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)
```

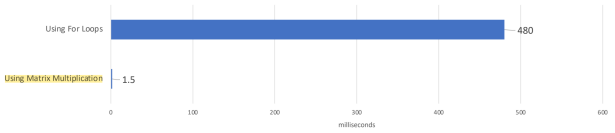
Use **Boradcasting** to work with arrays of different sizes. In Hardware data is take from the same memory space multiple times.

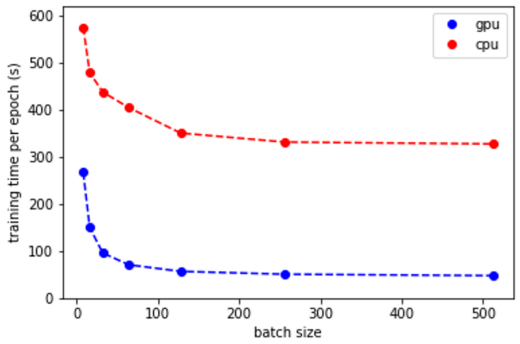
```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11,
↪  12]])
v = np.array([1, 0, 2])
y = x + v.T  # Add v to each row of x using
↪  broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5 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

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

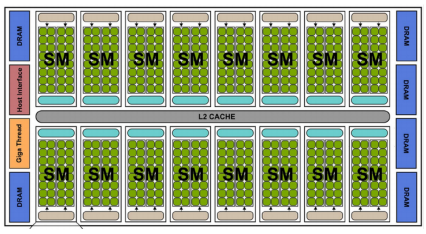
```
C = np.dot(A,B)
F = np.matmul(D,E)
```

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.*

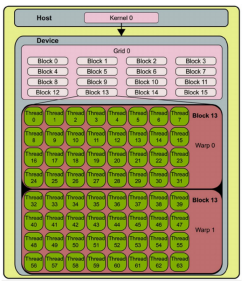




Architecture of a CUDA-capable GPU



CUDA compute model



Morals of AI

Gender Shades

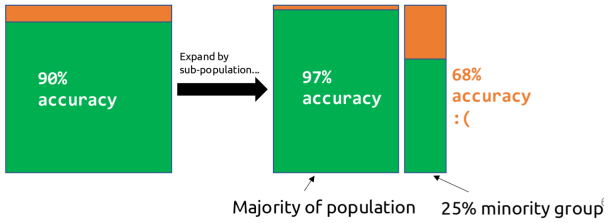
Gender detecting algorithm is trained on a highly biased dataset, which leads to different results, depending on the target. Read More

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%

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

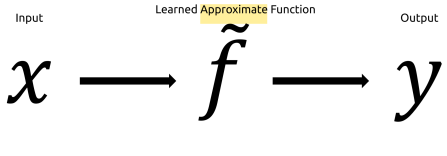


Reknognition

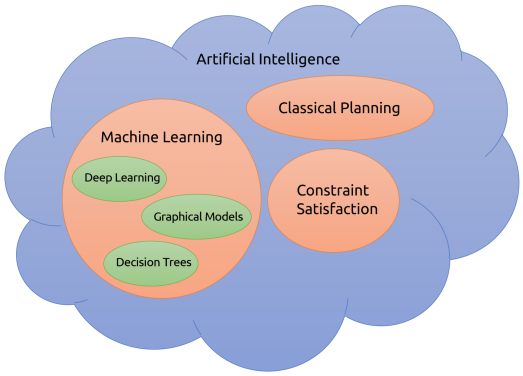
Facial Recognition algorithm wrongly matches government members to mugshots from a criminal database. The algorithm had **5% false positives** which isn't to bad, but when deployed states big issues of wrongly accusing innocent people. Read More

Machine Learning Concepts

Machine Learning == Function Approximation

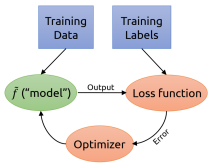


...so our goal is to ***learn*** approximations of these functions ***from data***

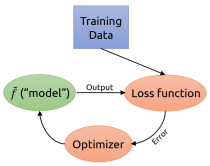


Types of Learning

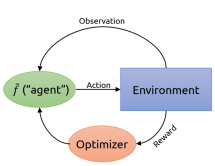
Supervised Learning

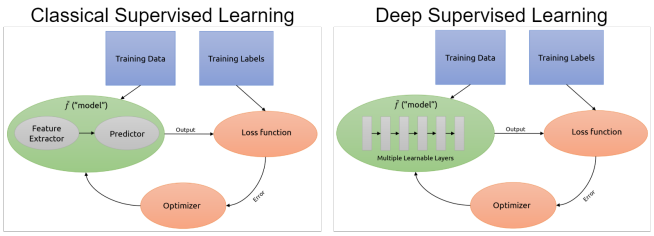


Self-Supervised Learning



Reinforcement Learning





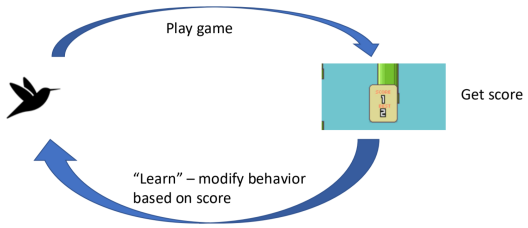
Self-Supervised Learning

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)
- ...

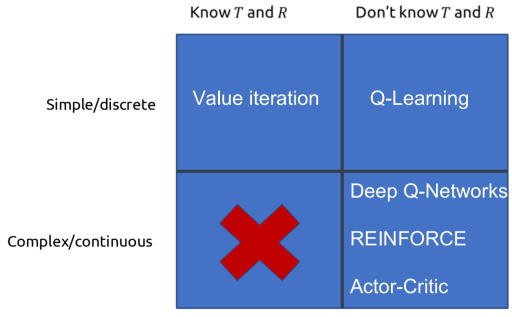
Reinforcement Learning

Requires no pre-collected data, it learns from experience by playing a game over and over again. **Agent** wants to choose actions that will give it higher scores



RL Reservations

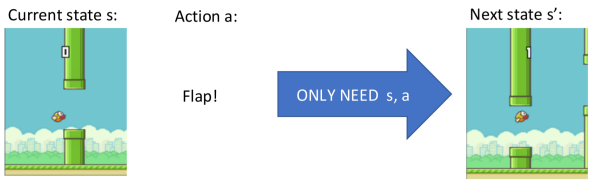
- Inefficient with data
  - AlphaGo learned from playing 100mio times
  - Human Go champion only played 50k times
- Sensitive to small perturbations in the environment



Markov Decision Process (MDP)

- States S - set of possible situations in the world
- Actions A - set of different actions an agent can take
- Transistion function $T(s, a, s')$ - returns **possibility** of transitioning to state s' after taking action a in state s
  - $T(s, a, s') = P(s_{t+1} = s'|_{s_t=s, a_t=a})$
- Reward function $R(s, a, s')$ - returns reward received for transition to state s' after action a on state s is taken

In the *Markov Decision Process* history doesn't matter, only the current state.



Goal: maximise sum of discounted future rewards G_t, "return"

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

γ is the discount factor - value soon rewards not later ones. **Rewards exponentially decay the later they are received**.

Policies Policy Functions say what action should a agent take in a given state

$$\pi : S \to A$$

Output: $\pi(s) = a$ means in state s, take action a.

The goal of RL is to learn an optimal policy π, that maximises future cumulative reward

Value Iteration (V-Function)

- Function that returns the "value" of each state
- Value of state s under policy π is the expected return when starting in s and following π
- "Value of my current state is the total discounted future reward I expect from following a policy π from now on"
- $V_\pi: S \to \mathbb{R}$
- $V_\pi(s) = E[G_t \mid S_t = s]$ for all $s \in S$, where G_t is the return
- $= E[R(s, a, s') + \gamma V_\pi(s') \mid S_t = s]$
- $V_\pi(s) = \sum_{s' \in S} P(s'|s, a)\ [R(s, a, s') + \gamma V_\pi(s')]$

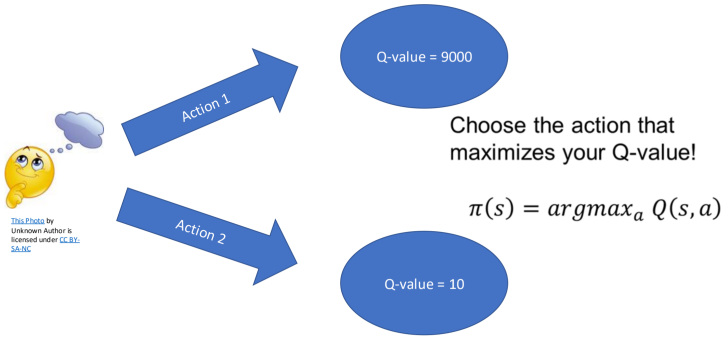
Expectation across transition function - deals with the potential stochasticity of transition function

NOTE: recursively defined! Literally "reward agent receives now + value of the next state"

- **Discrete state spaces** can be represented by a lookup table
- **Continuous or very large discrete state spaces** will need to be represented by a neural network

Q-Function

- $q_\pi: S \times A \to \mathbb{R}$
- $q_\pi(s, a) = E[G_t \mid S_t = s, A_t = a]$ for all $s \in S, a \in A$
- AKA "action-value function"
- Outputs expected return from taking action a in state s and following policy π thereafter



Q-Function in terms of V-Function

$$Q^\pi(s,a) = E[R(s,a,s') + \gamma V^\pi(s')]$$

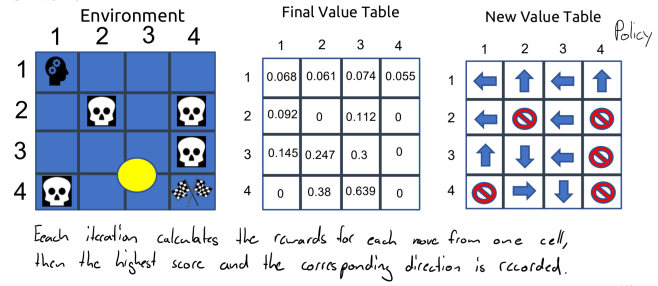
$$Q^\pi(s,a) = \sum_{s' \in S} P(s'|_{s,a})[R(s,a,s') + \gamma V^\pi(s')]$$

Learn V* and Q*

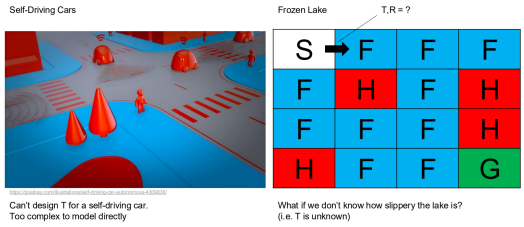
1. For all s, set $V(s) := 0$.
2. Repeat until convergence:
   1. For all s:
      1. For all a, set $Q(s,a) := \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V(s')]$
      2. $V(s) := max_a\, Q(s,a)$
3. Return Q

Example:

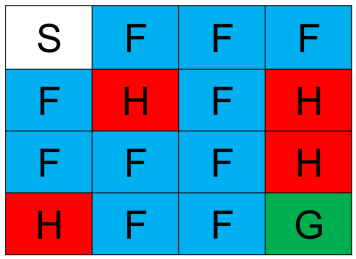
T is known as 1/3 in forward/left/right when stepping forward.



Deep-Q Learning If the Transition T or Reward R are not known



- Every time we take an action, use the observed reward to update our estimate of Q
- $V(s)$ is still $\max_a Q(s,a)$, so it only matters how we update Q
- Importantly, we use our estimates of Q at each step to pick what we think is the best action, instead of just moving randomly
  - action = $\text{argmax}_a Q(s,a)$



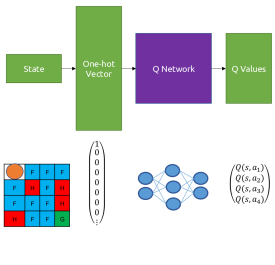
$Q^\pi(s,a) = E[R(s,a,s') + \gamma V^\pi(s')]$
$Q^\pi(s,a) = \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^\pi(s')]$

The limits of tabular learning

The above approach still uses a look up table to determine what action on what field to take. To work with big (continuous environments), we use neural networks as an approximator

Frozen Lake Example

- Feed a one-hot representation of the state into a neural network to approximate the Q-value for each action
  - One-hot vector of length 16 for frozen lake
- Can also pass in a more complex state to the Q-network rather than a one hot representation.
  - Useful when the number of states is very large, making a one hot representation impractical
  - Or when states are continuous



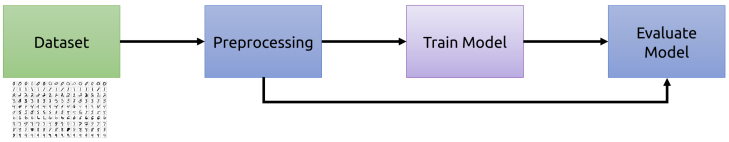

```
# Weights for Q-Network
Q =
↪  tf.Variable(tf.random.uniform([16,4],0,0.01))
↪
# Q-value function
def qVals(inptSt):
    oneH = tf.one_hot(inptSt,16)
    qVals = tf.matmul([oneH],Q)
    return qVals
# argmax over q-values to get estimated best
↪  action
action = tf.argmax(qVals(st),1)
```

For training such a network, we replace the Q-values LUT with the Network

```
Q_values = qVals(st)
if np.random.rand(1) < epsilon:
    act = env.action_space.sample()
else:
    act = tf.argmax(Q_values)
nst, rwd, done, _ = env.step(act)
nextQ = qVals(nst)
loss = tf.reduce_sum(tf.square(Q_values -
↪  rwd+gamma*nextQ))
↪  optimizer.apply_gradients(tape.gradient(loss,Q),Q)
```

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:

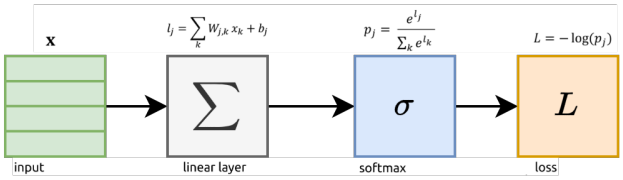
$$w_i = w_i + \lambda \left(a^k - f(\mathbf{x}^k)\right) x_i^k$$

   - 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**.

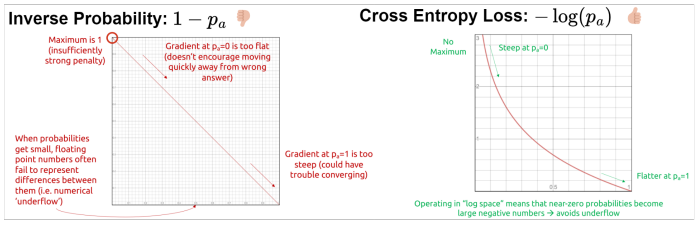


To get a probability for **binary classification**, we introduce a **probability layer**. One of the possible function is **Softmax**

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

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**

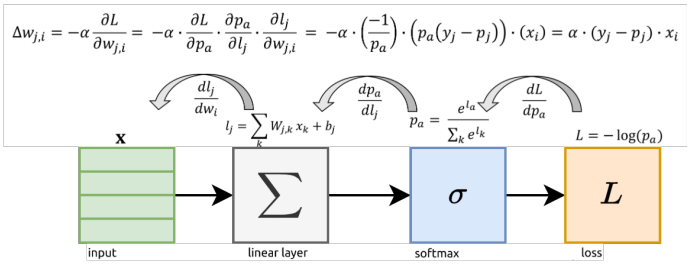


Gradient Descent

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

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
```

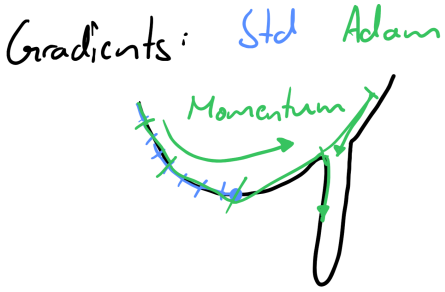
Stochastic Gradient Descent (SGD)

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*

> **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

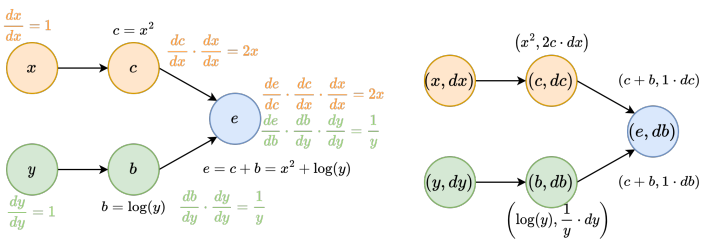
Symbolic differentiation

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

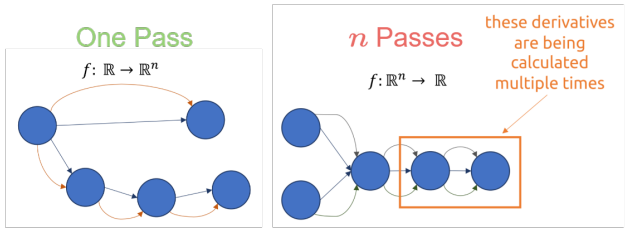
Automatic differentiation

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

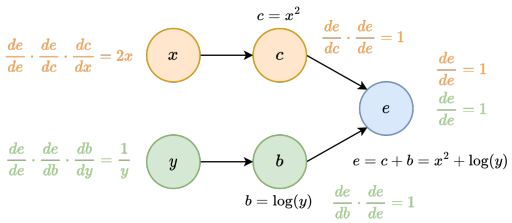
Forward Mode Autodiff Every node stores its `(value, derivative)` in a tuple, called **dual numbers**. To compute the overall derivative, each derivative can be chained up. This is implemented via **Overloading**, every function / operator has multiple definitions based on the types of the arguments.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

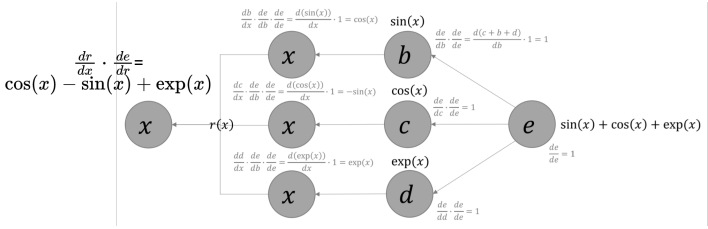
> **Issue w/ forward mode**
>
> 
>

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

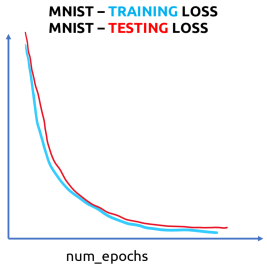
Fan-Outs (Reverse) The way to handle fan-out is to **add** the derivatives of the fanned-out nodes through replication $r(x)$.

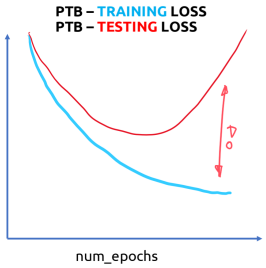


Diagnosis Problems

Overfitting

Training on a complex dataset can lead to **overfitting** (PTBis 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

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

For both, this is a term added to the existing loss function.
λ controls the strength of the penalty

Regularization can be applied to certain layers on Keras through `tf.keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(lambda), activation='relu')`.

Deep Learning Concepts

Common Misconception

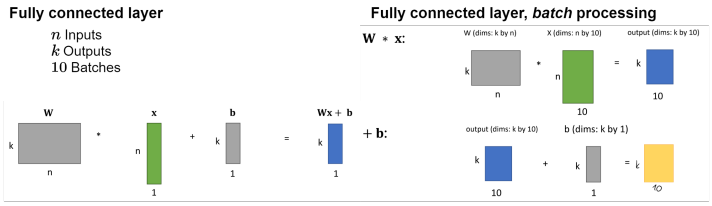
Deep Learning != AI, Just because deep learning algorithms are used doesn't mean there is any intelligence involved.
Deep Learning != Brain, Modern deep nets don't depend solely on *biologically mimiced neural nets* any more. A fully connected layer represents such a neural net the closest.
Deep Learning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

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

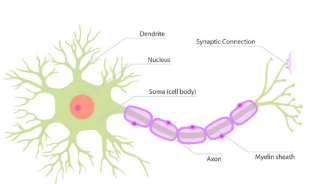
Vectorized Operations

For efficient operation, use **Matrices**.

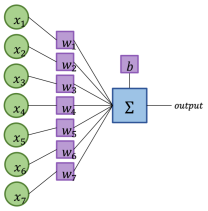


Neural Networks

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

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

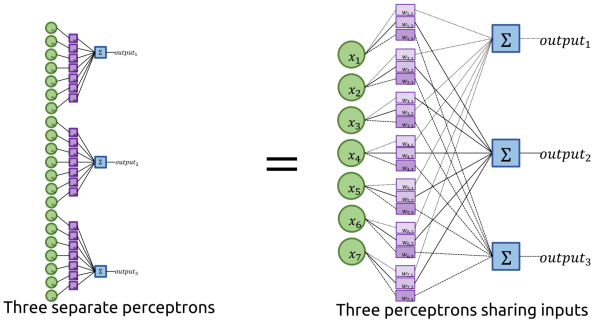
Parameters

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



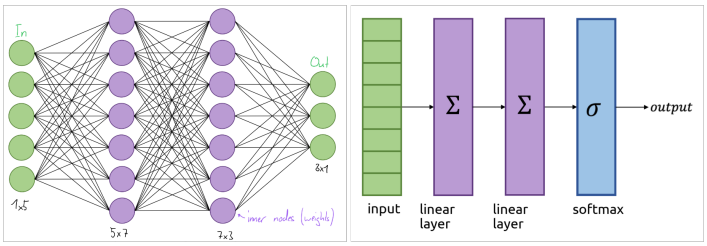
Three separate perceptrons = Three perceptrons sharing inputs

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

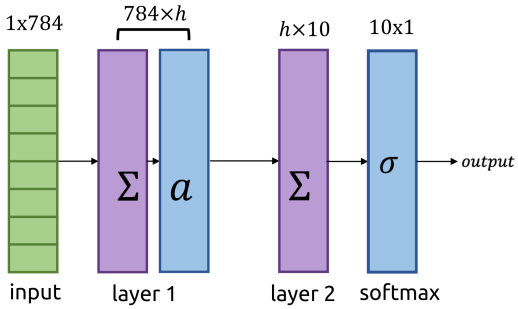
$$\Delta w_i = \begin{cases} 0, & \text{for } a^k = \hat{y} \\ -x_i^k, & \text{for } \hat{y} = 1, a^k = 0 \\ x_i^k, & \text{for } \hat{y} = 0, a^k = 1 \end{cases}$$

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.

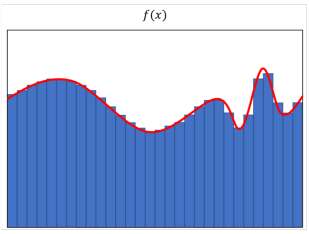


Universal Approximation Theorem

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

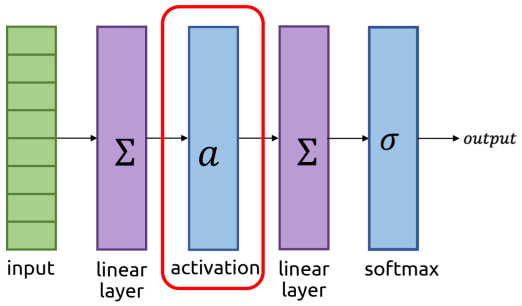
When simplifying the linear equation we get

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

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

Activation Functions

We introduce a **nonlinear** layer



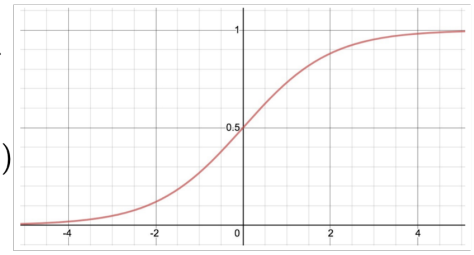
A activation function binds network outputs to a particular range. In the last layer this can be used to restrict the range, for example *age is strictly positive*.

Futher PyTorch activation functions can be found here.

Sigmoid .

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

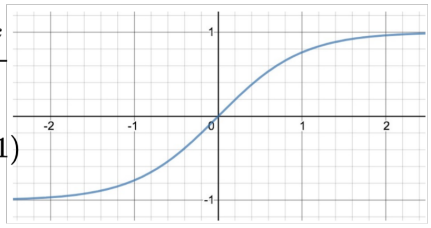
$$\sigma(x) : \mathbb{R} \to (0,1)$$



Tanh .

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

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



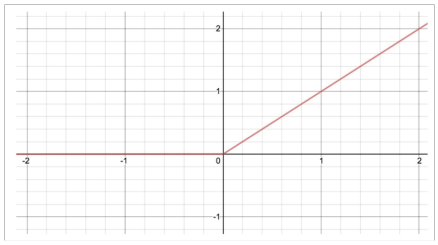
Vanishing Gradient

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

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

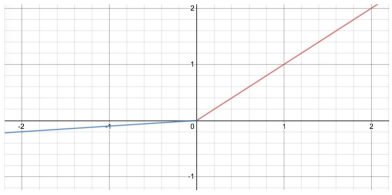


Dead ReLU

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**.

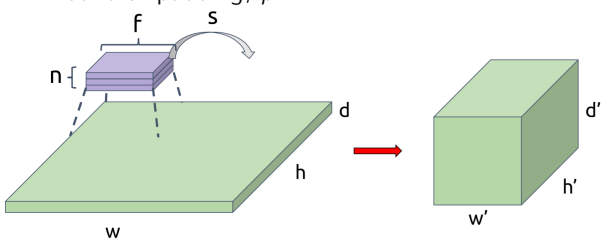
$$V(x,y) = (I \otimes K)(x,y) = \sum_m \sum_n I(x+m, y+n) K(m,n)$$

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

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

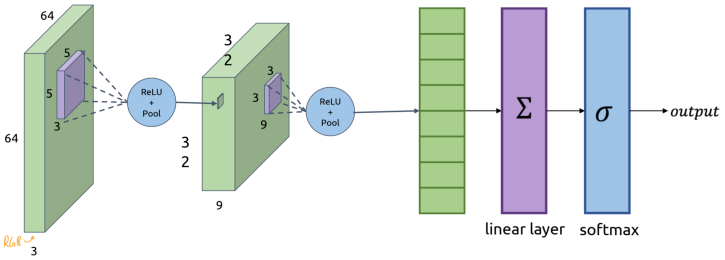
For **VALID** padding $p = 0$, for **SAME** padding p is chosen so output is same

```
# 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")
```

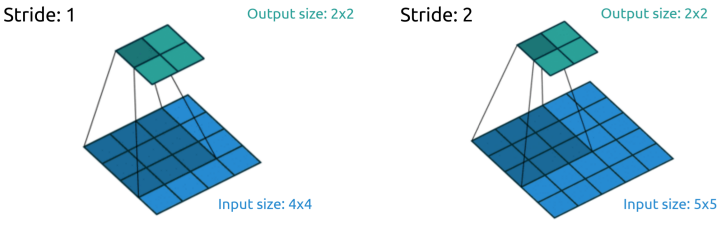
The inputs to `tf.nn.conv2d(...)` are:

- `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'`

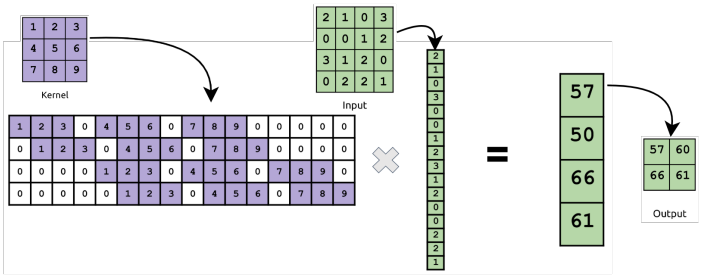
Tipically there are several convolutional layers and then a fully connected layer. This can be achieved through flattening a layer `flat = tf.reshape(conv, [conv.shape[0],-1])`.



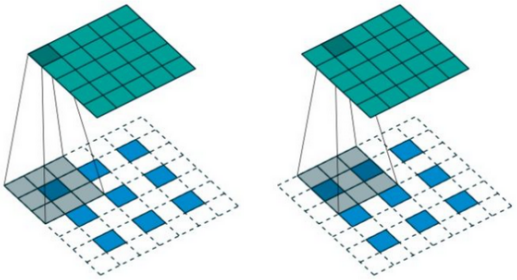
Stride The distance we slide a filter on each iteration is called **stride**. With a bigger stride, you compress a same size input into a smaller output. This decreases the image resolution controlled, **Downsampling**. The filters are **Kernels** and are made of **learnable parameters**.



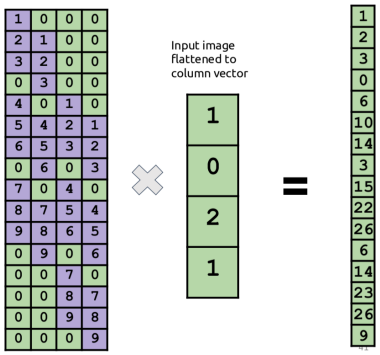
Computation The image is flattened and the kernel is unrolled into a bigger matrix. This leads to a normal **matrix/vector multiplication**



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

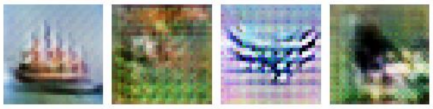


Computation Same as with the convolution, we flatten all matrices into vectors/matrices but now we **transpose** the kernel matrix, which gives us the **de-convolution**:



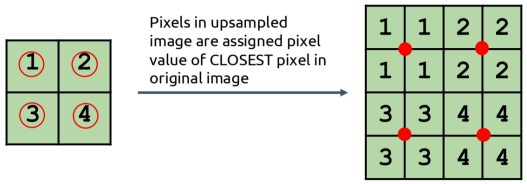
Checkerboard Artifacts

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).

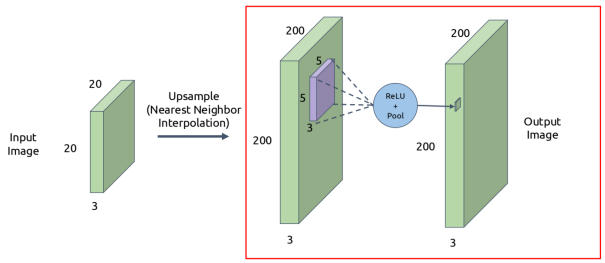


Prevention:

1. Upsampling using nearest neighbour interpolation

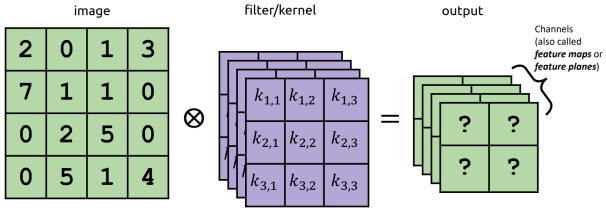


2. Perform a convolution with `'SAME'` padding on upsampled image

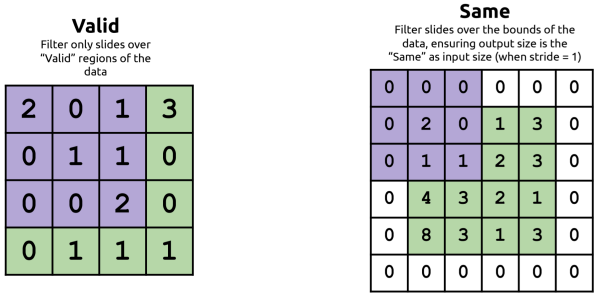



```
# Layer to upsample the image by a factor of 5
↪  in x and y using nearest
# neighbor interpolation
tf.keras.layers.UpSampling2D(size=(5, 5),
↪  interpolation='nearest')
# Do a convolutional layer on the result
tf.keras.layers.Conv2D(filters = 1, kernel_size
↪  = (10,10), padding = "SAME")
```

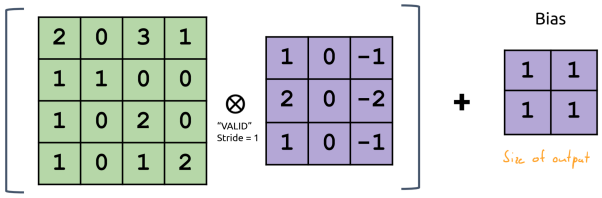
Filter Banks Futhermore, use several kernels per image, this block of kernels is a **filter bank**. The output is then a **mutli-channel** image. Multiple filters are able to extract *different features* of the image.



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

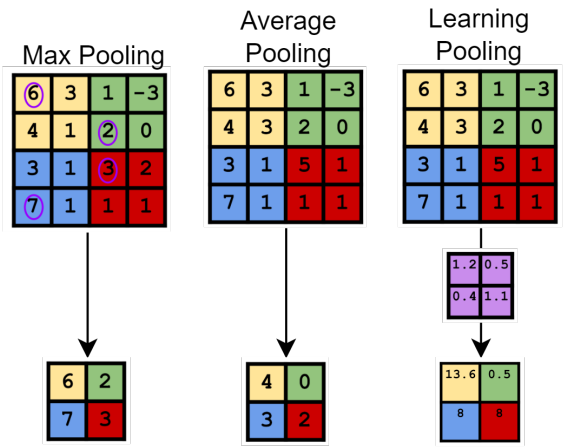


This can be done through `tf.nn.bias_add(value, bias)`. When using keras layers, a bias is included by de-

fault `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.



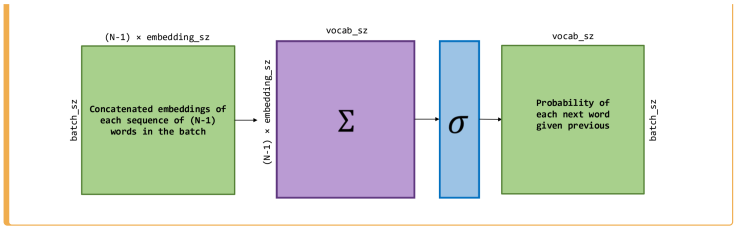
Invariances

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

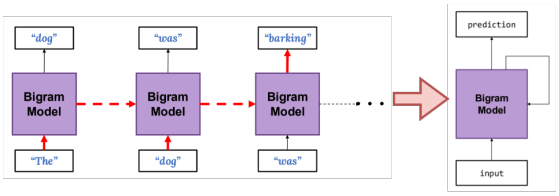
There are further invariances, which can hurt a CNNs performance. CNNs don't do to good on these, for good performance, lots of training is needed.

Recurrent Networks

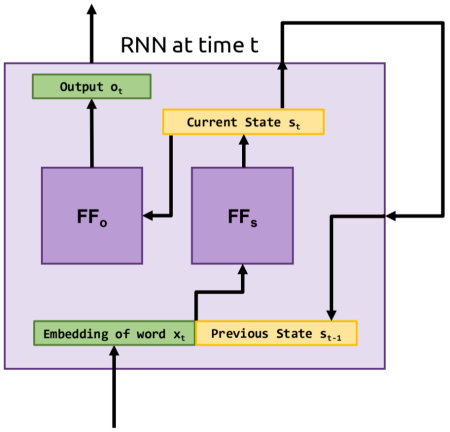
> ⚠ Problems of N-gram Model
>
> The issue with N-gram models is, that they're **not flexible**, each additional word means more weights to train. That leads to a fixed input size.



To takle this we introduce **Recurrent Networks (RNN)**, based on Bigram models in a recurrent (not recursive) connection.



A **RNN** cell consists of two Feed Forward (FF) layers, one to compute the next state and one to compute the output.



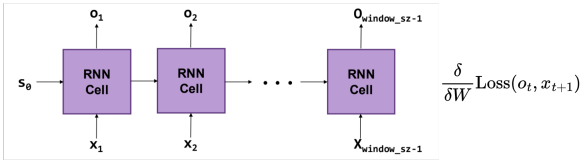
$$s_t = \rho((e_t, s_{t-1})W_r + b_r)$$

$$o_t = \sigma(s_t W_o + b_o)$$

Note: s_0 is typically a zero vector.

Training RNNs

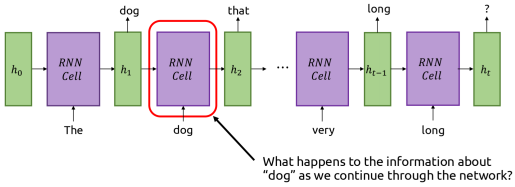
Because the gradients for o_t depend on x_t and all previous inputs, we **backpropagate through time**. Because we reuse the same block over and over again, the gradient accumulates over time. The size of these iterations can be used as a hyperparameter window_{sz} .



Long Short Term Memory (LSTM)

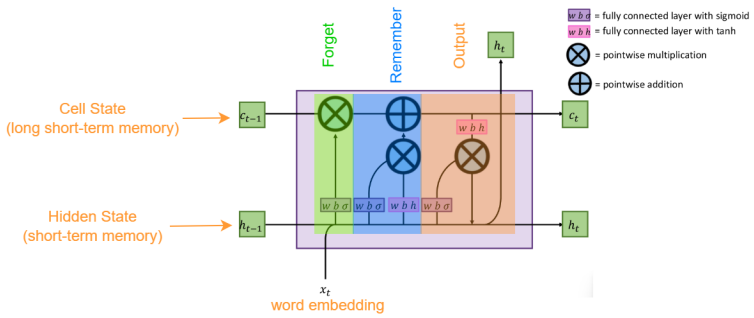
Short Term Memory of RNNs

When there is a long sequence of words, we must somehow make shure, that information from the beginning isn't lost through iterating over several blocks.



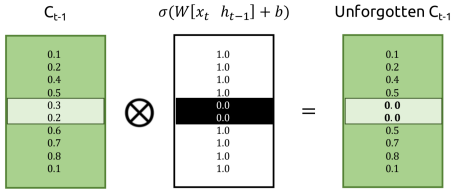
The information will slowly fade away over the recurrent convolutions.

To handle this, we have the **LSTM**-Cell, consisting of three modules

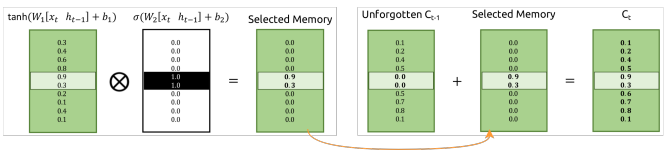


Forget Module Filters what gets allowed into the LSTM-cell from the last state *(e.g. gender pronouns are coming in c_{t-1}, a new subject is seen x_t, → forget old pronouns).*

Forgetting is handled by a point wise multiplication with a mask, it zeros out part of the cell state.



Remember Module We can save information into the previously emptied slots in the cell state. First a point wise multiplication is used to decide what to remember, then this selective memory to the cell state



Solution: Cell state never goes through a fully connected layer → No mix-up of information.

Output Module Provides path for short-term memory h_t to temporarily acquire from the long-term cell state.

Sequential Networks

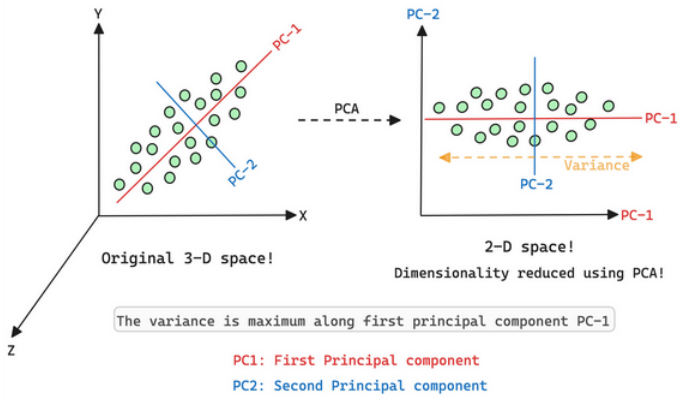
Latent Space

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**.

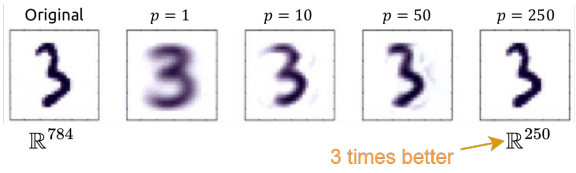
Principal Component Analysis (PCA)



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**.

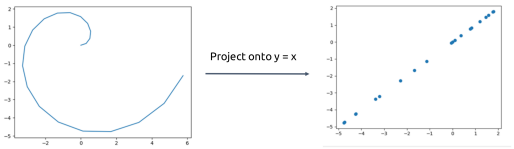
How: Find the direction of maximal variation, project onto this vector, repeat m times.

Example with MNIST:



Limitations of PCA

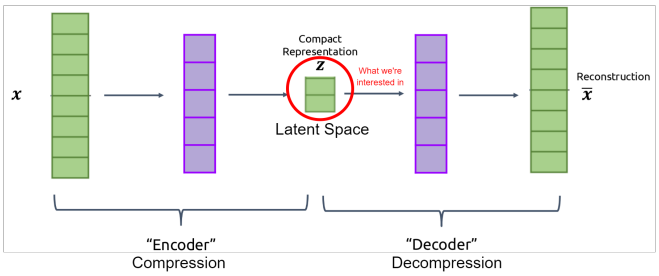
PCA can't figure out **non-linear** projections from $\mathbb{R}^2$ to $\mathbb{R}^1$.



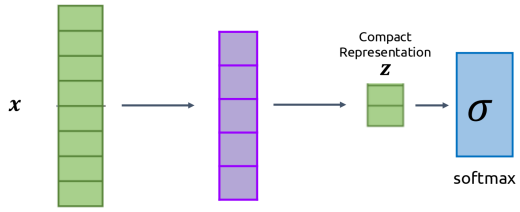
Butterfly-Network (Autoencoder)

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

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

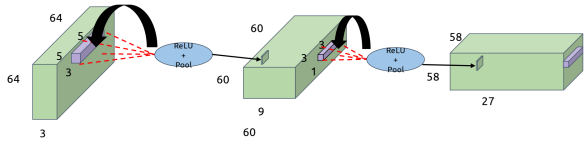


After training the network with the unlabelled data, the second part can be cut off and a binary layer be added. Now fine tune the last layer on **labelled data through supervised learning**.



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]

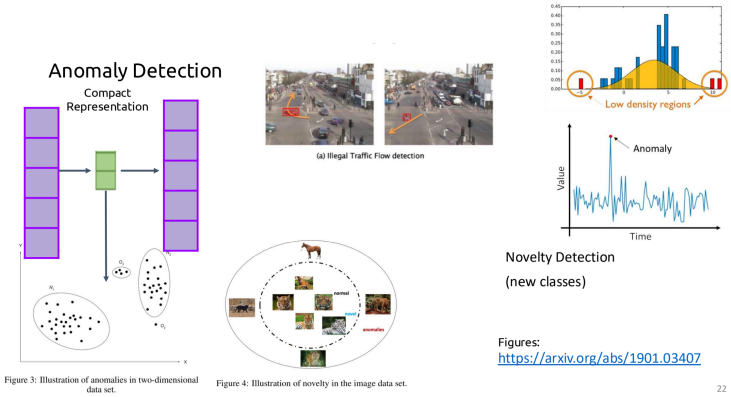
length 4 1D tensor representing the output shape.

Strides along each dimension (list of integers)

String representing type of padding

Autoencoder Applications

Anomaly / Novelty Detection .

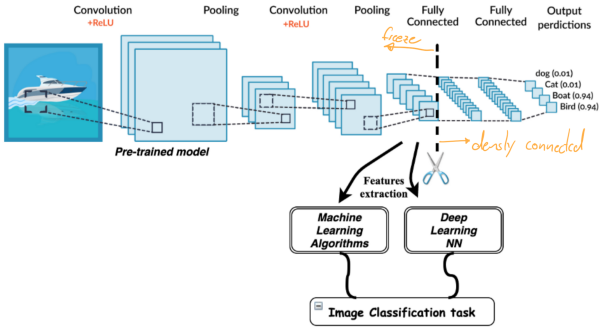


Others

- 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

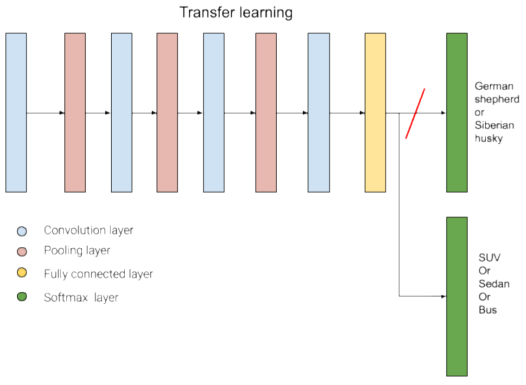
Taking a pre-trained network of a similar domain and adjust the last layers.



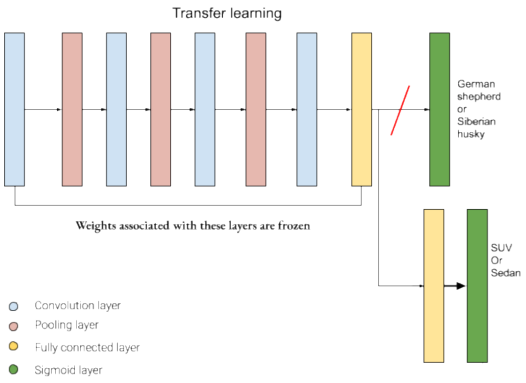
- Saves training time & money
- Network is more likely to generalise

Fine Tuning

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

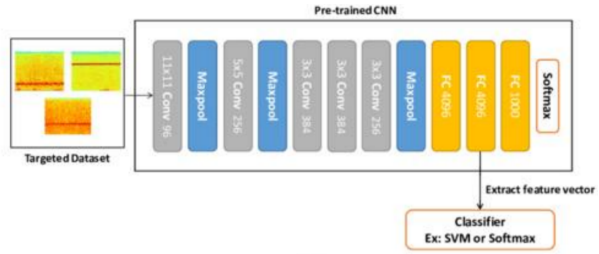


Or also switch add/replace a fully connected layer if the differences in the domain are bigger



Zero Training

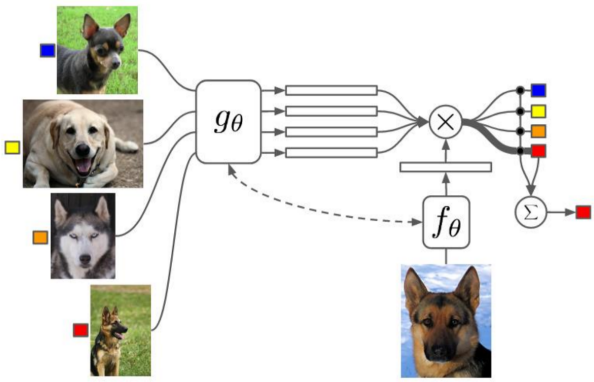
The last layer can be replaced by a *Logistic Regression* or *Nearest Neighbor Classifier*. The *transferred network* is then just a feature extraction pipeline. Use a database with known targets and classify new data through searching for nearest neighbour.



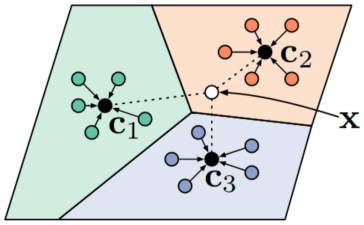
Few-Shot Learning

When there are only **few example of a sample** to be detected.

Siamese Networks where the sample and the test go through the same network. Then they are compared in the latent space.



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



Training Methods and Tricks

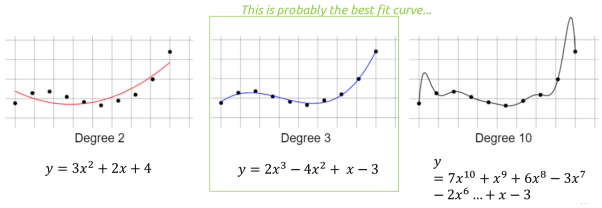
Early Stopping

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
```

Reduce Parameters

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



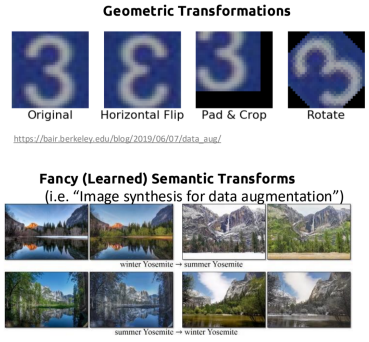
Reducing parameters can mean. . .

- . . . 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 Agumentation

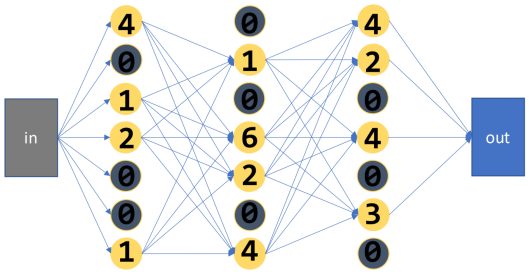
Generate **random variations** on your training data.



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.

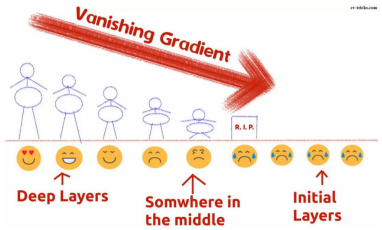


Dropout can be handled with Keras through `tf.keras.layers.dropout(rate)`, where `rate`is a *hyperparameter* between $[0, 1]$. `rate=0.5` is drop $\frac{1}{2}$, keep $\frac{1}{2}$. `rate=0.25` is drop $\frac{1}{4}$, keep $\frac{3}{4}$.

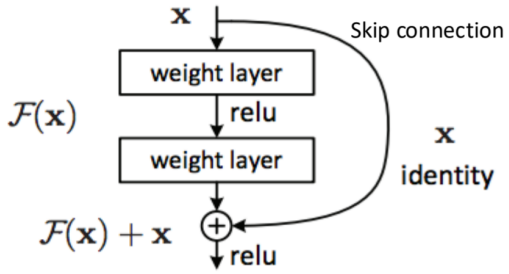
Skip Connections / Residual Blocks

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



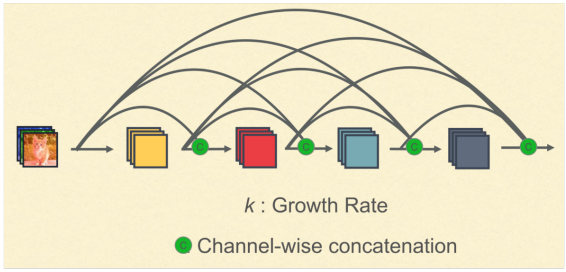
To mitigate the *vanishing gradient problem*, we use **Residual Blocks**



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.**

Dense Connectivity

To reduce redundancy, **dense connectivity** can be used. for this a network with fewer parameters has increased amounts of connections



Example:

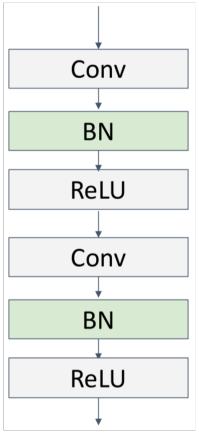
- ResNet: 18 layers, 11.7M parameters, 30% validation error
- DenseNet: 121 layers, 8M parameters, 25% validation error

Skipping connections and connecting densely is improving the loss surface (better trainability)

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.BatNormalization(input)`.

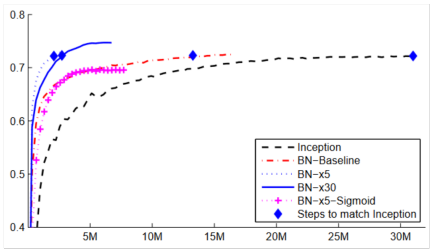


$$\mu_{\text{batch}} = \frac{\sum_{i=0}^{\text{batch}_{\text{size}}} x_i}{\text{batch}_{\text{size}}}$$

$$\sigma_{\text{batch}} = \sqrt{\frac{\sum_{i=0}^{\text{batch}_{\text{size}}} (x_i - \mu_{\text{batch}})^2}{\text{batch}_{\text{size}}}}$$

$$x' = \frac{x - \mu_{\text{batch}}}{\sigma_{\text{batch}}}$$

normalization



- 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 your run your program. Fast prediction. Export your trained weights for use in other applications

```
checkpoint = tf.train.Checkpoint( ... )
```
"trackable objects"

- 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

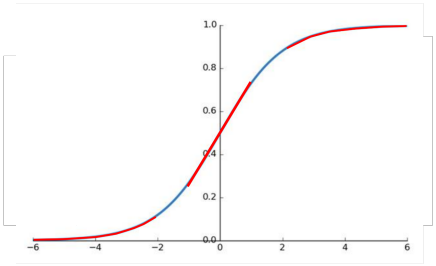
Restores checkpoint with counter = 3

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**.
Event 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: $y_i = Wi,1x_1 + \cdots + Wi,jx_j + \cdots + W_{i,n}x_n$
- 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).

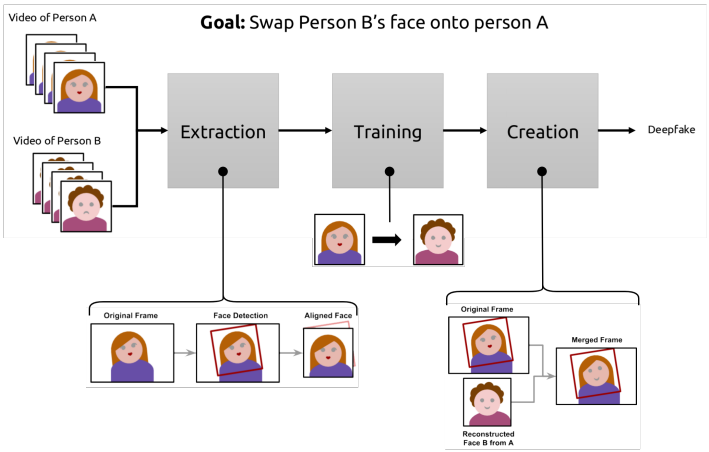
Check the Demo

Deep Learning Problems, Models & Research

Computer Graphics and Vision

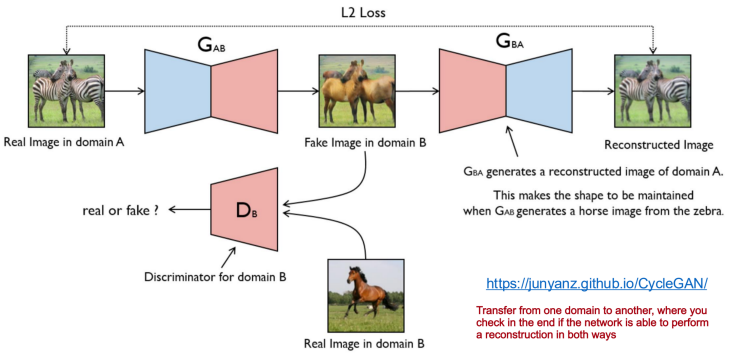
Deepfakes

1. Use face detector to identify the face in all frames → Crop out and rectify
2. Train NN to transform aligned images of person A's to person B's face with **same facial expressions**
3. Merge the person B face onto the original person A frame
   - Merging incorporates: colour correction, seam blending



CycleGAN To train a network to swap faces (**or any other domain**), we use a CycleGAN. This trains two Autoencoders and a Discriminator to:

1. Generate a fake image in Domain B
2. Check with discriminator if it looks "real"
3. Generate a reconstructed image from the fake image into Domain A
4. Calculate the loss on how different the reconstructed image is to the original



Deepfake detection There are several ways to detect deepfakes:

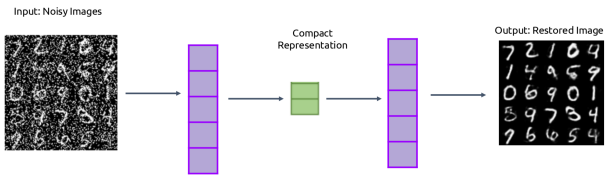
- Train CNNs on real videos + fakes to detect fakes
- Look for optical flow aberrations
- Look for heart rate aberrations
- Look for camera signatures

The issue is, that the deepfakes evolve with every new detection techniques (generator / discriminator!).

But there is a deeper issue, not all deepfakes are bad (parody, human aid, ...). On the other hand, misinformation and propaganda doesn't need deep fakes to be spread. Solutions(?): more human-in-the-loop, better-paid moderators, education, ...

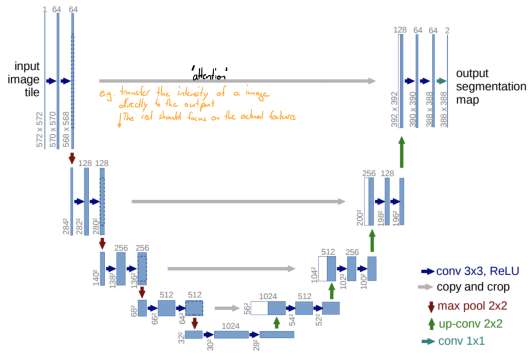
Denoising

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



Attention

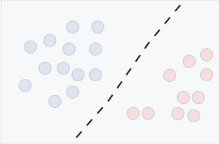
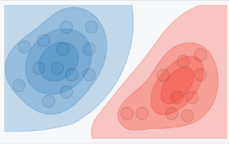
With attention we can transfer information from the input to the output. We pass a weighted sum of encoder states to the decoder.



The attention can also be visualised

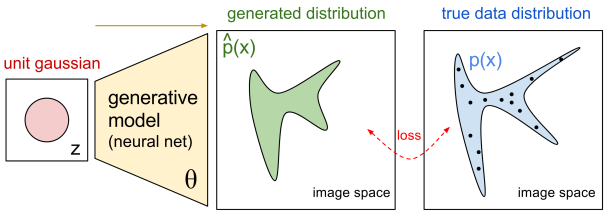
Generative- | Diffusion-Models

In comparison to discriminative models, the generative model doesn't have a clear decision boundary. Therefore a image which shows non-sense, isn't classified into a random class, but shown as very low likely hood on every class.

	Discriminative model	Generative model
Goal	Directly estimate $P(y\|x)$	Estimate $P(x\|y)$ to then deduce $P(y\|x)$
What's learned	Decision boundary	Probability distributions of the data
Illustration		
Examples	Regressions, SVMs	GDA, Naive Bayes

Generative Modeling

Generative models are trained to generate a data distribution $\hat{p}(x)$ from a noise input (uniform, gaussian, ...). The loss is calculated in reference to the actual data distribution of the ground truth dataset.



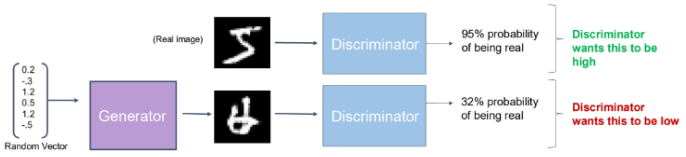
Use Cases:

- Text-to-Image Generation, $p(image|text_caption)$
- Super Resolution, $p(image|low_res)$
- Class-Conditional Generation, $p(image|class_label)$

Generative Adversarial Network (GAN)

A **GAN** consists of a **Generator** and a **Discriminator** which try to outwork each other. They are trained in the same loop.

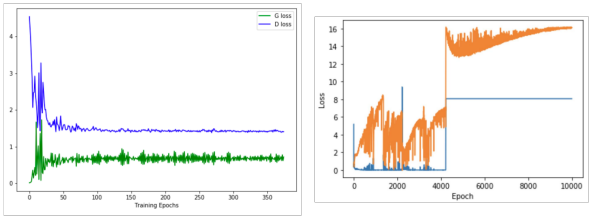
Training the Discriminator The Discriminator want's to distinguish the real from the fake samples as good as possible and is trained for that.



Training the Generator The Generator want's to generate images which are classified as real as possible. The Generator updates it's own weights to generate a more real looking image.

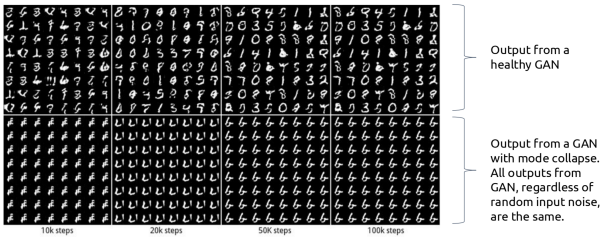
GAN training dynamics GAN training losses don't typically converge to zero because GANs involve a balance between the two competing networks generator and discriminator.

Attention: There can be issues with instability in the training of GANs.



Mode Collapse

A issue arises if the generator figures out a output that looks real, and continues to only show this as it's only output.



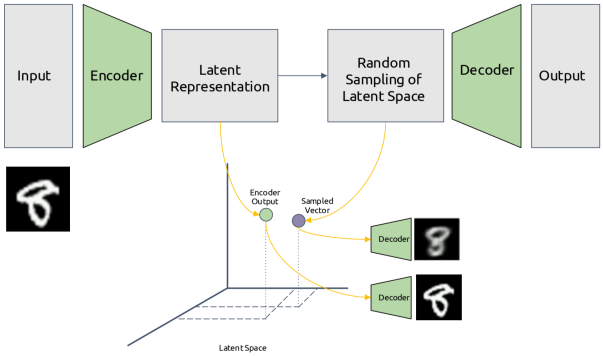
Evaluate GANs

Evaluating GANs can be hard, as we train the discriminator and the generator. A simple way to do it is to **Eyeball**, a human looking at the data and saying if it looks better or not. This isn't very scientific.

Variational Autoencoder (VAE)

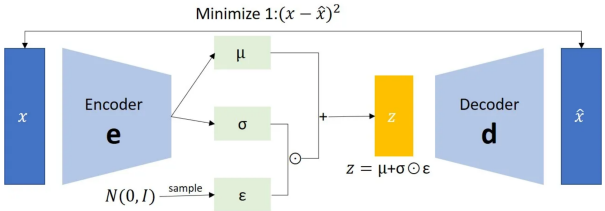
A **VAE** uses the a autoencoder as a basis, but the latent space gets sampled randomly before it's feed back into the decoder.

- The random sampling should be designed to produce random points in the latent space that are close to the output of the encoder
- Nearby points in the latent space should decode similar images



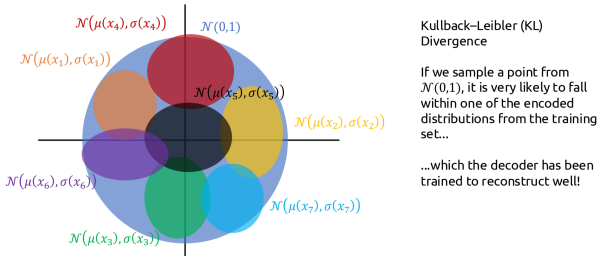
Because "random sampling" is non-differentiable, we introduce a point wise multiplication of the encoder output σ and a uniform distribution $\epsilon = N(0,1)$ (**seed**). This will be added to the output μ of the encoder, giving us the input for the decoder

$$z = \mu + \epsilon \cdot \sigma$$

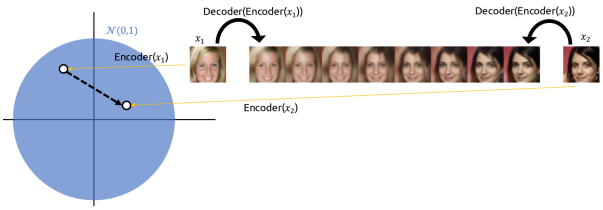


It can also be shown as

Latent Space Interpolation The training with the random sampling forces the latent space towards are Normal distribution.

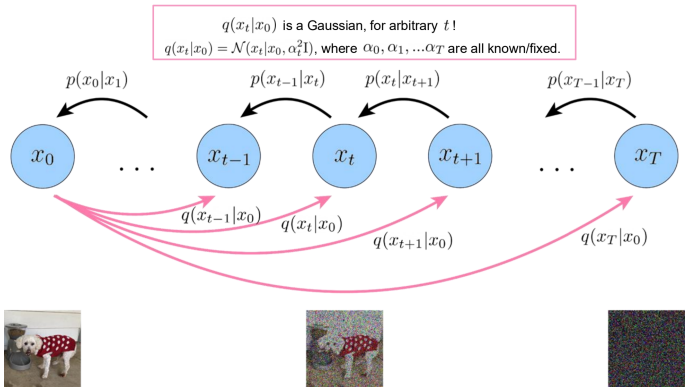


Because of that, nearly every point in the latent space has a valid output. This gives us the option of *latent space interpolation*



Hierarchical VAEs

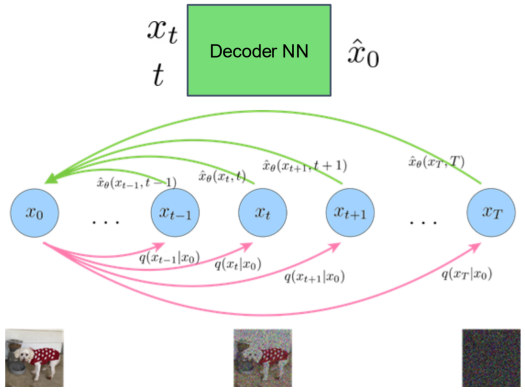
Stacking several VAEs into each other, leads to a deeper encoding of the latent space. This in the end leads to the the network being able to generate images out of noise.



Diffusion Models

A diffusion model is a hierarchical VAE with the following assumptions:

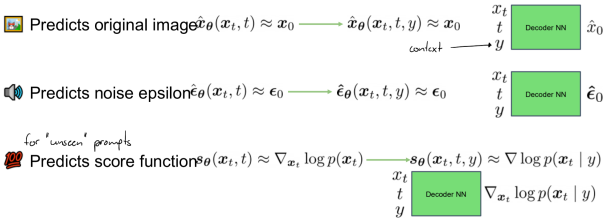
- All dimensions are the same.
- All encoder transitions are known Gaussians centred around their previous input.



A diffusion model predicts a clean image from a noisy version of the image and is trained like this

Algorithm 1 Training	**Algorithm 2** Sampling
1: **repeat**	1: $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$	2: **for** $t = T, ..., 1$:
3: $t \sim \texttt{Uniform}(1, ..., T)$	3: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\boldsymbol{\epsilon} = 0$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$	4: $\boldsymbol{x}_{t-1} = \hat{\boldsymbol{x}}_\theta(\boldsymbol{x}_t, t) + \alpha_{t-1}\boldsymbol{\epsilon}$
5: Take gradient descent step on	5: **end for**
6: $\nabla_\theta \|\|\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_\theta(\boldsymbol{x}_0 + \alpha_t\boldsymbol{\epsilon}, t)\|\|^2$	6: **return** $\boldsymbol{x}_0$
7: **until** converged	

Conditional Diffusion Models To incorporate conditional information, to control the data generation we give the *Decoder NN* **context**



Natural Language

Natural language is a **sequence of words**. Each word is a discrete unit, predicting the next part of the sequence means predicting words. Same applies for individual letters in a letter-sequence.

Probabilistic LM-Implementations

Tokenization

"They went to the grocery store; there, they bought bread, peanut butter, and jam."

- Consistent casing
- Strip punctuation
- One word is one token
- Split on spaces

```
["they", "went", "to", "the",
"grocery", "store", "there",
"they", "bought", "bread",
"peanut", "butter", "and", "jam"]
```

Attention: this is easier in english than other languages *(e.g. Chinese)*.

Vocabularies A *Vocabulary* is a set of all known words of the model. The **hyperparameter** `vocab_size` defines how many words are in the vocabulary. It is implemented by only keeping the vocab_size-most-important words, everything else is replaced by the **UNK** *(unknown)*-Token.

- Original sentence:
  - *"They galloped to the Ratty for dinner, and ate exactly seventy-three waffle fries and chocolate peamilk."*
- Tokenized:
  - ["they", "galloped", "to", "the", "ratty", "for", "dinner", "and", "ate", "exactly", "seventy-three", "waffle", "fries", "and", "chocolate", "peamilk"]
- UNKed:
  - ["they", "UNKed", "to", "the", "UNK", "for", "dinner", "and", "ate", "exactly", "UNK", "waffle", "fries", "and", "chocolate", "UNK"]

Maths

`P(any sequence)` is determined by `P(the words in the sequence)`.

Said differently, we can represent a sequence as $w_1, w_2, ... w_n$, and

$$P(w_1, w_2, ... w_n) = P(w_1) * P(w_2|w_1) * P(w_3|w_1, w_2) * \cdots P(w_n|w_1 ... w_{n-1})$$

telling probability

"The probability of a sentence is the product of the probabilities of each word given the previous words"

- This is an application of the ***chain rule for probabilities***

Punctuations are usually not embedded

Counting Counting is based on predicting a sentence based on the probability of this sentence appearing in the data. This strategy depends on having instances of sentence prefixes.

- *he shouted ___*
- Strategy: iterate through all words in vocabulary, and calculate $\frac{Count(he\ shouted\ <word>)}{Count(he\ shouted)}$ for each word
- What if our training sentences were:
  - *"He exclaimed loudly"*
  - *"She shouted angrily"*
  - *"He shouted angrily"*
  - *"She shouted loudly"*
  - *"He exclaimed frustratedly"*
- *"He shouted loudly"* has 0 probability!

To mitigate this dependency, **N-gram** counting can be used, that only looks at **N** words at a time.

(in this case, **bi**grams look at **2** words at a time)

- *"exclaimed loudly"*
- *"shouted angrily"*
- *"shouted angrily"*
- *"shouted loudly"*
- *"exclaimed frustratedly"*

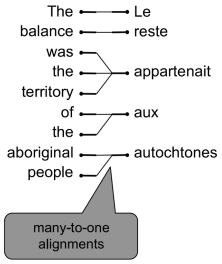
"He shouted loudly" now has 1/3 probability!

This approach still doesn't understand synonyms for either of the two words: He shouted frustratedly has *probability 0*.

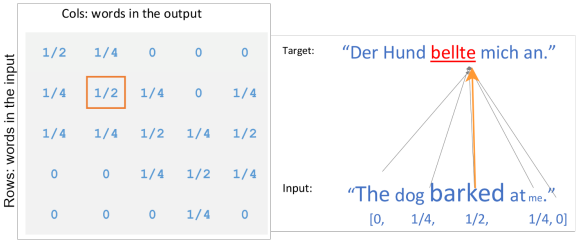
Attention

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**

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

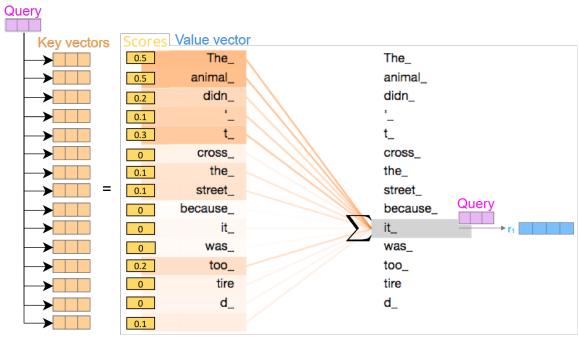
which is the *softmax of some predefined alignment score*. And this shows how well the two words y_t and x_i are aligned. For the score(...) function there are several options

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ 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

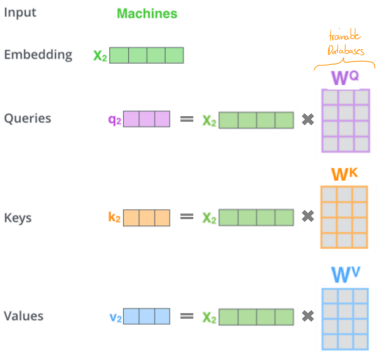
Self-attention computes the output vector r_i for each word via a weighted average of vectors extracted from each word in the input sentence. Here, self-attention learns that "it" should pay attention to "The animal".



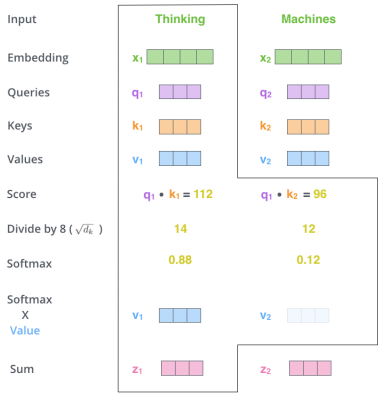
1. We compute a query vector for the word and compare it to a key vector for every other word. This computes a score.
2. For the output vector r_1 we sum all value vectors , weighted by the score in step 1.

- 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:

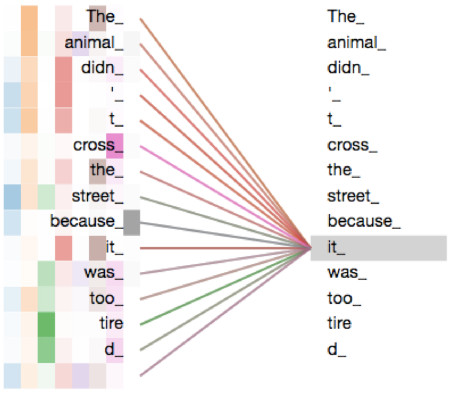


To get the **self-attention for Thinking**, we do:

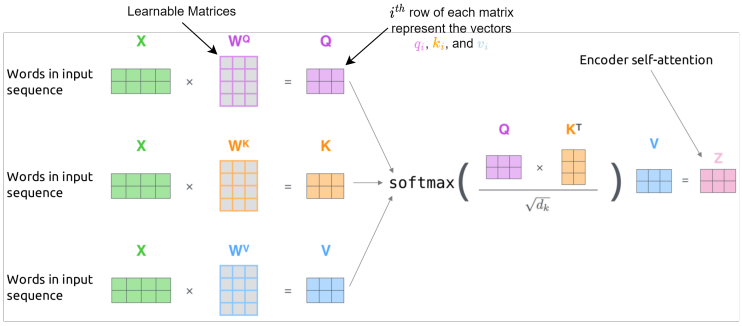


1. **Score**: Dot product of the query vector q_1 of "Thinking" with the key vectors $\{k_1, k_2\}$ of each word.

2. **Scale**: Divide each score by $\sqrt{\text{dimensionality } d_k}$ for more stable gradients.
3. **Softmax**: Apply softmax to transform scores into attention weights
4. **Weighting**: Multiply with the value vectors $\{v_1, v_2\}$
5. **Sum**: Sum the weighted value vectors into the final **self-attention for "Thinking"**



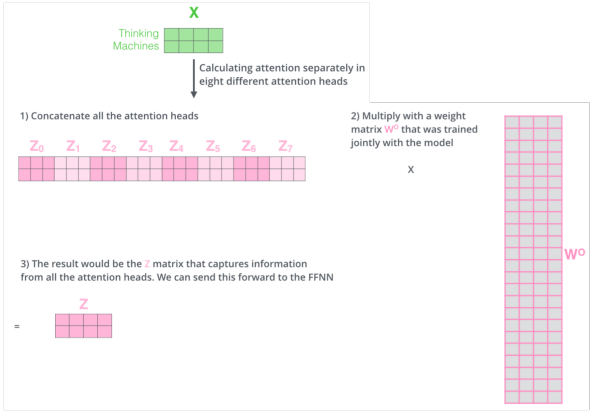
Computation



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.

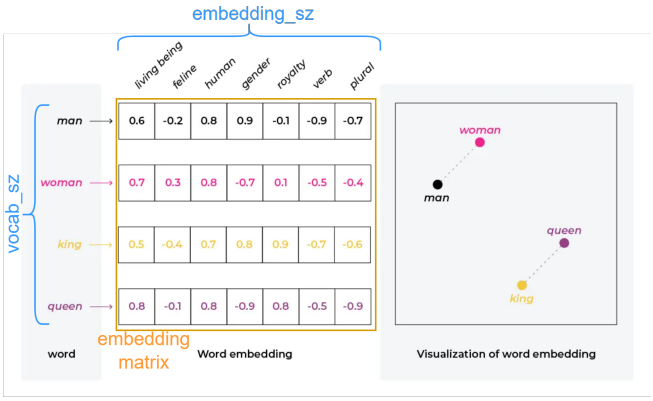
To get one set of attention vectors, we concatenate all the heads and apply a linear layer in order to get Z.



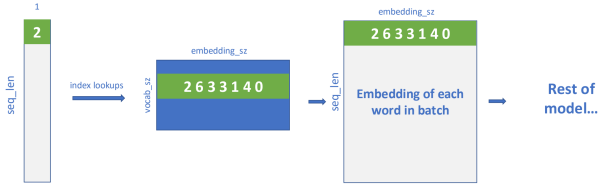
Multiple heads allow for each head to learn different relationships between words in the sentence. The network (W^O) then learns how to handle this. These multiple attentions can be visualised with this tool

Word Vectors

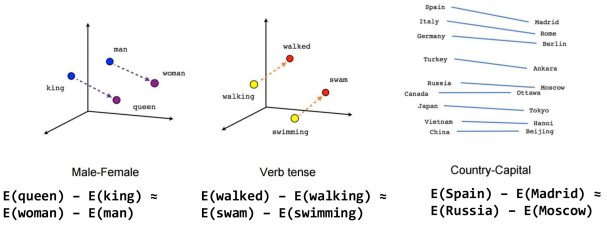
Words can be stored as **vectors**, the size of the vector is set by the embedding sz and is a hyperparameter. The embedding matrix is learnable and initialised randomly. Each embedding *can* have a meaning, but doesn't have to (trained with the network).



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

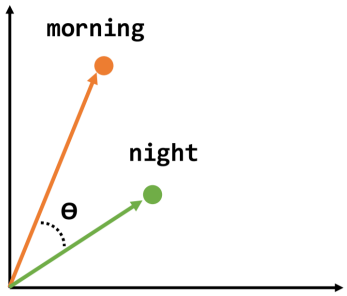


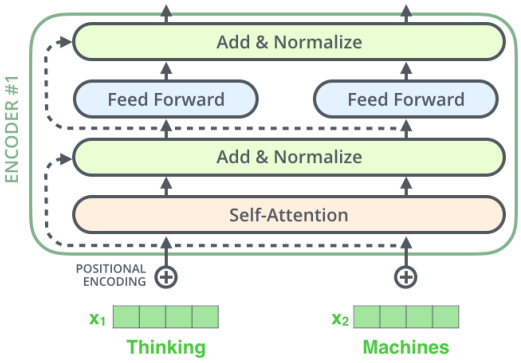
Semantic Directions In a trained embedded space, words get ordered by (random) meanings, words with the same meaning are clustered. Therefore related word pairs have same distances



The **similarity** can be quantified through the cosine similarity

$$\cos(\Theta) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

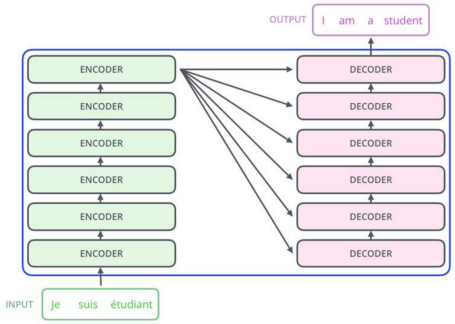




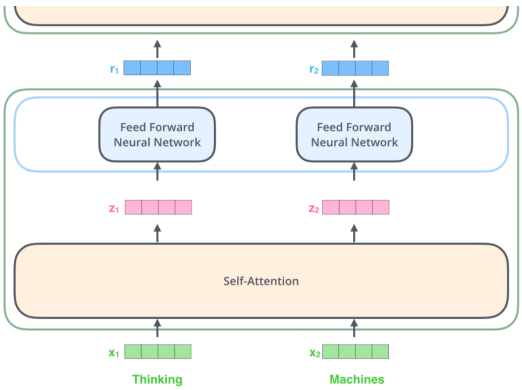
Transformer

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**.

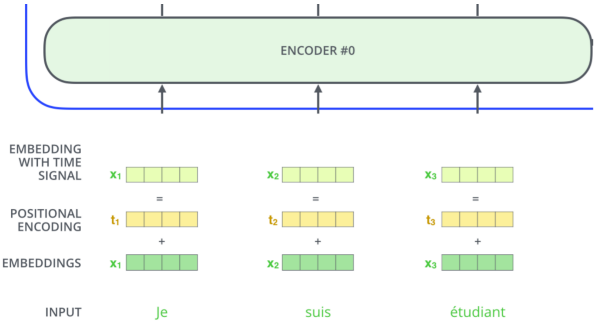


Encoder Block Map If we input a word sequence $\{x_1, x_2\}$, first the **self attention** is applied to get the value vectors $\{z_1, z_2\}$. Then a Feed Forward Network is applied to each word individually.

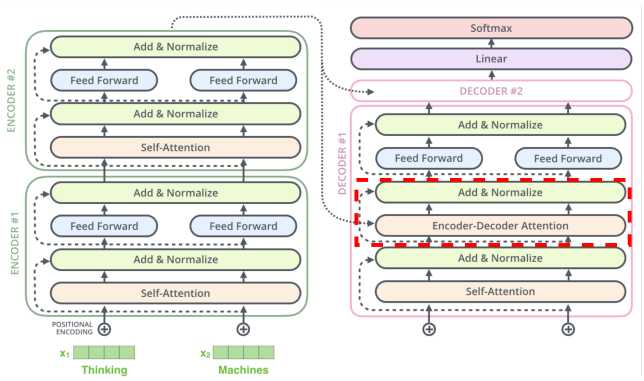


For performance improvments we add **Residuals** to negate vanishing gradients. Furthermore, we apply **LayerNorm**, which improves convergence time

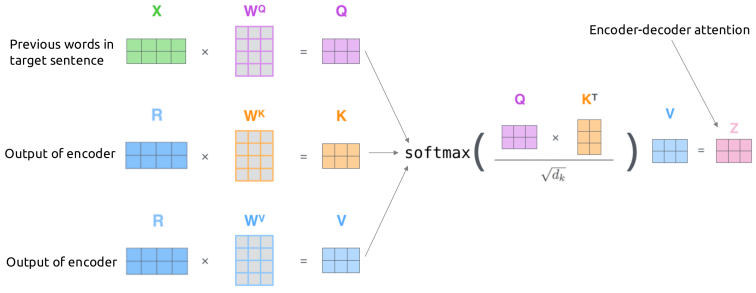
Positional Encoding Instead of passing a Embedding vector to the encoder, we pass a **Embedding with Time Signal** vector. Positional encoding can be learned or defined by a fixed function.



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**.



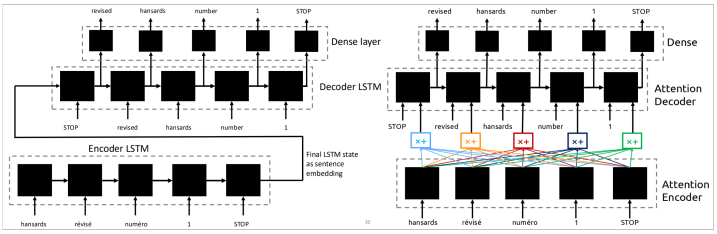
The **Encoder-Decoder Attention** takes the output of the encoder instead of the input words



Sequence-to-Sequence

Because rule based translation (word for word) often don't work, algorithms are trained on **parallel corpora**, documents which are available in several languages (e.g. legal documents of multilingual countries).

To go from one language to another, a sentence meaning must be encoded in on language and decoded in another language. This can be done through **Attention** or a **LSTM-Autoencoder** architecture.



Applications: Seq-to-Seq can also be used for Summaries, Chatbots, part of Speech Tagging, Speech Recognition, Speech Generation.

Machine Translation Evaluation

There are two measures to evaluate how good a translation is:

- **BLEU**: **B**i-**L**ingual **E**valuation **U**nderstudy, looking for a fraction of words generated that are in a given ground-truth sentence. Favors shorter sentences.
- **ROUGE**: **R**ecall-**O**riented **U**nderstudy for **G**isting **E**valuation, looking for a fraction of words in the correct translated sentence, that are in the generated sentence. Favors longer sentences.

This can be combined to the F_1-Score:

$$F_1 = \frac{2}{\frac{1}{BLEU} + \frac{1}{ROUGE}} = \frac{2(BLEU \cdot ROUGE)}{BLEU + ROUGE}$$

> ⚠ F_1-Score is Dumb
>
> Because the F_1-Score can't understand meaning of sentences. Therefore differently translated sentences can be wrongfully punished.
> To tackle this Human Evaluation is used.

Audio and Video Synthesis

Time Series Forecasting

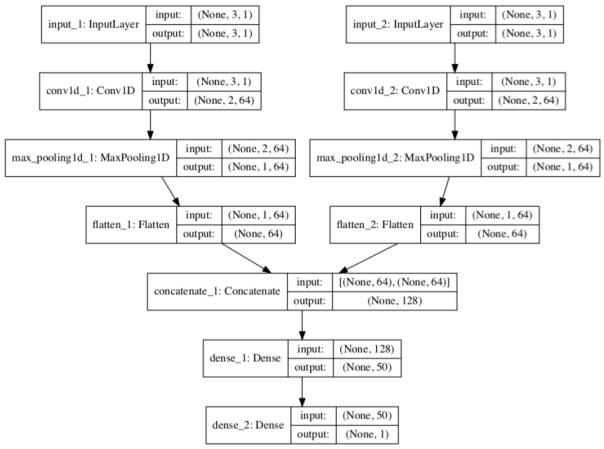
Sequence: Ordered datapoints *(doesn't have to be timedependent)*

Time-series (simple univariate): A single series of observations with temporal ordering

Traditionally **autoregression** is used to predict time-series, each output is predicted based on past outputs. For example:

$$X(t) = b_0 + b_1 * X(t-1) + b_2 * X(t-2)$$

Deep Learning **1D CNN**: convolutional layers where the input variable is a window of previous time series values. Can be extended to multivariate data through concatenating outputs of multiple 1D CNNs



A well studied example of time-series data is Natural Language.

Search using Deep Reinforcment Learning

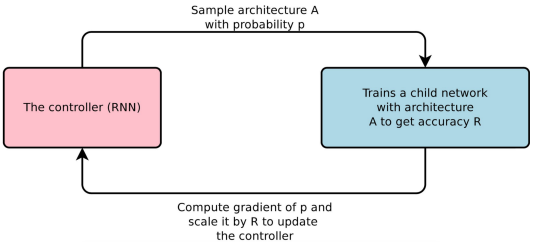
Anomaly Detection

Irregular Networks

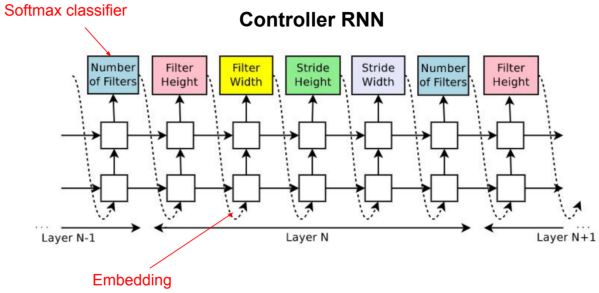
Neural Architecture Search with Reinforcment Learning

To automate the task of finding optimal network structures, we use **Neural Architecture Search**

- Specify structure using a configuration string: `["Filter Width: 5", "Filter Height: 3", "Num Filters: 24"]`
- Use a RNN **controller** to generate string
- Train this architecture (**child network**) to check performance
- Use **reinforcement learning** to update the parameters of the controller model

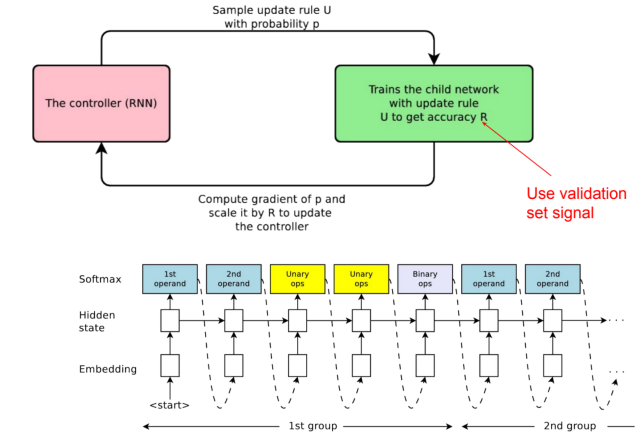


The controller RNN has this structure:



Neural Optimizer Search Optimizers are also hard to design, many different optimizers exist such as ADAM, RMSProp, ADADelta, Momentum, SGD, ...

We can use the previous method to also search over optimizers.



The controller iteratively selects subsequences of length 5.

- First selects the 1st and 2nd operands op1 and op2
- Selects 2 unary functions u1 and u2 to apply to the operands
- Selects a binary function b that combines the outputs of the unary functions.
- The resulting b(u1(op1), u2(op2)) then becomes an operand that can be selected in the subsequent group of predictions or becomes the update rule.

(Energy) Optimized Deep Learning

Deep Learning is extremely power hungry

Consumption	CO_2e (lbs)
Air travel, 1 passenger, NY↔SF	1984
Human life, avg, 1 year	11,023
American life, avg, 1 year	36,156
Car, avg incl. fuel, 1 lifetime	126,000
Training one model (GPU)	
NLP pipeline (parsing, SRL)	39
w/ tuning & experimentation	78,468
Transformer (big)	192
w/ neural architecture search	626,155

Table 1: Estimated CO_2 emissions from training common NLP models, compared to familiar consumption.[1]

Consumer	Renew.	Gas	Coal	Nuc.
China	22%	3%	65%	4%
Germany	40%	7%	38%	13%
United States	17%	35%	27%	19%
Amazon-AWS	17%	24%	30%	26%
Google	56%	14%	15%	10%
Microsoft	32%	23%	31%	10%

Table 2: Percent energy sourced from: Renewable (e.g. hydro, solar, wind), natural gas, coal and nuclear for the top 3 cloud compute providers (Cook et al., 2017), compared to the United States,[4] China[5] and Germany (Burger, 2019).

To mitigate this issue, we have to make models more efficient.

Mid Term:

- Prioritizing computationally efficient networks (e.g. Mobile Net)
- Report training time, hardware, and *hyperparameter sensitivity* (for awareness)
- Network Pruning
  1. Random weight initialization on large network
  2. After training cut connections with weights ≈ 0 (only 10-20% have meaningful weights)
  3. Drop (**prune**) 80-90% of connections and re-train from here on
  4. $\rightarrow$ increased network speed / decreased power consumption
  5. **the network trains well, sometimes even better than the full network**
- Network Quantization
  - Reduce the number of bits representing a number (float32 $\rightarrow$ bfloat16)
  - Quantization-aware training to reduce precision to four bits with accuracy losses 2-10%
- Tensor Processing Unit
  - Special purpose GPU without the graphics part and high throughput at low precision
  - Specially designed for specific frameworks (Google TPU $\leftarrow$ TensorFlow)
  - Edge versions for inference only
  - **TPUs and AI-specialised GPUs are getting more similar**

Long Term:

- More efficient *physical substrates*
  - Massive GPU parallelism is powerhungry
  - Are there other physical computing devices? (e.g. **optical compute**)
  - Neumorphic Computation
- "Physical" Networks
  - ASIC, or Co-Processor
  - Analog Circuits, **memristors**
  - $\rightarrow$ 1000s of times more energy efficient