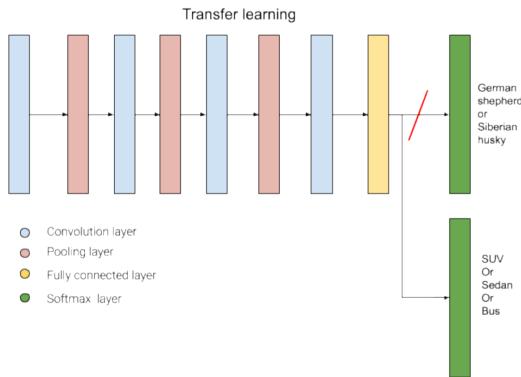
Taking a pre-trained network of a similar domain and adjust the last layers.



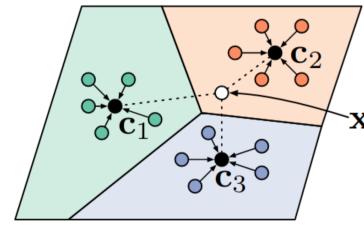
- Saves training time & money
- Network is more likely to generalise

## Fine Tuning

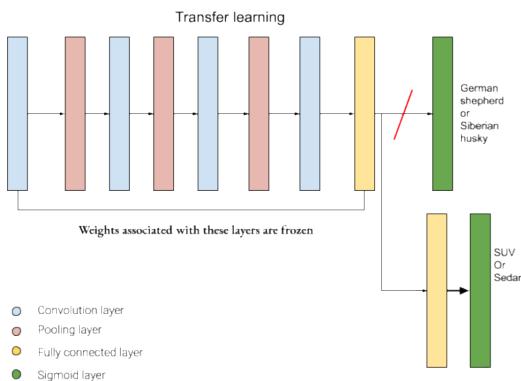
One can switch out the **last layer** and add a different classification



Nominate or Generate a **representative** from each class



Or also switch add/replace a fully connected layer if the differences in the domain are bigger



## Training Methods and Tricks

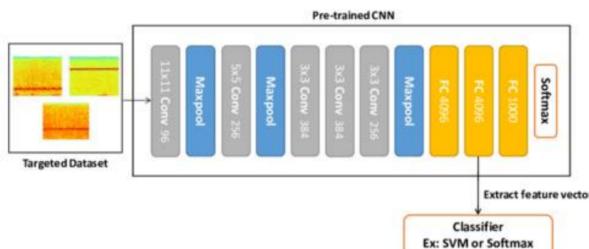
### Early Stopping

If one stops training when the testing loss is starting to rise, at least the loss won't get bigger

```
# Pseudo Code Early Stopping
curr_test_loss = inf
for i in range(n_epochs):
    train model()
    new_test_loss = model.get_test_loss()
    if new_test_loss > curr_test_loss:
        break
    else:
        curr_test_loss = new_test_loss
```

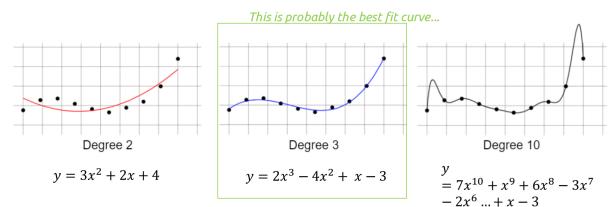
## Zero Training

The last layer can be replaced by a *Logistic Regression* or *Nearest Neighbor Classifier*. The *transferred network* is then just a feature extraction pipeline. Use a database with known targets and classify new data through searching for nearest neighbour.



### Reduce Parameters

Reducing parameters, means less possibilities to learn or even memorize a dataset.



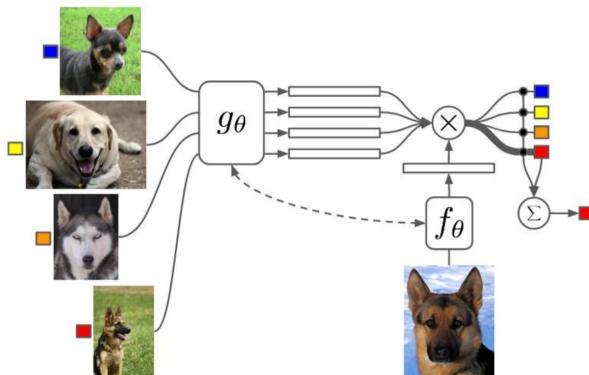
Reducing parameters can mean...

- ... reducing layer size
- ... decrease number of channels in a convolution
- ... decrease number of layers

### Reducing Parameters

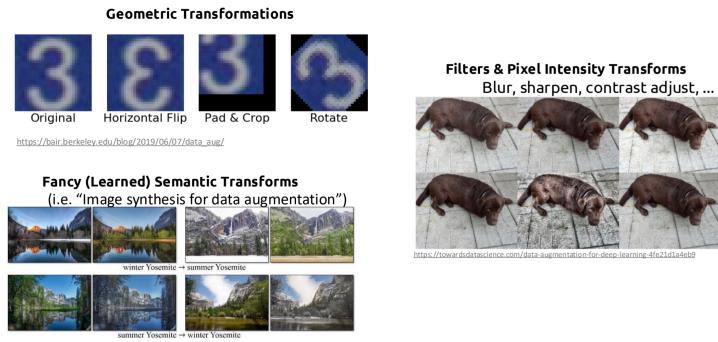
Can also be used to check if parts of a network **are actually needed** → remove part → retrain model → if it behaves the same, it wasn't needed.

Requires obnoxious tuning of hyperparameters.



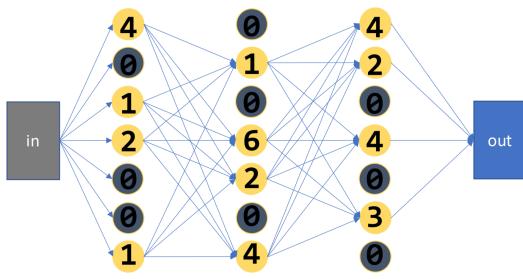
## Data Augmentation

Generate **random variations** on your training data.



## Dropout

*Make it harder for the network.* In a single training pass, the output of randomly selected nodes from each layer are set to 0. The nodes that drop out are **different each pass**. This builds **Resillience**. During testing all nodes are active again.

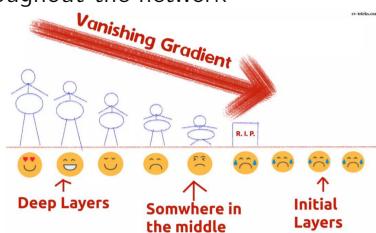


Dropout can be handled with Keras through `tf.keras.layers.dropout(rate)`, where `rate` is a hyperparameter between  $[0, 1]$ . `rate=0.5` is drop  $\frac{1}{2}$ , keep  $\frac{1}{2}$ . `rate=0.25` is drop  $\frac{1}{4}$ , keep  $\frac{3}{4}$ .

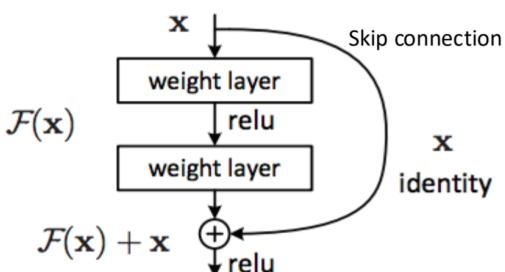
## Skip Connections / Residual Blocks

### Vanishing Gradients

The deeper a net gets (more layers) the more learnable parameters are present. This leads to vanishing of the gradient throughout the network



To mitigate the *vanishing gradient problem*, we use **Redidual Blocks**

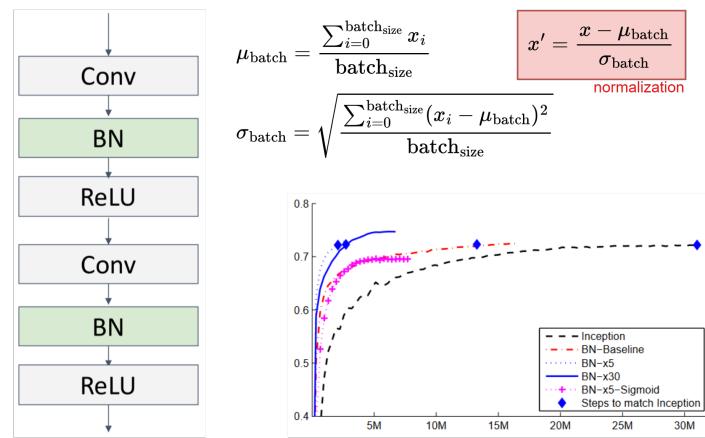


The output of each layer is the identity + some deviation (residual) from it. It allows the gradient to flow through two pathways. **Significantly stabilises training of very deep networks.**

## Batch Normalisation

The idea is to stabilise inputs, which should lead to **faster training**. This is done through normalisation of the layers' inputs by re-centring and re-scaling.

A **normalisation layer BN** is added after a fully connected or a convolution, before the non-linear activation. In Tensorflow this is done with `tf.keras.layers.BatchNormalization(input)`.



- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- **Not well-understood theoretically (yet)**
- **Behaves differently during training and testing: this is a very common source of bugs!**

## Checkpointing

### ⚠ Training is Expensive

Training takes a while and sometimes you want to "save" data for use in future instances.

**Checkpointing** allows you to save your Tensorflow model! No need to retrain every time you run your program. Fast prediction. Export your trained weights for use in other applications

```
checkpoint = tf.train.Checkpoint( ... )
```

- Trackable objects examples:
  - `tf.train.Variable`
  - `tf.train.Optimizer`
- Only restores Tensorflow variables, not Python variables

Example: (more [here](#))

```

counter = tf.Variable(1)
checkpoint = tf.train.Checkpoint(var = counter)
manager = tf.train.CheckpointManager(checkpoint, filepath)
counter.assign_add(2)
manager.save()
...
checkpoint.restore(manager.latest_checkpoint)

```

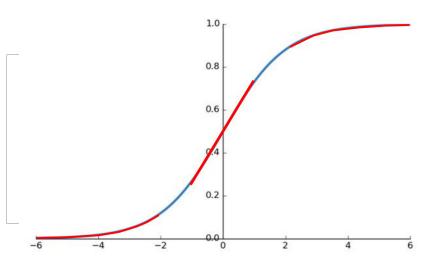
We will be tracking 'counter'  
Performing any operation on counter is linked to checkpoint  
Specify the directory you want the checkpoints to be in  
Restores checkpoint with counter = 3

33

## Xavier Initialisation

### ⚠ Bad Initialization

With certain activation functions, it's possible to get bad initialisations. For example with the sigmoid function we have the issue of linearity around  $x \approx 0$  and the flatness / low gradient at  $x > |4|$ .



One solution is to use activations which don't have these issues, like **ReLU**.

Even then we want to keep values in the same range when they flow through a network. Values that drastically fluctuate in magnitude can lead to **numerical instability** → slow convergence.

Consider a weight matrix  $W$  of size  $mn$ :

- One entry  $y_i$  of the product  $Wx$  is:  $y_i = W_{i,1}x_1 + \dots + W_{i,n}x_n$
- If  $n$  increases, then the magnitude of  $y_i$  would also tend to increase
- If  $m$  increases, then the output vector  $Wx$  would have a larger dimension
- As the output becomes the input for the next layer, the next layer would add up in terms

We want the magnitude of weights to be **inversely proportional** to  $m$  and  $n$ .

To tackle this issue, we use the **Xavier Initialization**, we calculate the *standard deviation* on each layer  $i$  new:

$$\sigma_i = \sqrt{\frac{2}{n_i + m_i}}$$

## Keras

Tensorflow code frequently gets terribly cumbersome, **Keras** **shortens the amount of code** you need to write through higher-level APIs for constructing, training, and evaluating models.

## Tensorboard (Visualization)

A powerful visualisation, logging, and monitoring tool designed to be integrated with Tensorflow (although you can technically use it with any Python code).

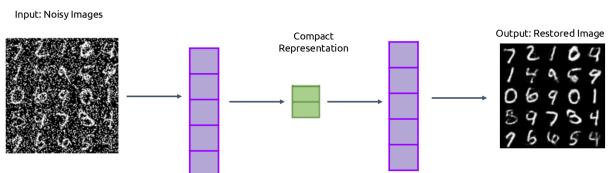
[Check the Demo](#)

## Deep Learning Problems, Models & Research

### Computer Graphics and Vision

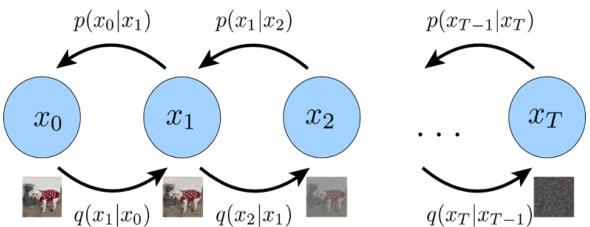
#### Denoising

**Autoencoder** networks can be trained to denoise images. The training is easy, as you just add the noise to original images.



#### Diffusion Models

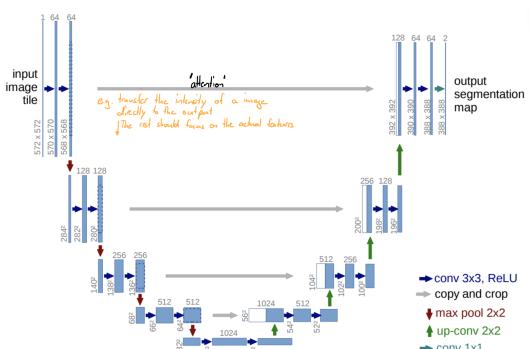
A *Diffusion Model* is a denoising autoencoder with fancy math, it's the inverse of adding noise.



#### Examples

#### Attention

With attention we can transfer information from the input to the output. We pass a weighted sum of encoder states to the decoder.



The attention can also be visualised

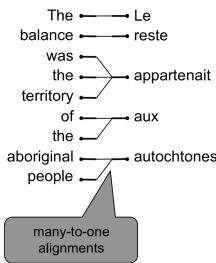
A dog is standing on a hardwood floor.A stop sign is on a road with a mountain in the background.A group of people sitting on a boat in the water.A giraffe standing in a forest with trees in the background.

## Natural Language

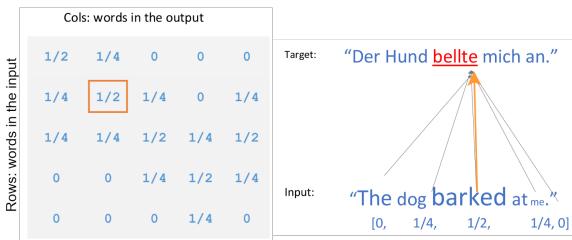
### Attention

With attention we can transfer information from the input to the output. We pass a weighted sum of encoder states to the decoder.

In natural language this can mean, **different words in the output pay attention to different words in the output**



The “attention” is represented as a weights matrix. It states \*how much attention does output word  $j$  pay to input word  $i$ . Each column shows the importance of the input words to one output words. Each Column **sums to one**.



The score of each word (one column) is the **attention alignment score function**

$$\alpha_{t,i} = \text{align}(y_t, x_i) = \frac{\exp(score(s_{t-1}, h_i))}{\sum_{i'=1}^n \exp(score(s_{t-1}, h_{i'}))}$$

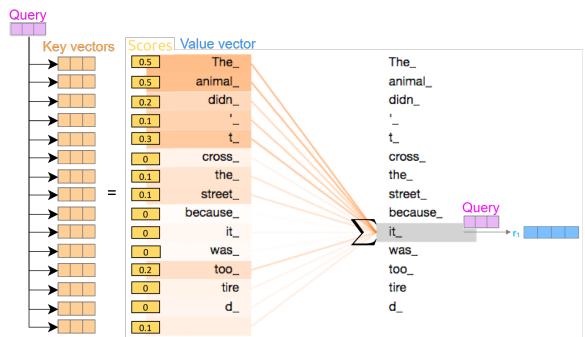
which is the *softmax of some predefined alignment score*. And this shows how well the two words  $y_t$  and  $x_i$  are aligned. For the  $\text{score}(\dots)$  function there are several options

Name	Alignment score function	Citation
Content-base attention	$\text{score}(s_t, h_i) = \text{cosine}[s_t, h_i]$	Graves2014
Additive(*)	$\text{score}(s_t, h_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[s_t; h_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a s_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(s_t, h_i) = s_t^\top \mathbf{W}_a h_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(s_t, h_i) = s_t^\top h_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(s_t, h_i) = \frac{s_t^\top h_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where $n$ is the dimension of the source hidden state.	Vaswani2017

Check [here](#) for more

### Self-Attention

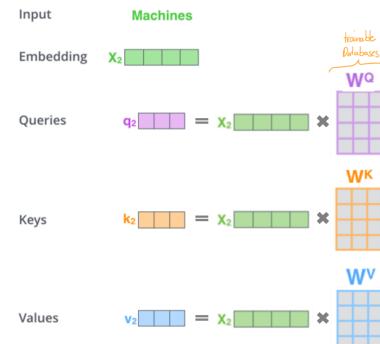
Self-attention computes the output vector  $r_i$  for each word via a weighted average of vectors extracted from each word in the input sentence. Here, self-attention learns that “it” should pay attention to “The animal”.



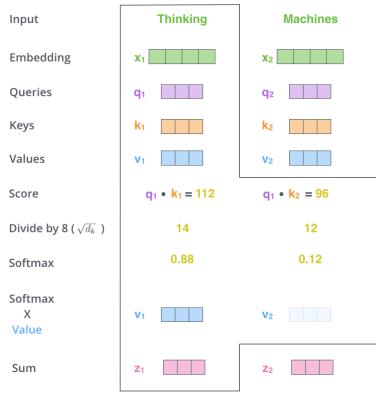
1. We compute a **query vector** for the word and compare it to a **key vector** for every other word. This computes a **score**.
2. For the output vector  $r_1$  we sum all **value vectors**, weighted by the **score** in step 1.

- Attention significantly **improves NMT performance**
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention **solves the bottleneck problem**
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention **helps with vanishing gradient problem**
  - Provides shortcut to faraway states
- Attention provides **some interpretability**
  - By inspecting attenti

For the **exampleMachines** we get the 3 vectors:

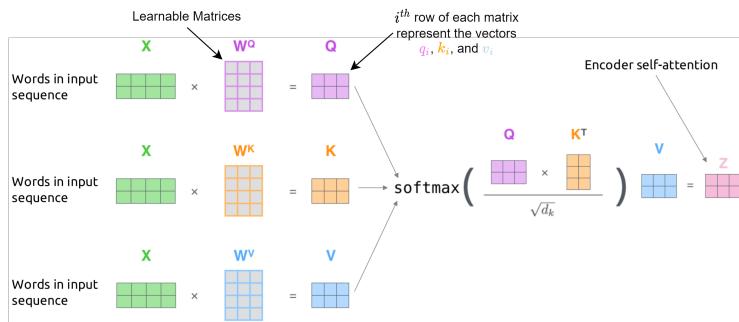


To get the **self-attention for Thinking**, we do:



- Score:** Dot product of the query vector  $q_1$  of "Thinking" with the key vectors  $\{k_1, k_2\}$  of each word.
- Scale:** Divide each score by  $\sqrt{\text{dimensionality } d_k}$  for more stable gradients.
- Softmax:** Apply softmax to transform scores into attention weights
- Weighting:** Multiply with the value vectors  $\{v_1, v_2\}$
- Sum:** Sum the weighted value vectors into the final **self-attention for "Thinking"**

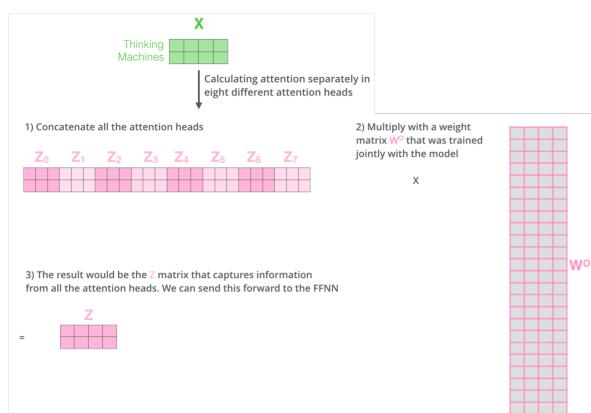
## Computation



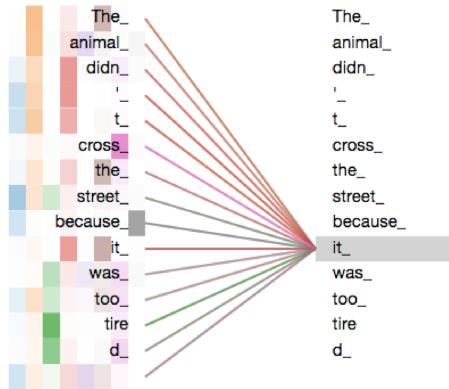
## Multi-Head Attention

Multi-head Attention is used to improve the performance of regular self-attention. We compute self-attention as before some number of times. Call these "attention heads". The size of the attention heads are smaller than when just using regular self-attention.

To get one set of attention vectors, we concatenate all the heads and apply a linear layer in order to get  $Z$ .

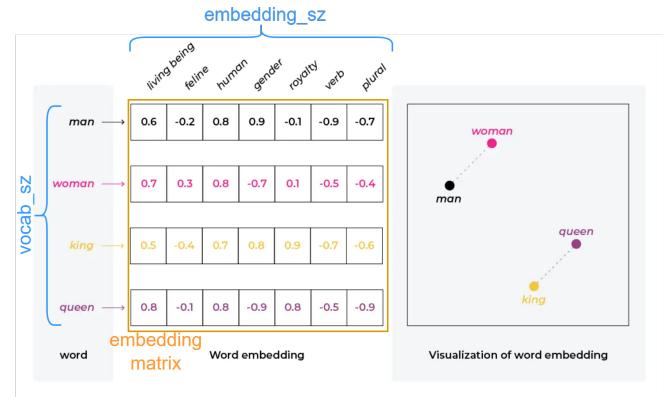


Multiple heads allow for each head to learn different relationships between words in the sentence. The network ( $W^O$ ) then learns how to handle this. These multiple attentions can be [visualised with this tool](#)

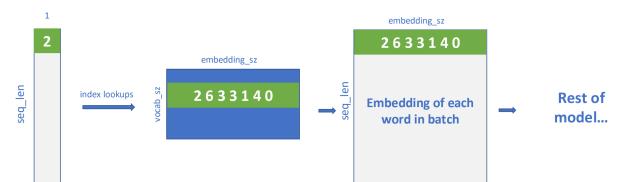


## Word Vectors

Words can be stored as **vectors**, the size of the vector is set by the **embedding sz** and is a hyperparameter. The **embedding matrix** is learnable and initialised randomly. Each embedding *can* have a meaning, but doesn't have to (trained with the network).



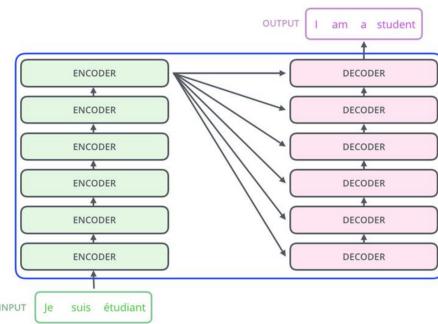
The embedding matrix is then used as a lookup table for all the words. (Example, first word in input sentence with ID=2)



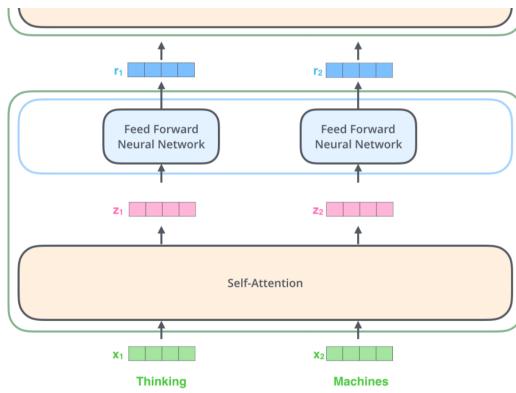
## Transformer

Transformers are based on Encoders and Decoders. The big difference to Autoencoders is, that they, are outputting a different language.

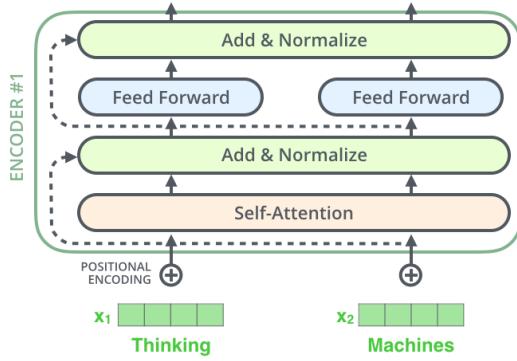
The multiple stack up of Encoders and Decoders is for **better performance**.



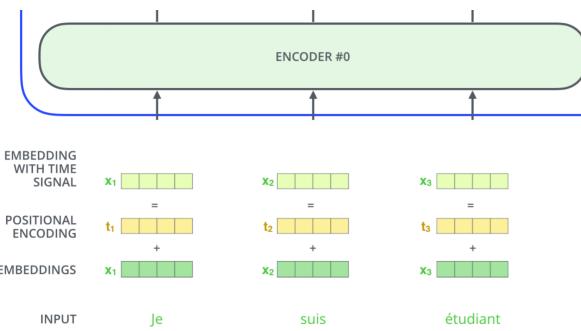
**Encoder Block Map** If we input a word sequence  $\{x_1, x_2\}$ , first the **self attention** is applied to get the value vectors  $\{z_1, z_2\}$ . Then a Feed Forward Network is applied to each word individually.



For performance improvements we add **Residuals** to negate vanishing gradients. Furthermore, we apply **LayerNorm**, which improves convergence time

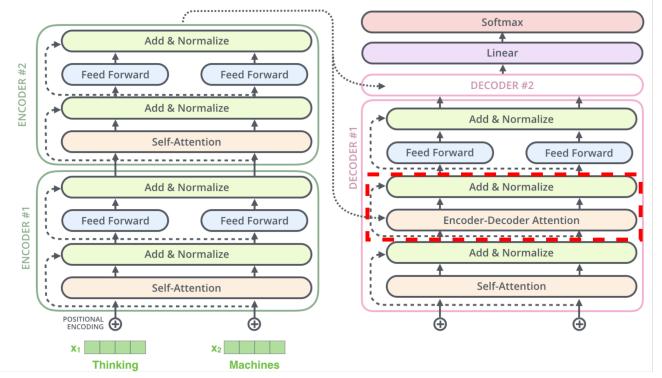


**Positional Encoding** Instead of passing a Embedding vector to the encoder, we pass a **Embedding with Time Signal** vector. Positional encoding can be learned or defined by a fixed function.

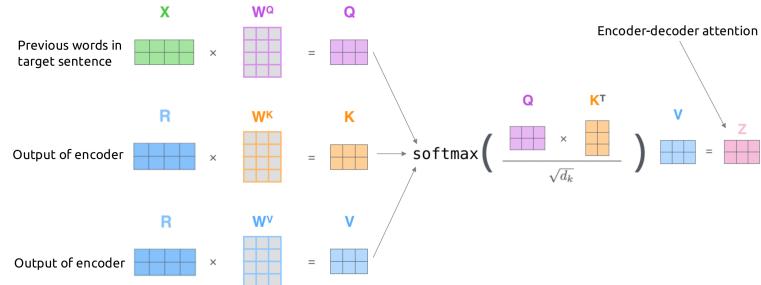


**Decoder Block Map** Following the encoder blocks, there are **Decoder Blocks**, these have are very similar to the encoders, but

with a additional layer of a **Encoder-Decoder Attention**.



The **Encoder-Decoder Attention** takes the output of the encoder instead of the input words



**Audio and Video Synthesis** .....

**Search using Deep Reinforcement Learning** .....

**Anomaly Detection** .....

**Irregular Networks** .....