# Robotics Knowledge

Willow Mandil

June 2022

## Contents



`https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html`

# 1 Machine Learning

## 1.1 Linear Algebra and Calculus

### General Notation

- Vectors - Denoted as $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x}_i$ is the ith element.
- Matrix - Denoted as $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{A}_{i,j}$ is the ith row and the jth column.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \quad , \quad \mathbf{A} = \begin{bmatrix} A_{1,1} & \dots & A_{1,n} \\ \vdots & & \vdots \\ A_{m,1} & \dots & A_{1,1} \end{bmatrix} \tag{1}$$

- Identity Matrix - $I \in R^{n \times n}$, 1's along the diagonal.
- Diagonal Matrix - $D \in R^{n \times n}$, values only along the diagonal, $D = diag(d_1, \dots, d_n)$

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad , \quad \mathbf{A} = \begin{bmatrix} d_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & d_n \end{bmatrix} \tag{2}$$

### Matrix Operations

- Vector × Vector - Two types of v × v product:
  1. The Inner Product: Given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, we have:
  2. The Inner Product: Given $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$, we have:

$$\mathbf{x}^T\mathbf{y} = \sum_{i=1}^{n} x_i y_i \in \mathbb{R} \quad , \quad \mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_1 y_1 & \dots & x_1 y_n \\ \vdots & \ddots & \vdots \\ x_m y_1 & \dots & x_m y_n \end{bmatrix} \tag{3}$$

- Matrix × Vector - The product of $A \in \mathbb{R}^{m \times n}$ and vector $x \in \mathbb{R}^n$ is a vector of size $\mathbb{R}^m$

$$Ax = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix}$$

- Matrix × Matrix - The product of $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ is a vector of size $\mathbb{R}^{n \times p}$

$$Ax = \sum_{i=1}^{n} a_{c,i} x_i \in \mathbb{R}^m = \begin{bmatrix} a_{r,1}^T x \\ \vdots \\ a_{r,m}^T x \end{bmatrix} \quad , \quad AB = \sum_{i=1}^{n} A_{r,i} B_{i,c} \tag{4}$$

- Transpose - $\forall_{i,j} = A_{i,j}^T = A_{j,i}$ note: $(AB)^T = B^T A^T$
- Inverse - $AA^{-1} = A^{-1}A = I$ note: $(AB)^{-1} = B^{-1}A^{-1}$
- Trace - Sum of the diagonal entries
  $tr(A) = \sum_{i=1}^{n} A_{i,i}$ note: $tr(A^T) = tr(A)$ and $tr(AB) = tr(BA)$
- Determinant - of a square matrix $A \in \mathbb{R}^{n \times n}$ and noted as $|A|$ or $\det(A)$ is below, note: $|AB| = |A||B|$ and $|A^T| = |A|$

$$|A| = sum_{i=1}^{n} (-1)^i A_{\neg r, i \neg c, j} \tag{5}$$

### Matrix Properties

- Symmetric Decomposition - Matrix A can be expressed as:

$$A = \underbrace{\frac{A + A^T}{2}}_{\text{Symmetric}} + \underbrace{\frac{A - A^T}{2}}_{\text{Antisymmetric}} \tag{6}$$

- Norm - a function $N : V \to [0, +\infty]$, V = vector space, such that for all $x, y \in V$ we have:
  1. $N(x + y) \leq N(x) + N(y)$
  2. $N(ax) = |a|N(x)$ for a scalar
  3. $if N(x) = 0$, then $x = 0$

$$||x||_p = (\sum_{i=1}^{n} x_i^p)^{\frac{1}{p}} \tag{7}$$

- Linearly Dependence - A set of vectors are linearly dependant if one of the vectors in the set can be defined as a linear combination of the others.
- Matrix Rank - Number of columns of a matrix
- Positive Semi-Definate Matrix - A matrix $A \in R^{n \times n}$ is PSD and denoted as $A \succeq 0$ if we have:

$$A = A^T \quad \text{and} \quad \forall x \in R^n \ , \ x^T A x \geq 0 \tag{8}$$

- EigenValue and Eigenvector - Given square matrix $A \in R^{n \times n}$, $\lambda$ is said to be an eigenvalue of $A$ if there exists a vector $v \in R^n \setminus \{0\}$, called eigenvector, such that we have:

$$Av = \lambda v \tag{9}$$

- Diagonalisation - conversion of a square matrix $A \in R^{n \times n}$ to a diagonal matrix, $D$ consisting of the eigenvales of $A$ and $X$ is made of the eigenvectors of $A$ (process only works if $A$ has unique eigenvalues or duplicate eigen values with linearly independent eigenvectors)

$$A = XDX^{-1} \quad = \quad X^{-1}AX = D \tag{10}$$

- Spectral Theorem - Given square matrix $A \in R^{n \times n}$. $A$ is diagonalisable if there is a basis
- Singular-value Decomposition -

### Matrix Calculus

- Gradient -
- Hessian -
- Gradient Operations -

## 1.2 Probabilities and Statistics

### Basics & Combinatorics

- Sample Space - the set of all possible outcomes of an experiment, denoted as $S$.
- Event - A subset of possible outcomes, $E$ from $S$.
- Axioms of Probability - The probability of each event $E$ occurring. With the 3 axions:

$$(1)\ 0 \leq P(E) \geq 1 \quad (2)\ \ P(S) = 1 \quad (3)\ \ P(\bigcup_{i=1}^{n} E_I) = \sum_{n}^{i=1} P(E_i) \tag{11}$$

- Permutation and Combination - The permutation and combination are an arrangement of $r$ objects from a pool of $n$ objects, in a given order, or without order respectively. The number of such arrangements is given by $P(n,r)$ and $C(n,r)$:

$$P(n,r) = \frac{n!}{(n-r)!} \quad (2) \quad C(n,r) = \frac{P(n,r)}{r!} - \frac{n!}{r!(n-r)!} \tag{12}$$

### Conditional Probabilities

- Bayes Rule - For events $A$ and $B$ such that $P(B) > 0$. $P(A|B)$ is the prob that A occurs given B. $P(A \cup B)$ is $A$ or $B$.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \quad \text{where } A|B = \text{A given B} \tag{13}$$

$$P(A \cap B) = P(A)P(B|A) = P(A|B)(B), \quad \text{where } \cap = \text{and} \tag{14}$$

- Partition - ?
- Extended Bayes Rule - Let $\{A_i, i \in [1,n]\}$be a partition if the sample space. We have:

$$P(A_k|B) = \frac{P(B|A_k)P(A_k)}{\sum_{i=1}^{n} P(B|A_i)P(A_i)} \tag{15}$$

- Partition - Two events $A$ and $B$ are independent only if we have:

$$P(A \cap B) = P(A)P(B) \tag{16}$$

### Random Variables

- Random Variable - A random variable, often noted a $X$, is a function that maps every element in a sample space to a real line.
- Cumulative Distribution Function (CDF) - The CDF, $F$c which is monotonically non-decreasing and is such that $\lim_{x \to -\infty} f(x) = 0$ and $\lim_{x \to +\infty} f(x) = 1$ is defined as:

$$F(x) = P(X \leq x) \tag{17}$$

- Cumulative Distribution Function (CDF) - The CDF, $F$c which is monotonically non-decreasing and is such that $\lim_{x \to -\infty} f(x) = 0$ and $\lim_{x \to +\infty} f(x) = 1$ is defined as:

$$F(x) = P(X \leq x) \tag{18}$$

- Cumulative Distribution Function (CDF) - The CDF, $F$c which is monotonically non-decreasing and is such that $\lim_{x \to -\infty} f(x) = 0$ and $\lim_{x \to +\infty} f(x) = 1$ is defined as:

$$F(x) = P(X \leq x) \tag{19}$$

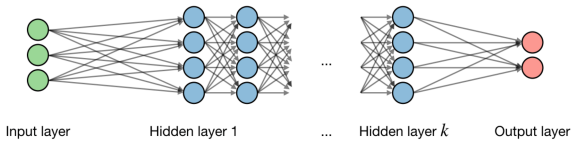
- Cumulative Distribution Function (CDF) - The CDF, $F$c which is monotonically non-decreasing and is such that $\lim_{x \to -\infty} f(x) = 0$ and $\lim_{x \to +\infty} f(x) = 1$ is defined as:

$$F(x) = P(X \leq x) \tag{20}$$

## 1.3 Deep Learning

**Neural Networks** Neural networks are a class of models that are built with layers. Commonly used types of neural networks include convolutional and recurrent neural networks.

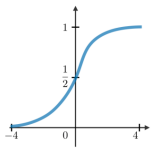
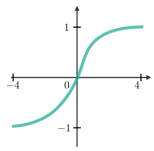
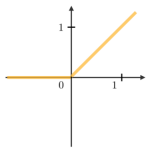
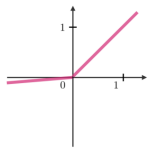
- Architecture - Basic vocab shown below:




Noting $i$ the $i^{th}$ layer of the network and $j$ the $j^{th}$ hidden unit of the layer, we have below, where $w$, $b$, $z$ are the weight, bias and output.

$$z_j^{[i]} = w_j^{[i]^T} x + b_j^{[i]}$$

- Activation Function - are used at the end of a hidden unit to introduce non-linear complexities to the model, x-axis input, y axis value output.

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

- Cross Entropy Loss - measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. Cross entropy loss $L(z, y)$ is:

$$L(z, y) = -\sum_{c=1}^{M} y_{o,c} log(z_{o,c})$$

where:

  - $M$ = number of predicted classes;
  - $y$ = binary indicator (0 or 1) if class label $c$ is the correct classification for observation $o$;
  - $z$ = predicted probability observation $o$ is of class $c$.

- Learning Rate - Often noted as $\eta$, indicates at which pace the weights get updated. This can be fixed or adaptively changed. Adam is the current adaptive standard.

- Back Propagation - Method to update the weights by taking into account the actual and desired output. The derivative with respect to weight $w$ is computed using the chain rule and is (left). As a result the weight is updated as (right):

$$\frac{\partial L(z,y)}{\partial w} = \frac{\partial L(z,y)}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w} \quad , \quad w \leftarrow w - \eta \frac{\partial L(z,y)}{\partial w}$$

- Updating the Weights - weights are updated as follows:

  - Step 1: Take a batch of training data.
  - Step 2: Perform forward propagation to obtain the corresponding loss.
  - Step 3: Backpropagate the loss to get the gradients.
  - Step 4: Use the gradients to update the weights of the network.

### Convolutional Neural Networks

- Conv Layer Requirements - By noting $W$ the input volume size, $F$ the size of the conv layer neurons, $P$ the amount of zero padding, then the number of neurons $N$ that fit in a given volume is such that:

$$N = \frac{W - F + 2P}{S} + 1$$

- Batch Normalisation - is a step of hyperparameter $\gamma$, $\beta$ that normalizes the batch $x_i$. By noting $\mu_B$ , $\sigma_b^2$ the mean and varience of that we want to correct to the batch, it is done as below. It is usually done after a fully connected/convolutional layer and before a non linearity layer. It aims at allowing higher learning rates and reducing the strong dependence on initialisation.

$$x_i \leftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

### Recurrent Neural Networks

- Types of Gates - Here are the different types of gates that we encounter in a typical RNN:

| Input Gate | Forget Gate | Output Gate | Gate |
|---|---|---|---|
| Write to cell or not? | Erase a cell or not? | Reveal a cell or not? | How much Writing? |

- LSTM - Long Short Term Memory network is an RNN that avoids the vanishing gradient problem by adding 'forget' gates.

### Reinforcement Learning and Control

The goal of RL is for an agent to learn how to evolve in an environment.

- Markov Decision Process - An MDP is a 5-tuple ($S$, $A$, $\{P_{sa}\}$, $\gamma$, $R$) where:

  - $S$ is the set of state
  - $A$ is the set of actions
  - $\{PS_{sa}\}$ are the state-action transition probabilities for $s \epsilon S$ and $a \epsilon A$
  - $\gamma \epsilon [0, 1]$ is the discount factor
  - $R : S \times A \rightarrow \mathbb{R}$ or $R : S \rightarrow \mathbb{R}$ is the reward function that the algorithm wants to maximise

- Policy - A policy $\pi$ is a function $\pi S \rightarrow A$ that maps states to actions. We sat that we execute a given policy $\pi$ if a given state $s$ we take the action $a = \pi(s)$

- Value Function - For a given policy $\pi$ and a given state $s$, we define the value function $V^\pi$ as:

$$V^\pi(s) = E[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...|s_0 = s, \pi]$$

- Bellman Equation - The optimum Bellman equation characterises the value function $V^{\pi^*}$ of the optimal policy $\pi^*$ (left). We note that the optimal policy $\pi^*$ for a given state $s$ is such that (right)

$$V^{\pi^*}(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^{\pi^*}(s')$$

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')$$

- Value Iteration Algorithm - two steps:
  - We initialise the value: $V_0(s) = 0$
  - We iterate the value based on the values before:

$$V_{i+1}(s) = R(s) + \max_{a \in A} \left[ \sum_{s' \in S} \gamma P_{sa}(s') V_i(s') \right]$$

- Maximum Liklihood Estimate - The MLE for the state transition probabilities are as follows:

$$P_{sa}(s') = \frac{\#\text{ times took action } a \text{ in state } s \text{ and got to } s'}{\#\text{ times took action } a \text{ in state } s}$$

- Q-Learning - is a model-free estimation of $Q$, which is done as:
$Q(s,a) \leftarrow Q(s,a) + \alpha \left[ R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$

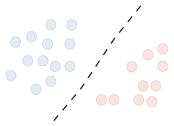
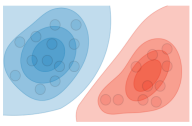
## 1.4 Supervised Learning

**Introduction** Given a set of data points $\{x^{(1)}, ..., x^{(m)}\}$ associated to a set of outcomes $\{y^{(1)}, ..., y^{(m)}\}$, we want to build a classifier that learns how to predict $y$ from $x$

- Type of Prediction - types of predictive models are summed up below:

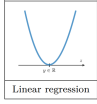
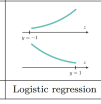
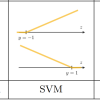
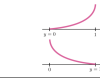
| | Regression | Classifier |
|---|---|---|
| **Outcomes** | Continuous | Class |
| **Examples** | Linear Regression | Logistic regression, SVM, Naive Bayes |

- Types of model - The different models are summed up in below:

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

### Notation and General Concepts

- Hypothesis - notes as $h_\theta$, it is the model that we chose. For a given input $x^{(i)}$, the model prediction output is $h_\theta(x^{(i)})$
- Loss Function - A function $L : (x, y) \in \mathbb{R} \times Y \to L(z, y) \in \mathbb{R}$ that takes as inputs the predicted vale $z$ corresponding to the real data valued $y$ and outputs how different they are. The common loss functions are below

| Least squared | Logistic | Hinge | Cross-entropy |
|---|---|---|---|
| $\frac{1}{2}(y-z)^2$ | $\log(1+\exp(-yz))$ | $\max(0, 1-yz)$ | $-\left[y\log(z)+(1-y)\log(1-z)\right]$ |
|  |  |  |  |
| Linear regression | Logistic regression | SVM | Neural Network |

- Cost Function - The cost function $J$ is commonly used to assess the performance of a model and is defined with the loss function $L$ as follows:

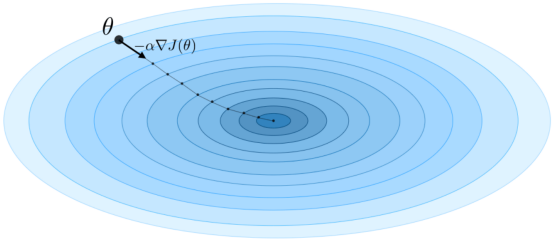
$$J(\theta) = \sum_{i=1}^{m} L(h_\theta(x^{(i)}), y(i))$$

- Gradient Descent - By noting $\alpha \in \mathbb{R}$ the learning rate, the update rule for gradient descent is expressed with the learning rate and the cost function $J$ as follows:

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

- Stochastic Gradient Descent (SDG) - is updating the parameter based on each training example, and batch gradient descent is on a batch of training examples.
- Liklihood - The liklihood of a model $L(\theta)$ given parameters $\theta$ is used to find the optimal parameters $\theta$ through maximising the liklihood. In practice, we use the log-liklihood $\ell(\theta) = \log(L(\theta))$ which is easier to optimize.

$$\theta^{\text{opt}} = \arg\max_\theta L(\theta)$$




- Newton's Algorithm - a numerical method that finds $\theta$ such taht $\ell'(\theta) = 0$. The rule is below. The multidimensional generalisation, also known as Newton-Raphson method, has the following update rule:

$$\theta \leftarrow \theta - \frac{\ell'(\theta)}{\ell''(\theta)} \quad , \quad \theta \leftarrow \theta - (\nabla^2_\theta \ell(\theta))^{-1} \nabla_\theta \ell(\theta)$$

**Linear Regression** We assume here that $y|x; \theta \sim \mathcal{N}(\mu, \sigma^2)$

- Normal Equations - By noting $X$ the matrix design, the value of $\theta$ that minimises the cost function is a closed-form solution that:

$$\theta = (X^T X)^{-1} X^T y$$

- Least Mean Squares algorithm - By noting $\alpha$ the learning rate, the update rule of the LMS algorithm for a training set of $m$ data points, which is also known as the Widow-Hoff learning rule is as follows: (the update rule is a particular case of the gradient ascent)

$$\forall j, \quad \theta_j \leftarrow \theta_j + \alpha \sum_{i=1}^{m} [y^{(i)} - h_\theta(x^{(i)})] x_j^{(i)}$$

- Locally Weighted Regression - LWR is a variant of linear regression that weights each training excample in its cost function by $w^{(i)}(x)$, which is defined with parameter $\tau \in \mathbb{R}$ as:

$$w^{(i)}(x) = \exp(-\frac{(x^{(i)} - x)^2}{2\tau^2})$$

### Classification and Logistic Regression

- Sigmoid Function - The sigmoid function $g$, also known as the logisitc function, is:

$$\forall z \in \mathbb{R}, \quad g(z) = \frac{1}{1 + \mathrm{e}^{-z}} \in [0, 1]$$

- Logistic Regression - We assume here that $y|x; \theta \sim$ Bernoulli$(\theta)$. We have the following form: (there is no closed loop form solution for the case of logistic regressions)

$$\phi = p(y = 1|x; \theta) = \frac{1}{1 + \exp(-\theta^T x)} = g(\theta^T x)$$

- Softmax Regression - A softmax regression, also called a multiclass logistic regression, is used to generalize logistic regression when there are more than 2 outcome classes. By convention, we set $\theta_k = 0$, which makes the Bernoulli parameter $\phi_i$ of each class $i$ equal to:

$$\phi_i = \frac{\exp(\theta_i^t) x}{\sum_{j=1}^{k} \exp(\theta_j^t) x)}$$

### Generalize Linear Models

- Exponential family - A class of distributions is said to be in the exponential family if it can be written in terms of a natural parameter, also called the canonical parameter or link function, $\eta$, a sufficient statistic $T(y)$ and a log-partition function $a(\eta)$ as follows:

$$0 - (X^T X)^{-1} X^T y$$

We will often have $T(y) = y$. Also, $\exp(-a(\eta))$ can be seen as a normalization parameter that will make sure that the probabilities sum to one. The most common exponential distributions are below:

| **Distribution** | $\eta$ | $T(y)$ | $a(\eta)$ | $b(y)$ |
|---|---|---|---|---|
| Bernoulli | $\log\left(\frac{\phi}{1-\phi}\right)$ | $y$ | $\log(1+\exp(\eta))$ | 1 |
| Gaussian | $\mu$ | $y$ | $\frac{\eta^2}{2}$ | $\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{y^2}{2}\right)$ |
| Poisson | $\log(\lambda)$ | $y$ | $e^\eta$ | $\frac{1}{y!}$ |
| Geometric | $\log(1-\phi)$ | $y$ | $\log\left(\frac{e^\eta}{1-e^\eta}\right)$ | 1 |

- Assumptions of Generalised Linear Models - (GLSM's) aim at predicting a random variable $y$ as a function of $x \in \mathbb{R}^{n+1}$ and rely on the following 3 assumptions:

1. $y|x;\theta \sim \text{ExpFamily}(\eta)$
2. $h_\theta(x) = E[y|x;\theta]$
3. $\eta = \theta^T x$

**Support Vector Machines** The goal of support vector machines is to find the line that maximizes the minimum distance to the line.

- Optimal Margin Classifier - The optimal margin classifier $h$ is such that (below). Where $(w, b) \in R^n \times (R)$ R is the solution of the following optimization problem:

$$h(x) = \text{sign}(w^T x - b)$$

$$\min \frac{1}{2}||w||^2 \quad \text{such that} \quad y^{(i)}(w^T x^{(i)} - b) \geq 1$$

# 2 Deep Learning

## 2.1 Backpropagation

### Overview

- Objective - To adjust each weight in the network in proportion to how much it contributes to overall error.
- Composit Functions - Meaning functions inside functions. Example for $\cos(x^2)$, we can let $\cos(x)$ be $f(x)$ and $x^2$ be $g(x)$, then $\cos(x^2) = f(g(x))$
- Chain Rule Maths - It tells us **how to differentiate composite functions**.

$$\frac{d}{dx}\big[f(g(x))\big] = f'(g(x)) \cdot g'(x)$$

- Chain Rule - Forward propagation is a set of nested equations. Therefore backprop is an application of the chain rule to find the derivatives of the cost with respect to any variable in the nested equation. FP is:

$$f(x) = A(B(C(x)))$$

$A$, $B$ and $C$ are activation functions at different layers. Using the chain rule we can calculate the derivative of $f(x)$ with respect to $x$;

$$f'(x) = f'(A) \cdot A'(B) \cdot B'(C) \cdot C'(x)$$

For derivative with respect to $B$, we set $B(C(x))$ to be a placeholder constant $B$ and find the derivative with respect to $B$:

$$f'(B) = f'(A) \cdot A'(B)$$

This simple technique extends to any variable within a function and allows us to precisely pinpoint the exact impact each variable has on the total output.

- Applying the Chain Rule - The chain rule will help us identify how much each weight contributes to our overall error and the direction to update each weight to reduce our error. Below are the functions and equations we need to make a prediction and then to calculate the total error/cost:

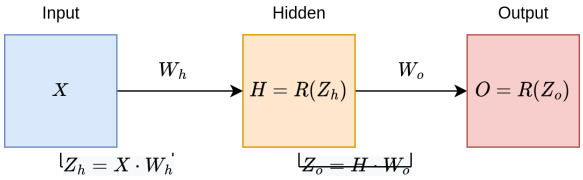
| Function | Formula | Derivative |
|---|---|---|
| Weighted input | $Z = XW$ | $Z'(X) = W$ , $Z'(W) = X$ |
| ReLU Activation | $R = \max(0, Z)$ | $R'(Z) = \begin{cases} 0 & \text{Z}< 0 \\ 1 & \text{Z}> 0 \end{cases}$ |
| Cost Function | $c = \frac{1}{2}(\hat{y} - y)^2$ | $C'(\hat{y}) = (\hat{y} - y)$ |

Given the network of just a single neuron, total cost is:

$$cost = C(R(Z(XW)))$$

Using the chain rule we can easily find the derivative of the Cost with respect to the weight $W$:

$$C'(W) = C'(R) \cdot R'(z) \cdot Z'(W) = (\hat{y} - y) \cdot R'(Z) \cdot X$$




What is the derivative with respect to $W_o$

$$C'(W_o) = C'(\hat{y}) \cdot \hat{y}'(Z_o) \cdot Z_o'(W_o) = (\hat{y} - y) \cdot R'(Z_o) \cdot H$$

And with respect to $W_h$? Just go further back in the function applying the chain rule recursively until you get to the function that has the $W_h$ term:

$$\begin{aligned} C'(W_h) &= C'(\hat{y}) \cdot O'(Z_o) \cdot Z_o'(H) \cdot H'(Z_h) \cdot Z_h'(W_h) \\ &= (\hat{y} - y) \cdot R'(Z_o) \cdot W_o \cdot R'(Z_h) \cdot X \end{aligned}$$

- 10 layer cost - For a model with 10 layers - What is the derivative of cost for the first weight $w_1$:

$$\begin{aligned} C'(w_1) = \frac{dC}{d\hat{y}} \cdot \frac{d\hat{y}}{dZ_{11}} \cdot \frac{dZ_{11}}{dH_{10}} \cdot \\ \frac{dH_{10}}{dZ_{10}} \cdot \frac{dZ_{10}}{dH_9} \cdot \frac{dH_9}{dZ_9} \cdot \frac{dZ_9}{dH_8} \cdot \frac{dH_8}{dZ_8} \cdot \frac{dZ_8}{dH_7} \cdot \frac{dH_7}{dZ_7} \cdot \\ \frac{dZ_7}{dH_6} \cdot \frac{dH_6}{dZ_6} \cdot \frac{dZ_6}{dH_5} \cdot \frac{dH_5}{dZ_5} \cdot \frac{dZ_5}{dH_4} \cdot \frac{dH_4}{dZ_4} \cdot \frac{dZ_4}{dH_3} \cdot \\ \frac{dH_3}{dZ_3} \cdot \frac{dZ_3}{dH_2} \cdot \frac{dH_2}{dZ_2} \cdot \frac{dZ_2}{dH_1} \cdot \frac{dH_1}{dZ_1} \cdot \frac{dZ_1}{dW_1} \end{aligned}$$

- Memoization - To reduce computational costs you can store repeating equations:

$$\begin{aligned} C'(W_3) &= (O - y) \cdot R'(Z_3) \cdot H_2 \\ C'(W_2) &= (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot H_1 \\ C'(W_1) &= (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot H_0 \\ C'(W_0) &= (O - y) \cdot R'(Z_3) \cdot W_3 \cdot R'(Z_2) \cdot W_2 \cdot R'(Z_1) \cdot W_1 \cdot R'(Z_0) \cdot X \end{aligned}$$

- Derivative of Cost W.R.T any weight - Let's return to our formula for the derivative of cost with respect to the output layer weight $W_o$: and we know we can replace the first part with our equation for output layer error $E_o \cdot H$ represents the hidden layer activation.

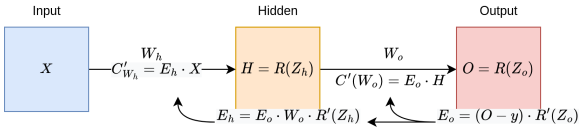
$$\begin{aligned} C'(W_o) &= (\hat{y} - y) \cdot R'(Z_o) \cdot H \\ &= E_o \cdot H \end{aligned}$$

So to find the derivative of cost with respect to any weight in our network, we simply multiply the corresponding layer's error times its input (the previous layer's output).

$$C'(w) = \text{CurrentLayerError} \cdot \text{CurrentLayerInput}$$

The final 3 equations that form the foundation of backpropagation are therefor:

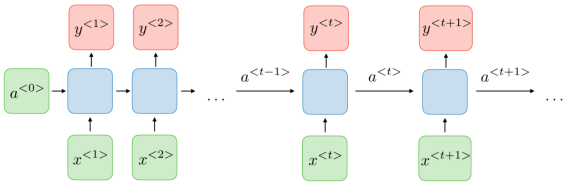
| | |
|---|---|
| Output Layer Error | $E_o = (o - y) \cdot R'(Z_o)$ |
| Hidden Layer Error | $E_h = E_o \cdot W_o \cdot R'(Z_h)$ |
| Cost-Weights Deriv | $\text{LayerError} \cdot \text{LayerInput}$ |



## 2.2 Recurrent Neural Networks

### Overview

- Architecture of a traditional RNN - RNNs are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:



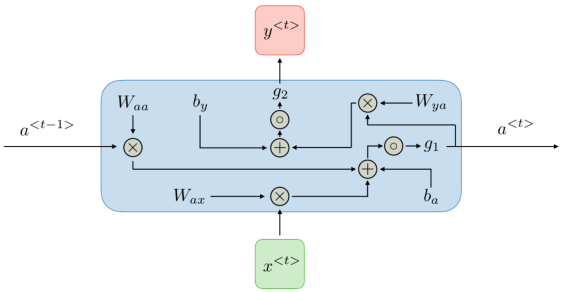
For each timestep $t$ and input $x^{<t>}$, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t1>} + W_{ax}x^{<t>} + b_a)$$

and

$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

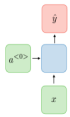
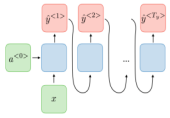
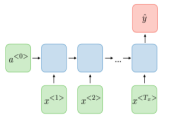
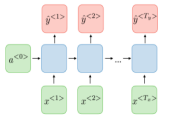
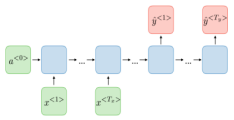
where: $W_{ax}$, $W_{aa}$, $W_{ya}$, $b_a$, $b_y$ are coefficients that are shared temporally and $g_1$, $g_2$ activation functions



- Typical RNN pro's and cons:

| Advantages | Drawbacks |
|---|---|
| - Possibility of processing input of any length<br>- Model size not increasing with size of input<br>- Computation takes into account historical information<br>- Weights are shared across time | - Computation being slow<br>- Difficulty of accessing information from a long time ago<br>- Cannot consider any future input for the current state |

- RNN applications - RNN models are mostly used in the fields of natural language processing and speech recognition. The different applications are summed up in the table below:

| Type of RNN | Illustration | Example |
|---|---|---|
| One-to-one<br>$T_x = T_y = 1$ |  | Traditional neural network |
| One-to-many<br>$T_x = 1, T_y > 1$ |  | Music generation |
| Many-to-one<br>$T_x > 1, T_y = 1$ |  | Sentiment classification |
| Many-to-many<br>$T_x = T_y$ |  | Name entity recognition |
| Many-to-many<br>$T_x \neq T_y$ |  | Machine translation |

- Loss Function - In the case of a recurrent neural network, the loss function $L$ of all time steps is defined based on the loss at every time step as:

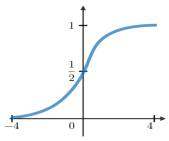
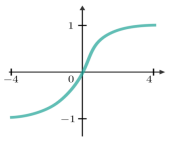
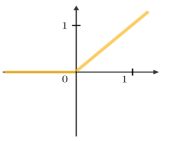
$$L(\hat{y}, y) = \sum_{t-1}^{T_y} L(\hat{y}^{<t>}, y^{<t>})$$

- Backpropagation - Backpropagation is done at each point in time. At timestep $T$, the derivative of the loss $L$ with respect to weight matrix $W$ is expressed as follows:

$$\frac{\delta L^{(T)}}{\delta W} = \sum_{t=1}^{T} \frac{\delta L^{(T)}}{\delta W}|_{(t)}$$

### Long Term Dependencies

- Common Activation Functions -

| Sigmoid | Tanh | RELU |
|---|---|---|
| $g(z) = \frac{1}{1+e^{-z}}$ | $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ | $g(z) = \max(0,z)$ |
|  |  |  |

- Vanishing/Exploding Gradient - Are phenomena often encountered in the context of RNNs. They happen because it is difficult to capture long term dependencies because of multiplicative gradients that can be exponentially decreasing/increasing with respect to the number of layers.

- Gradient Clipping - Is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By capping the maximum value for the gradient, this phenomenon is controlled.

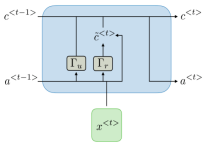
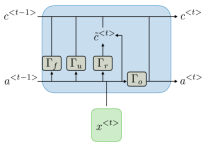
- Types of Gates - In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted $\Gamma$ and are equal to:

$$\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)$$

where $W$, $U$, $b$ are coefficients specific to the gate and $\sigma$ is the sigmoid function. The main ones are summed up in the table below:

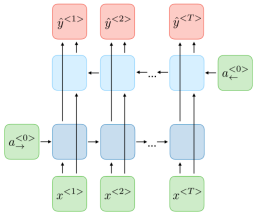
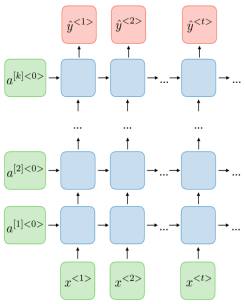
| Type of gate | Role | Used in |
|---|---|---|
| Update gate $\Gamma_u$ | How much past should matter now? | GRU, LSTM |
| Relevance gate $\Gamma_r$ | Drop previous information? | GRU, LSTM |
| Forget gate $\Gamma_f$ | Erase a cell or not? | LSTM |
| Output gate $\Gamma_o$ | How much to reveal of a cell? | LSTM |

- GRU/LSTM - Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU. Below is a table summing up the characterizing equations of each architecture:

| | Gated Recurrent Unit (GRU) | Long Short-Term Memory (LSTM) |
|---|---|---|
| $\tilde{c}^{<t>}$ | $\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$ | $\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$ |
| $c^{<t>}$ | $\Gamma_u \star \tilde{c}^{<t>} + (1 - \Gamma_u) \star c^{<t-1>}$ | $\Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$ |
| $a^{<t>}$ | $c^{<t>}$ | $\Gamma_o \star c^{<t>}$ |
| Dependencies |  |  |

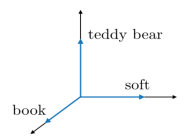
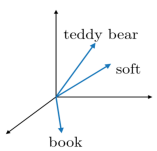
Remark: the sign $\star$ denotes the element-wise multiplication between two vectors.

- RNN variants -

| Bidirectional (BRNN) | Deep (DRNN) |
|---|---|
|  |  |

**Long Term Dependencies**
In this section, we note $V$ the vocabulary and $|V|$ its size.

- Learning word representations - The two main ways of representing words are summed up in the table below:

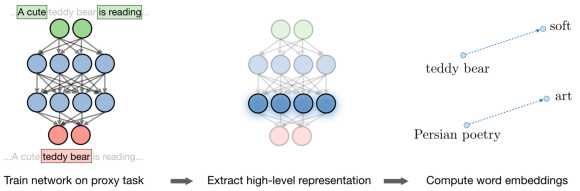
| 1-hot representation | Word embedding |
|---|---|
|  |  |
| - Noted $o_w$<br>- Naive approach, no similarity information | - Noted $e_w$<br>- Takes into account words similarity |

- Learning word representations - For a given word $w$, the embedding matrix $E$ is a matrix that maps its 1-hot representation $o_w$ to its embedding $e_w$ as follows:

$$e_w = E0_w$$

Remark: learning the embedding matrix can be done using target/context likelihood models

- Word2vec – Word2vec is a framework aimed at learning word embeddings by estimating the likelihood that a given word is surrounded by other words. Popular models include skip-gram, negative sampling and CBOW




- Skip-gram - The skip-gram word2vec model is a supervised learning task that learns word embeddings by assessing the likelihood of any given target word $t$ happening with a context word $c$. By noting $\theta_t$ a parameter associated with $t$, the probability $P(t|c)$ is given by:

$$P(t|c) = \frac{\exp(\theta_t^T e_c)}{\sum_{j=1}^{|V|} \exp(\theta_j^T e_c)}$$

Remark: summing over the whole vocabulary in the denominator of the softmax part makes this model computationally expensive. CBOW is another word2vec model using the surrounding words to predict a given word.

- Negative Sampling - Is a set of binary classifiers using logistic regressions that aim at assessing how a given context and a given target words are likely to appear simultaneously, with the models being trained on sets of $k$ negative examples and 1 positive example. Given a context word $c$ and a target word $t$, the prediction is expressed by:

$$P(y = 1|c, t) = \sigma(\theta_t^T e_c)$$

Remark: this method is less computationally expensive than the skip-gram model.

- GloVe - The GloVe model, short for global vectors for word representation, is a word embedding technique that uses a co-occurence matrix $X$ where each $X_{i,j}$ denotes the number of times that a target $i$ occurred with a context $j$. Its cost function $J$ is as follows:

$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{|V|} f(X_{ij})(\theta_i^T e_j + b_i + b_i' - \log(X_{ij}))^2$$

where $f$ is a weighting function such that $X_{i,j} = 0 \implies f(X_{i,j}) = 0$, Given the symmetry that $e$ and $\theta$ play in this model, the final word embedding $e_w^{(\text{final})}$ is given by:

$$e_w^{(\text{final})} = \frac{e_w + \theta_w}{2}$$

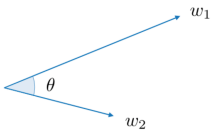
Remark: the individual components of the learned word embeddings are not necessarily interpretable.

**Comparing Words**

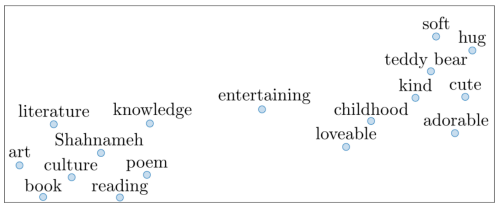
- Cosine Similarity - The cosine similarity between words w1 and w2 is expressed as follows:

$$\text{similarity} = \frac{w_1 \cdot w_2}{||w_1||||w_2||} = \cos(\theta)$$

Remark: $\theta$ is the angle between words $w_1$ and $w_2$




- $t$-SNE - (t-distributed Stochastic Neighbor Embedding) is a technique aimed at reducing high-dimensional embeddings into a lower dimensional space. In practice, it is commonly used to visualize word vectors in the 2D space.




### Language Models

- Overview - A language model aims at estimating the probability of a sentence $P(y)$
- $n$-gram model - This model is a naive approach aiming at quantifying the probability that an expression appears in a corpus by counting its number of appearance in the training data.
- Perplexity - Language models are commonly assessed using the perplexity metric, also known as PP, which can be interpreted as the inverse probability of the dataset normalized by the number of words $T$. The perplexity is such that the lower, the better and is defined as follows:

$$PP = \prod_{t=1}^{T} \left( \frac{1}{\sum_{j=1}^{|V|} y_j^t \hat{y}_j^t} \right)^{\frac{1}{T}}$$

Remark: PP is commonly used in $t$-SNE

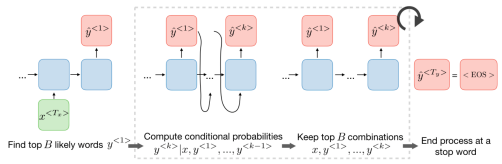
### Machine Translation

- Overview - A machine translation model is similar to a language model except it has an encoder network placed before. For this reason, it is sometimes referred as a conditional language model. The goal is to find a sentence $y$ such that:

$$y = \underset{y^{<1>},\ldots,y^{<T_y>}}{\arg\max} \; P(y^{<1>},\ldots,y^{<T_y>}|x)$$

- Beam Search - It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence $y$ given an input $x$.
  - Step 1: Find top $B$ likely words $y^{<1>}$
  - Step 2: Compute conditional probabilities $y^{<k>}|x, y^{<1>}, \ldots, y^{<k-1>}$
  - Step 3: Keep top $B$ combination $x, y^{<1>}, \ldots, y^{<k>}$

Remark: if the beam width is set to 1, then this is equivalent to a naive greedy search.




- Beam width - The beam width $B$ is a parameter for beam search. Large values of $B$ yield to better result but with slower performance and increased memory. Small values of $B$ lead to worse results but is less computationally intensive. A standard value for $B$ is around 10.

  Length normalization - In order to improve numerical stability, beam search is usually applied on the following normalized objective, often called the normalized log-likelihood objective, defined as:

$$\text{Objective} = \frac{1}{T_y^\alpha} \sum_{t=1} T_y \log\Big[\log p(y^{<t>}|x, y^{<1>}, \ldots, y^{<t-1>})\Big]$$

Remark: the parameter $\alpha$ can be seen as a softener, and its value is usually between 0.5 and 1

- Error Analysis - When obtaining a predicted translation $\hat{y}$ by that is bad, one can wonder why we did not get a good translation $y$ by performing the following error analysis:

| Case | $P(y^*\|x) > P(\hat{y}\|x)$ | $P(y^*\|x) \leqslant P(\hat{y}\|x)$ |
|---|---|---|
| **Root cause** | Beam search faulty | RNN faulty |
| **Remedies** | Increase beam width | - Try different architecture<br>- Regularize<br>- Get more data |

- Bleu Score - – The bilingual evaluation understudy (bleu) score quantifies how good a machine translation is by computing a similarity score based on n-gram precision. It is defined as follows:

$$\text{Bleu Score} = \exp\left(\frac{1}{n}\sum_{k=1}^{n} p_k\right)$$

where $p_n$ is the bleu score on $n$-gram only defined as:

$$p_n = \frac{\sum_{\text{n-gram}\in\hat{y}} \text{count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram}\in\hat{y}} \text{count}(\text{n-gram})}$$

### Attention

- Attention Model - This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice. By noting $\alpha^{<t,t'>}$ the amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ and $c^{<t>}$ the context at time $t$, we have:

$$c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \quad \text{with} \quad \sum_{t'} \alpha^{<t,t'>} = 1$$

Remark: the attention scores are commonly used in image captioning and machine translation.

A cute teddy bear is reading Persian literature

A cute teddy bear is reading Persian literature

- Attention weight - The amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ is given by $\alpha^{<t,t'>}$ computed as follows:

$$a^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t''=1}^{T_x} \exp(e^{<t,t''>})}$$

  Remark: computation complexity is quadratic with respect to $T_x$.

## 2.3 Transformers

### Attention

- Transformer Basic Architecture -

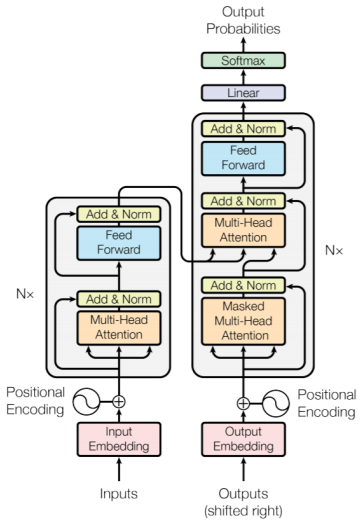



Figure 1: The Transformer - model architecture.

- Architecture Overview -

1. The transformer has the structure of neural sequence transduction models have an encoder-decoder structure.
2. The encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time
3. At each step the model is auto-regressive, consuming the previous symbols as additional input when generating the next.

- Key Components - Overall, the transformer takes a sequence as an input and outputs

1. Encoder and Decoder Stacks
2. Input Embedding
3. Positional Encoding
4. Multi-head Attention
5. Feed Forward Layer
6. Residual Layer
7. Output Embedding
8. Masked Multi-head Attention

- Encoder

1. The encoder is composed of a stack of $N = 6$ identical layers.
2. Each layer contains two sub layers: (i) Multi-head self attention and (ii) position-wise fully connected feed forward network. As well as residual connections around the two sub-layers followed by layer normalisation.

- Decoder

1. The encoder is composed of a stack of $N = 6$ identical layers.
2. Each layer contains three sub layers: (i) Masked Multi-head self attention (ii) Multi-head attention and (ii) position-wise fully connected feed forward network. As well as residual connections around the three sub-layers followed by layer normalisation.
3. The masked self attention network is modified to prevent hte positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures the predictions for position $i$ can depend only on the known outputs at positions less than $i$.

- Input Embedding - Learned embeddings convert the input tokens and output tokens to vectors of dimension $d_{\text{model}}$. We can do this in many ways:

1. One-hot encoding: mapping each input token to a distinct dimension in the vector space (this is shit though), for 50'000 words $d_{\text{model}} = 50'000$.
2. Linear layer that takes as input the one-hot encoding and converts it to a 256 dimension vector

The input $\mathbf{x}$ is embedded for a Transformer as, where $bs$ is batch size and $seq$ is the sequence length:

$$\mathbf{x} \in \mathbb{R}^{\text{bs}\times\text{seq}} \xrightarrow[\text{emb}]{} \mathbf{x} \in \mathbb{R}^{\text{bs}\times\text{seq}\times d_{\text{model}}}$$

- Attention Mechanism - For more information see Section 2.4 Common Neural Network Layers.

1. The queries and keys have dimension $d_k$
2. Multiplying the $QK^T$ by scaling factor of $frac1\sqrt{d_k}$
3. For large values of $d_k$, the dot products, $QK^T$, grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counter this effect they use a scaling factor of $frac1\sqrt{d_k}$

- Application of the Attention Mechanism - The transformer uses the attention mechanism in three different ways:

1.

- Why Self Attention - The transformer uses the multi-head attention in three different ways:

1. In the "encoder-decoder" layers, the queries come from the previous decoder layer and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models.
2.

- Position Encoding - As the model does not contain recurrence or convolution, to allow the model to make use of the order of the sequence we add information about the relative or absolute position of the tokens in sequence space.

1. Add the position feature vector to the input embeddings.
2. Position encodings have the same dimension as $d_{\text{model}}$
3. There are a variety of methods to do positional encoding
4. In the paper "attention is all you need" they use sine and cosine functions of different frequencies:

$$PE_{pos,2i} = sin(pos/10000^{2i/d_{\text{model}}}) \tag{21}$$

$$PE_{pos,2i+1} = cos(pos/10000^{2i/d_{\text{model}}}) \tag{22}$$

5. Where $pos$ is the position and $i$ is the dimension
6. learning the positional embeddings produced almost identical results as the above sin & cos method

- Residual Dropout - Apply dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of $P_{drop} = 0.1$
- Inference Time - Feed as the "target" sequence in the training step all start tokens, after one inference step, the output is then added to the "target" etc. For sentence production, the input is the target. For translation, the input becomes the new language one step at a time.

**Video Transformers**

- Introduction -

## 2.4 Common Neural Network Layers

### Normalisation

- Normalise and Standardise - Both are pre-processing stages where we prep the data for training an NN. Both transform the data place it all on the same scale.
- Normalise - scale the numerical data down to a scale of 0-1
- Standardise - It forces the data to take a mean of zero and a std of 1. Where z = output data point; x = input data point; m = mean of the dataset; s = datasets standard deviation.

$$z = \frac{x - m}{s}$$

- Why? - you want the different features of input data to have the same weight in the NN.

  For example age and miles drive; as humans are max 100 years old but can drive 100'000 miles, the data is on a different scale.

  It creates instability in the network as the relatively large inputs can cascade down through the layers in the network causing imbalance in the gradients - leading to the **Exploding Gradient Problem**

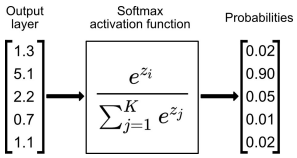
  So performing batch normalisation avoids the EGP and speeds up network training.

### ArgMax

- Normalise and Standardise -

### SoftMax

- probabilities - The softmax function converts a vector or matrix from a set of values to a probability distribution of values from 0 to 1.




### MiniBatch

- Optimization - This method is used to speed up training, instead of performing a training pass on the whole dataset for each epoch, just perform it on a random selection of data. Although the layer gradients are not perfect, they will be a strong representation of the total dataset in some cases.

Listing 1: Python example

```
mini_batch_size = 32
for _ in epoch:
    ix = torch.randint(0, input_data.shape[
    mini_batch_input = input_data[ix]
    model.forward(mini_batch_input)
    etc.
```

### Batch Normalisation

- Normalisation issue - it is only applied to the input layer - but not to the hidden layers
- BatchNorm - BatchNorm accelerates convergence by reducing internal covariate shift inside each batch. If the individual observations in the batch are widely different, the gradient updates will be choppy and take longer to converge.
- Batch Norm Maths - The batch norm layer perform normailisation on the incoming activations and outputs a new batch where the mean equals 0 and standard deviation equals 1. It subtracts the mean and divides by the standard deviation of the batch.

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

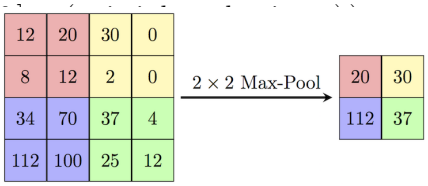
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

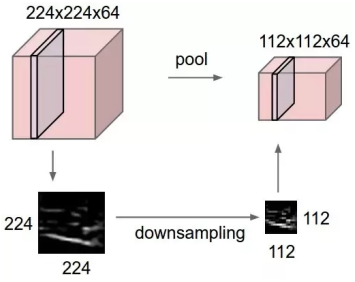
- Benefits of BatchNorm -
  - It enables us to use higher learning rate as the network is more stable.
  - It makes the network less dependant to the initialization strategy.

### Max-pooling

- Introduction - Max pooling is a sample-based discretization process. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.
- Max-Pooling Operation - Max pooling is done by applying a max filter to (usually) non-overlapping subregions of the initial representation.
- Max-Pooling Example - Let's say we have a 4x4 matrix representing our initial input. Let's say, as well, that we have a 2x2 filter that we'll run over our input. We'll have a stride of 2 (meaning the (dx, dy) for stepping over our input will be (2, 2)) and won't overlap regions.

  For each of the regions represented by the filter, we will take the max of that region and create a new, output matrix where each element is the max of a region in the original input.



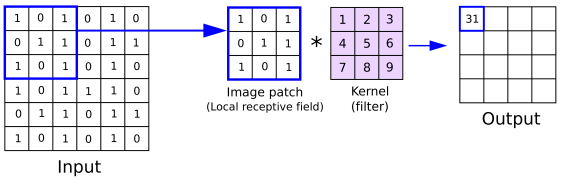



– Benefits of Max-Pooling -

* This is done to in part to help over-fitting by providing an abstracted form of the representation.
* It reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation.

**Convolution**

– convolution is a linear operation that involves multiplication of weight (kernel/filter) with the input




– Kernel (Filter) - A convolution layer can have more than one filter. The size of the filter should be smaller than the size of input dimension. It is intentional as it allows filter to be applied multiple times at difference point (position) on the input.Filters are helpful in understanding and identifying important features from given input. By applying different filters (more than one filter) on the same input helps in extracting different features from given input. Output from multiplying filter with the input gives Two dimensional array. As such, the output array from this operation is called "Feature Map".

– Stride - This property controls the movement of filter over input. when the value is set to 1, then filter moves 1 column at a time over input. When the value is set to 2 then the filer jump 2 columns at a time as filter moves over the input.

https://cs231n.github.io/convolutional-networks/

**LSTM**

– Kernel (Filter) -

– Kernel (Filter) -

**The Attention Mechanism**

– 3 Components -

1. The queries vector **Q** - a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
2. The keys vector **K** - roughly describes what the element is "offering", or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
3. The values vector **V**- This feature vector is the one we want to average over.

1. The queries vector **Q** - How do I make a Margherita Pizza? (but we have a set of questions)
2. The keys vector **K** - Set of Recipe Titles (Attention gives us the probability that a query to a key (question and recipe title) are aligned)
3. The values vector **V**- The actual recipe for a given recipe title (just a lookup from the key)

– The Basic Idea - A query vector will be compared to a set of key vectors to determine how compatible they are. Each key vector comes paired with a value vector, the greater the compatibility of a query vector with a key the greater the influence the corresponding value will have on the output of the attention mechanism. Queries, keys and values are all learned through training.

– The Mechanism -

1. Each word in the sentence is embedded. One embedding for each word.
2. We extract a key and value vector for each one - implemented as linear transformations of the embeddings.
3. We also get a query vector for each word/embedding.
4. We compute a dot product between the query and each key (including the key from the same word). Known as "dot-product attention". $N$ = number of words.

$$\alpha = \{\alpha_1, \alpha_2, \alpha_3 \dots \alpha_N\} = q_i \cdot k$$

5. The resulting $\alpha$ vector is a set of unnormalised weights.
6. To normalise we pass $\alpha$ through softmax.
7. Before softmax, we rescale the weights by dividing by the dimensionality of the query and key vectors, $d_k$.

$$\{\alpha_1, \alpha_2, \alpha_3 \dots \alpha_N\} = \text{softmax}(\frac{\{\alpha_1, \alpha_2, \alpha_3 \dots \alpha_N\}}{\sqrt{d_k}})$$

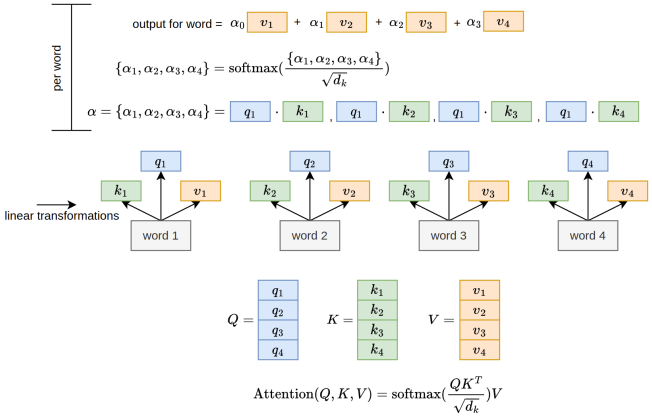
8. we then take a linear combination of the weights, $\alpha$, of the value vectors. Which is the output of the attention mechanism

$$\{\alpha_0 v_1 + \alpha_1 v_2 + \alpha_2 v_3 + \dots + \alpha_{N-1} v_N\}$$

9. We do this for each word in parallel by stacking the vectors into matricies $Q, k$ and $V$. The attention operation can now be defined efficiently as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

10. The output will be another matrix, where each row is an updated representation of each embedded word in the corresponding sequence position.




– Multi-head attention - Only using a single attention head, means that the linear combination of value vectors leads to a "kind of" averaging effect. That limits the resolution of the learned representations.

To alleviate this, we can use multiple attention heads, for $h$ attention heads, we have $h$ sets of learned projection matricies:

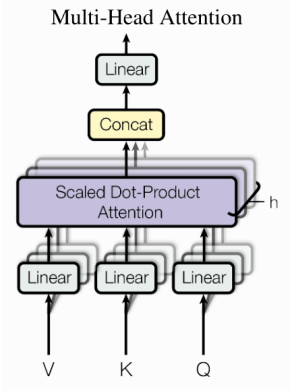
$$W_i^Q \in \mathbb{R}^{D \times d_k}$$
$$W_i^K \in \mathbb{R}^{D \times d_k}$$
$$W_i^V \in \mathbb{R}^{D \times d_v}$$

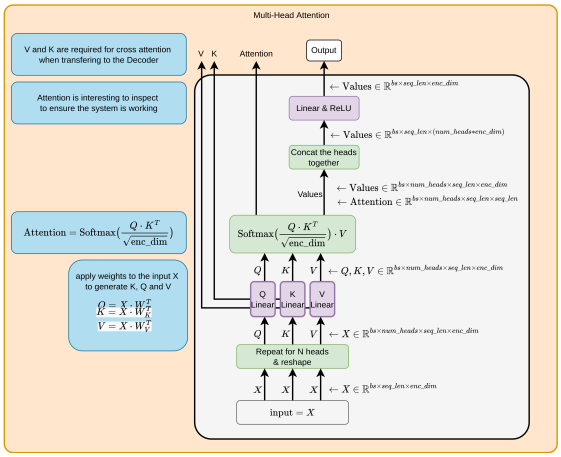
Where $D = d_{\text{model}}$, $i$ indexes over the heads and typically $d_v = d_k = \frac{D}{h}$, $h$ is the number of heads

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^o$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where $W^o \in \mathbb{R}^{D \times D}$ is another weight matrix which linearly projects the learned representations back to the original embedding dimensionality.




## Positional Encoding

– Introduction - Multi-head attention does not process the input sequentially. We can take word order into account through positional encoding. This layer enables non-recurrent models such as transformers to take into account the order of words.

– Learned positional Embedding -

– -

– -

– -

– -

– -

# 3 Maths Tools

## 3.1 Partial Differential Equations

An equation which imposes relations between the various partial derivatives of a multivariate function.

- Total Derivative - of a function $f$ at a point is the best linear approximation near this point of the function with respect to **all** its arguments.
- Partial Derivative - A partial derivative of a function of several variables is its derivative with respect to **one** of those variables, with the others held constant. The partial derivative of a function $f(x, y, ...)$ with respect to the variable $x$ is variously denoted by:

$$f_x, f'_x, \partial_x f, D_x f, D_1 f, \frac{\partial}{\partial x} f or \frac{\partial f}{\partial x} f \tag{23}$$

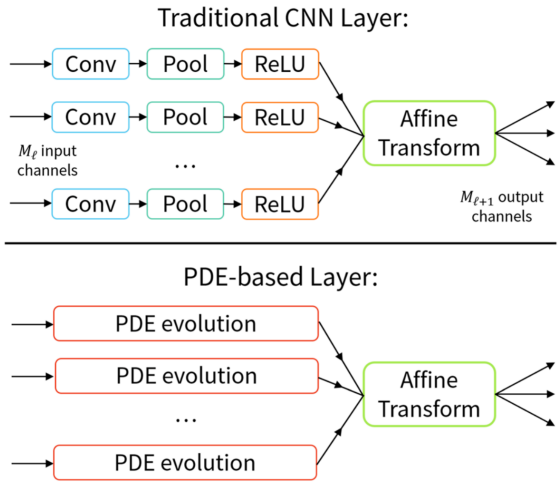
  It can be thought of as the rate of change of the function in the $x$-direction. PDE's are complex to calculate however computers can numerically approximate solutions of certain partial differential equations.

# 4 SOTA - Neural Networks

## 4.1 PDE-based CNN

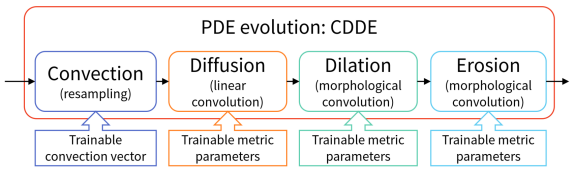
First explored in [2] We present a PDE-based framework that generalizes Group equivariant Convolutional Neural Networks (G-CNNs) shown in 4.2. In this framework, a network layer is seen as a set of PDE-solvers where geometrically meaningful PDE-coefficients become the layer's trainable weights.

– Outline - The key idea is to replace the standard approach of convolution → pooling → ReLUs with a solver for Hamilton-Jacobi type PDE. As shown below:




– PDE Layer - A PDE layer operates by taking its inputs as the initial conditions for a set of evolution equations, hence there will be a PDE associated with each input feature. The idea is that we let each of these evolution equations work on the inputs up to a fixed time $T > 0$. Afterwards, we take these solutions at time $T$ and take affine (linear) combinations (really batch normalized linear combinations in practice) of them to produce the outputs of the layer and as such the initial conditions for the next set of PDEs.

If we index network layers (i.e. the depth of the network) with $l$ and denote the width (i.e. the number of features or channels) at layer $l$ with $M_l$ then we have $M_l$ PDEs and take $M_{l+1}$ linear combinations of their solutions. We divide a PDE layer into the PDE solvers that each apply the PDE evolution to their respective input channel and the affine combination unit. This design is illustrated in Fig. 4.1




The four terms have distinct effects:

* convection: moving data around,
* (fractional) diffusion: regularizing data (which relates to subsampling by destroying data),
* dilation: pooling of data,
* erosion: sharpening of data

---

– PDEs in NNs - To embed a PDE in the neural network layer, we need to describe four components of the PDE:

* its exact form, i.e. $\Delta_{xy}$ - as the differential operator (contains partial differential operators for various interactions between the two dimensions in the input)
* a numerical solver
* initial guess of the solution
* choice of free parameters such as the function $f$

We will go through each of these below:

– PDE Constraint Features - We enforce the following PDE constraint on the output feature map H:

$$\Delta_{xy} H(x, y) = f(I(x, y)) \tag{24}$$

Where f is a function applied on the input feature map. The above operator applies globally on the feature map and does not restrict itself to the local receptive field of operators such as one-layer convolutions.

– 1. Advection-Diffusion PDE - Advection is also called convection. With the equation below, where $H$ is the output feature map:

$$\frac{\partial}{\partial t} H = \nabla \cdot (D \nabla H) + \nabla \cdot (\mathbf{v} H) + f(I) \tag{25}$$

It lets us **treat the input feature map pixels as particles in motion** with velocity v that interact with their neighborhood through diffusion coefficient $D$.

Starting at time $t = 0$ with initial guess of the concentration $H(t = 0)$, the solution of this advection-diffusion equation provides the final particle concentration $H(t = T)$ at time $T$.

The motion of the particles affects the concentration and is modelled by the advection term $\nabla \cdot (vH)$. Similarly, the term $\nabla(D \nabla H)$ describes the diffusion phenomenon, where particles shift between low and high concentrations to reach a steady state.

*Note that both $D$ and $v$ can be a function of the particle locations.*

Finally, the term $f(I)$ is the source of the particle concentration.

In our 2D world, the velocity and diffusion coefficients have two components, i.e. $v = (u, v)$ and $D = (Dx, Dy)$, and the Eq. below boils down to the following form:

$$\begin{aligned} &\frac{\partial}{\partial t} H(x, y, t) + \frac{\partial}{\partial x}(u(x, y, t) H(x, y, t)) + \frac{\partial}{\partial y}(v(x, y, t) H(x, y, t)) \\ &\quad = \frac{\partial}{\partial x}((D_x \frac{\partial}{\partial x} H(x, y, t)) + \frac{\partial}{\partial y}(D_y \frac{\partial}{\partial y} H(x, y, t)) + f(I(x, y)) \end{aligned} \tag{26}$$

– Diffusion Note In these experiments we found no benefit to adding diffusion to the networks. Diffusion likely would be of benefit when the input data is noisy but

neither datasets we used are noisy and we have not yet performed experiments with adding noise. We leave this investigation for future work

– 2. Iterative Solver - For this, we need a simple and efficient PDE solver that can be embedded in the neural network and can achieve approximate solutions easily. y. To obtain a finite element scheme, it is standard in the literature to expand the partial differential operators with their finite-difference elements. Assume the discrete steps for $x$, $y$ and $t$ by $\delta x$, $\delta y$ and $\delta t$ respective. You can discretise equation 26 to be:

$$\begin{aligned} LH_{x,y}^{k+1} = MH_{x,y}^{k-1} - 2(u_x + v_y)\delta_t H_{x,y}^k + 2\delta_t f(I(x,y)) \\ + (-A_x + 2B_x)H_{x+1,y}^k + (A_x + 2B_x)H_{x-1,y}^k \\ + (-A_y + 2B_y)H_{x,y+1}^k + (A_y + 2B_y)H_{x,y-1}^k \end{aligned} \tag{4}$$

where $L = (1 + 2B_x + 2B_y)$, and $M = (1 - 2B_x - 2B_y)$

$$u_x = \frac{u_{x+1,y} - u_{x-1,y}}{2\delta_x}; v_y = \frac{v_{x,y+1} - v_{x,y-1}}{2\delta_y};$$

$$A_x = \frac{u\delta_t}{\delta_x}; A_y = \frac{v\delta_t}{\delta_y}; B_x = \frac{D_x\delta_t}{\delta_x^2}; B_y = \frac{D_y\delta_t}{\delta_y^2};$$

– 3. Initialisation - An initial guess of the solution is crucial for the convergence of the previous recursion. A better initial guess leads to faster convergence. Multiple strategies exist to initialize the output feature map, namely (a) input feature map I, (b) fixed function of the input, and (c) a learnable function of the input ([1] use (c)) - Given an architecture, we use one of its building blocks as the initialization point and learn its parameters during the training stage with back-propagation. They run the PDE for $K$ steps to get the final feature map at time $\frac{K}{\delta t}$

– 4 Choice of the free parameters - There are some free parameters in equation 26. To complete the description of the Global feature layer, we list our parameterization for these free parameters, namely (a) function $f$ (b) particle velocity $(u, v)$, and (c) diffusion coefficient $(Dx, Dy)$. For simplicity, we keep $f$ as identity operator on the input and learn other parameters as the depth-wise convolution over the initialization.

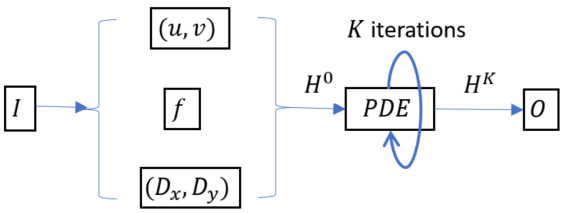
– 5. Boundaries - Ideally, one should carefully design the behavior of the PDE at the boundary. Instead, you can roll the image such that the first particle is a neighbor of the last.

– Implementation - [1] call this PDE layer the "Global feature block" and describe its psudo code as below.

By default, we take the discrete step sizes to be $\delta t = 0.2$, and run the recursions till $K = 5$ steps, resulting in the output state at $T = K\delta t = 1$. We take $\delta x = \delta y = 1$ as the pixel values are not available at any finer details. For all our experiments, free parameters in Eq. 3 are depthwise convolutional operators with the same kernel size as the original block.

**Algorithm 1** Pseudo Code for the Global Feature Block

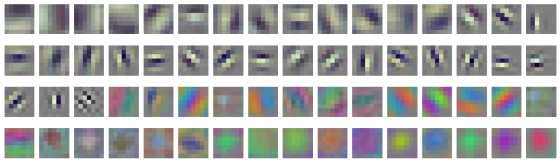
**Input :** Input feature map $I \in \mathbb{R}^{h \times w}$
**Input :** Initial solution guess $F(I)$, Function $f$
**Output :** Output feature map $O$
**Init :** $H^{-1} = H^0 = F(I)$
Compute velocity $(u, v)$, diffusion coefficient $(D_x, D_y)$
**for** $k = 1$ **to** $K$ **do**
  Compute $f(H^{k-1}, I)$
  **for** $x = 1, y = 1$ **to** $h, w$ **do**
    Set $H^{k+1}[x, y]$ as per Eq. 4
  **end for**
  Set output feature map $O = H^K$
**end for**



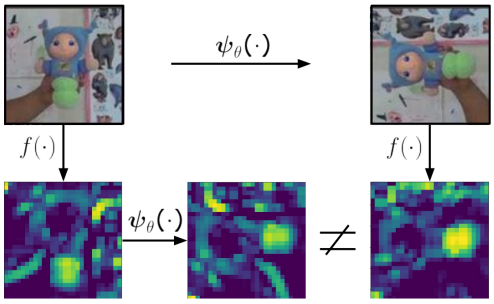
## 4.2 G-CNNs

**Introduction:**

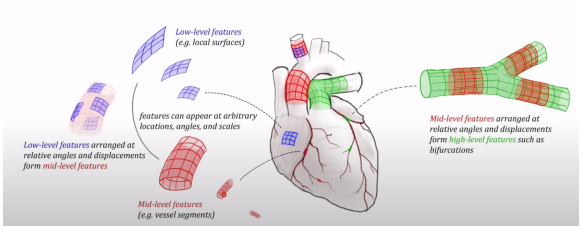
- Motivation for G-CNNs - Neural networks can fail on rotated data or reflected data, to get round these issues, data-augmentation approaches are used. However: (i) these do not guarantee invariance (ii) valuable net capacity is spent on learning invariance (iii) redundancy in feature representations as shown in repeated conv kernels and weights with rotations and symmetry occur in neural networks, as shown in (so much **REDUNDANCY**):



- Inveriance - A models inability to produce the same prediction due to rotational or scale changes in an input image:




- Geometric Guarentees (Equivariance) - Is a property of an operator (such as convolution) that if the input is translated/transformed then the output is transformed in a predictable way - useful because you can handle structure equally well regardless of the pose/transformation/rotation of a feature in an image.
- Importance of Equivariance - (i) No information is lost when the input is transformed and (ii) Guaranteed stability to (local + global) transformations.
- Convolution Equivariance - Traditional convolution layers are equivariant to translation (position changes only). Equivariant Convolution allows for (i) weight sharing and (ii) enables local data analysis.
- Question - But how to have equivariance for functions beyond translation! What about rotation, scaling, reflections etc.
- G-CNNs - (i) Allow for equivariance beyond translation, such as rotation, (ii) Geometric guarantees and (iii) Increased weight sharing.
- Equivariant operators for invariant problems - If the data is typically oriented in a correct manor (people walking will be upright) the core features of the problem will still require equivariant solutions as G-CNNs preserve the structure of the data. For example, peoples arms rotate (Rotation equivariance), they become smaller and larger in images (Scale equivariance), symmetry in the world (left arm vs right arm raised) (Reflection equivariance).
- Psychology of Vision: Recognition of Components - As humans we observe and classify objects with respect to components, for example, identifying a heart:



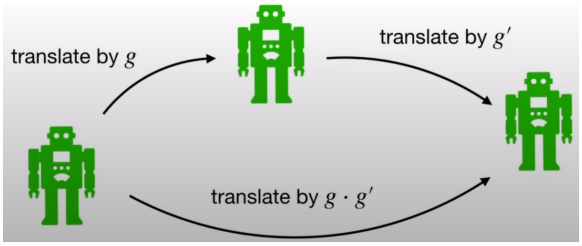

- Symmetry - We observe that incorporating group theory and symmetry into neural networks allows for a more efficient understanding of shape and structure.

**Group Equivariant Deep Learning:**

- What is a Group - A group $(G, \cdot)$ is a set of elements $G$ equipped with a group product $\cdot$, a binarry operator , that satisfies the following axioms:
  1. **Closure**: Given two elements $g$ and $h$ of $G$, the product $g \cdot h$ is also in $G$
  2. **Associativity**: For $g, h, i \in G$ the product $\cdot$ is associative, i.e., $g \cdot (h \cdot i) = (g \cdot h) \cdot i$
  3. **Identity element**: There exists an identity element $e \in G$ such that $e \cdot g = g \cdot e = g$ for any $g \in G$
  4. **Inverse element**: For each $g \in G$ there exists an inverse element $g^{-1} \in G$ s.t. $g^{-1} \cdot g = g \cdot g^{-1} = e$
- Example: Translation Group $(\mathbb{R}^2, +)$ - The translation group consists of all possible translations in $\mathbb{R}^2$ and is equipped with the group product and group inverse:

$$g \cdot g' = (x + x') \quad \text{and} \quad g^{-1} = (-x) \tag{27}$$

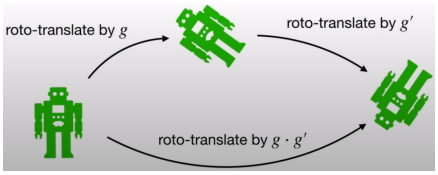
with $g = (\mathbf{x}), g' = (\mathbf{x}.)$ and $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^2$




- Example: Roto-Translation Group $SE(2)$ - (**2**D **S**pecial **E**uclidian motion group) The group $SE(2) = \mathbb{R}^2 \rtimes \mathbf{SO}(2)$ consists of the couples space $\mathbb{R}^2 \times S^1$ of translations vectors in $\mathbb{R}^2$, and rotations in $SO(2)$ 9or equivalently orientations in $S^1$), and is equipped with the group product and group inverse:

$$g \cdot g' = (\mathbf{x}, \mathbf{R}_\theta) \cdot (\mathbf{x}', \mathbf{R}_{\theta'}) = (\mathbf{R}_\theta x' + x, \mathbf{R}_{\theta+\theta})$$

$g^{-1} = (-R_\theta^{-1} x, R_\theta^{-1})$
With $g = (\mathbf{x}, \mathbf{R}_\theta)$, $g' = (\mathbf{x}', \mathbf{R}_\theta)$




Matrix representation: the group can also be represented by matrices:

$$g = (\mathbf{x}, \mathbf{R}_\theta) \leftrightarrow G = \begin{pmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_\theta & x \\ 0^T & 1 \end{pmatrix} \tag{28}$$

With the group product and inverse simply given by the matrix product and matrix inverse simply given by the matrix product and matrix inverse.

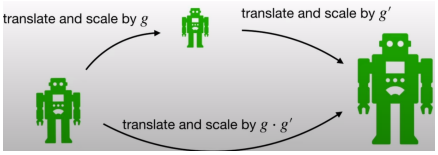
In matrix form:

$$\begin{pmatrix} \mathbf{R}_\theta & x \\ 0^T & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}'_\theta & x' \\ 0^T & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_{\theta+\theta'} & \mathbf{R}_\theta x' + x \\ 0^T & 1 \end{pmatrix} \tag{29}$$

– Scale Translation Group $R^2 \rtimes \mathbb{R}^+$ - The scale-translation group of space $\mathbb{R}^2 \times \mathbb{R}^+$ of translations vectors in $\mathbb{R}^2$ and scale/dilation factors in $\mathbb{R}^2$, and is equipped with the group product and group inverse:

$$g \cdot g' = (\mathbf{x}, s) \cdot (\mathbf{x}', s') = (s\mathbf{x}' + \mathbf{x}, ss')g^{-1} = (-\frac{1}{s}\mathbf{x}, \frac{1}{s}) \tag{30}$$

with $g = (\mathbf{x}, s)$, $g' = (x', s')$




– Affine Groups - all the example groups are examples of Affine groups. They are semi-direct product groups of some group $H$ with an action on $\mathbb{R}^d$ from which we derive the following group product and inverse

$$g \cdot g' = (\mathbf{x}, h) \cdot (\mathbf{x}', h') = (h \cdot \mathbf{x}' + \mathbf{x}, h \cdot h')g^{-1} = (-h^{-1} \cdot \mathbf{x}, h^{-1}) \tag{31}$$

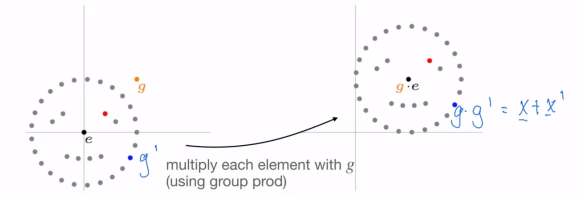
with group elements $g = (\mathbf{x}, h), g' = (\mathbf{x}', h')$ and the terms $h \cdot \mathbf{x}'$ is the **transform point** and the $+$ shift point; The term $h \cdot h'$ is the transform/shift sub-group.

– Translation Group $(\mathbb{R}^2, +)$ - The translation group consists of all possible translations $\mathbb{R}^2$ and is equipped with the **group product** and **group inverse**:

$$g \cdot g' = (\mathbf{x} + \mathbf{x}')$$

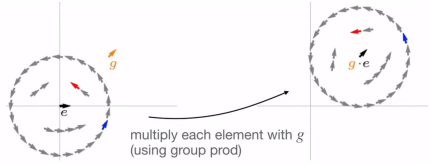
$$\mathrm{g}^{-1} = (-\mathbf{x}) \tag{32}$$

with $g = (\mathbf{x}), g' = (\mathbf{x}')$ and $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^2$

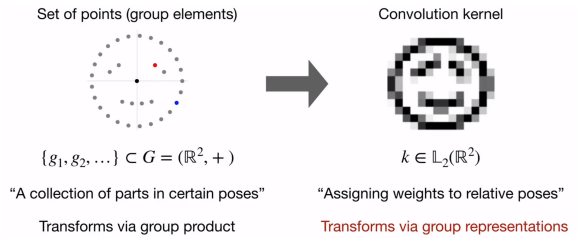



Each point represents a part of the face shown above. We can multiply every point with $g$. If we can describe every point locally (at the origin), we also know what it looks like at any other origin in space.

Likewise we can do the same with the roto-translation group:

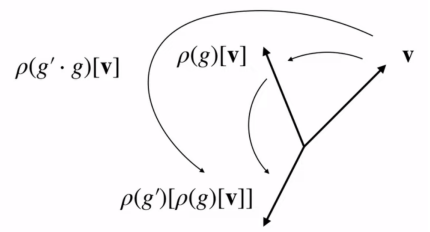



– So... how to translate this to (G-)CNNs? - We can think of collections of group elements as describing some feature or some object made out of lower level parts. But we want to use this knowledge to build G-CNNs. These kernels assign features by assigning weights to relative locations.



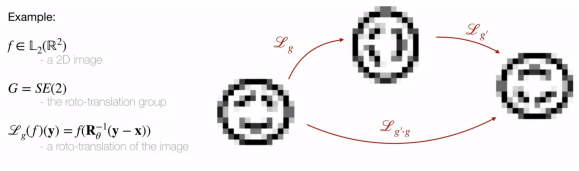

– Representations - A representation $\rho : GL(V)$ is a group homomorphism from G to the general linear group $GL(V)$. That is $\rho(g)$ is a linear transformation that is parameterized by group elements $g \in G$ that transforms some vector $\mathbf{v} \in V$ (e.g. an image) sucvh that

$$\rho(g') \cdot \rho(g)[v] = \rho(g' \cdot g)[v] \tag{33}$$

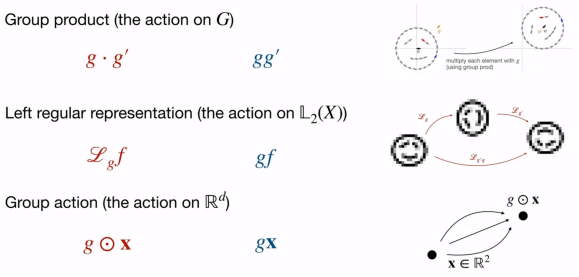



– Left-regular Representations - Is how to transform images! A left-regular representation $\mathcal{L}_g$ is a representation that transforms functions $f$ by transforming their domains via the inverse group action: ($\cdot$ "group action" equals group product when domain is $G$)

$$L_g[f](x) := f(g^{-1} \cdot x)$$

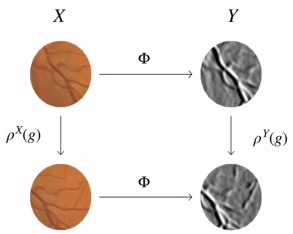



– Group Actions - We can use these different representations to show different forms of the actions and the objects on which they act.




– Equivariance - Is a property of an operator (maybe an NN layer) that maps an element in x to another element in y, so the output space can be different to the input space. Equivariance is a property of an operator $\Phi : x \to Y$ (such as an NN) by which it commutes with the group action:

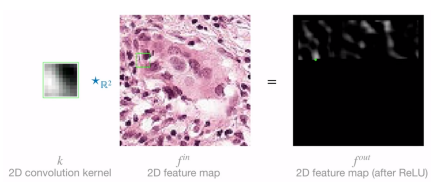
$$\Phi \cdot \rho^X(g) = \rho^Y(g) \cdot \Phi \tag{34}$$



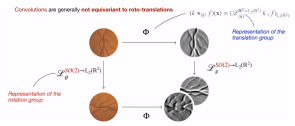

**Regular Group Convolutions:**

– Cross-correlations - They are convolutions with reflected conv kernels (and vice versa). The below equations is the equation for convolution. The term $L_g$ is the Representation of the translation group! and $k$ is the convolution kernel

$$(k \star_{\mathbb{R}^2} f)(\mathbf{x}) = \int_{\mathbb{R}^2} k(\mathbf{x}' - \mathbf{x}) f(\mathbf{x}') d\mathbf{x}' = (L_g k, f)_{\mathbb{L}_2(\mathbb{R}^2)}$$



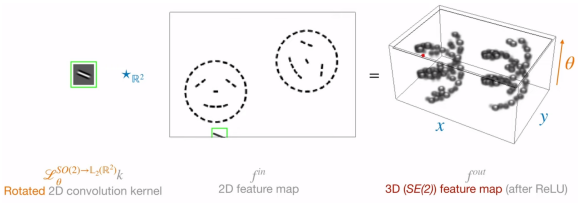

– Equivariance -

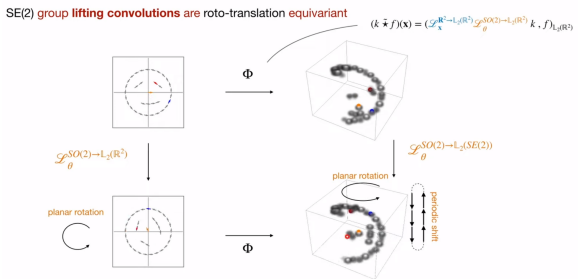


– SE(2) Equivariant cross-correlations (roto-translation equivariant) - Lifting correlations: $(k\tilde{\star}f)(\mathbf{x}, \theta) = (L_g^{SE(2)\to\mathbb{L}_2(\mathbb{R}^2)} k, f)_{\mathbb{L}_2(\mathbb{R}^2)}$ Where the term $L_g^{SE(2)\to\mathbb{L}_2(\mathbb{R}^2)}$ is a representation of the roto-translation group! So we can split the term into a translation and rotation term respectively:

$$L_g^{SE(2)\to\mathbb{L}_2(\mathbb{R}^2)} = L_{\mathbf{x}}^{\mathbf{R}^2\to\mathbb{L}_2(\mathbb{R}^2)} L_{\mathbf{y}}^{SO(2)\to\mathbb{L}_2(\mathbb{R}^2)} = R_\theta^{-1}(x' - x))$$

we can rotate the convolutional kernel through $\theta$ to produce a 3D representation of the image, in the case below the convolutional kernel is rotated and passed over the image, to produce the 3D structure shown:



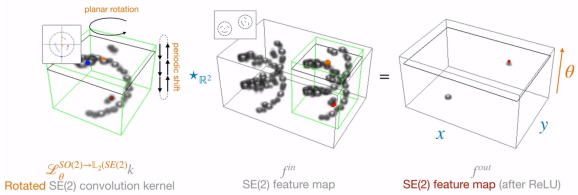

– SE(2) G-correlations (roto-translation equivariant) - So if we perform the step above on our 2D input image, we get a 3D image as output. We now want to perform cross correlations on this outputted 3D image (layered convolution). The equation for such is show below. Group correlations:

$$(k \star f)(\mathbf{x}, \theta) = (L_g^{SE(2)\to\mathbb{L}_2(SE(2))} k, f)_{\mathbb{L}_2(SE(2))}$$

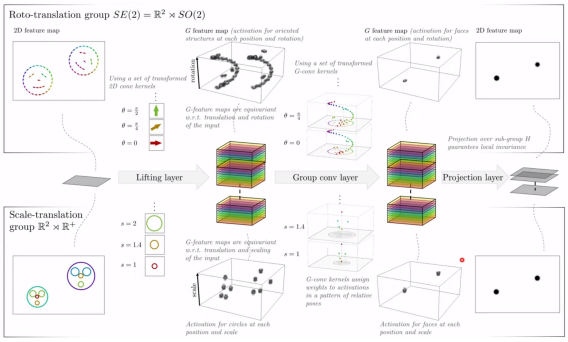
$$= (L_{\mathbf{x}}^{\mathbf{R}^2\to\mathbb{L}_2(SE(2))} L_{\mathbf{y}}^{SO(2)\to\mathbb{L}_2(SE(2))} k, f)_{\mathbb{L}_2(SE(2))}$$

where:

$$L_{\mathbf{x}}^{\mathbf{R}^2\to\mathbb{L}_2(SE(2))} L_{\mathbf{y}}^{SO(2)\to\mathbb{L}_2(SE(2))} k = k(R_\theta^{-1}(x' - x), \mathbf{R}_{\theta' - \theta})$$

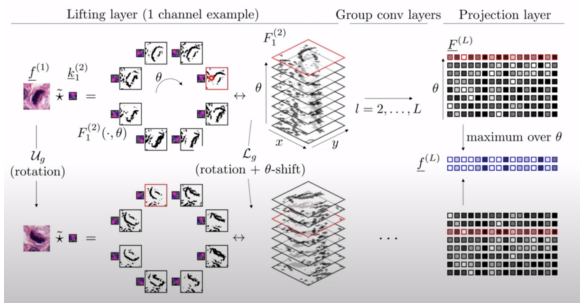


– G-CNN Use - Below is the basic structure of a G-CNN, used for face detection:

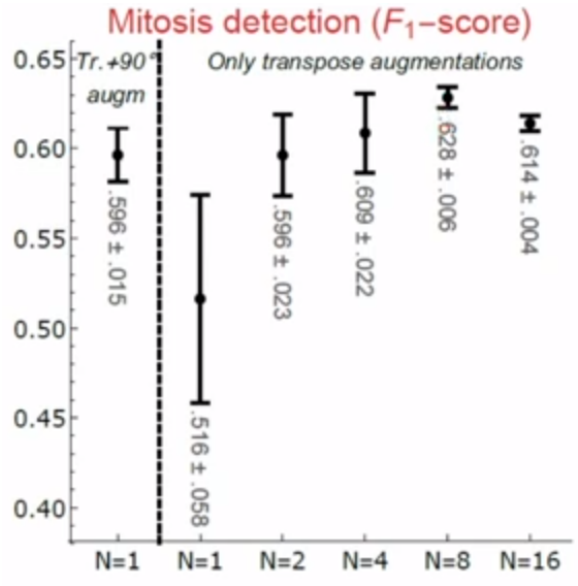



**Basic Application and Findings:**

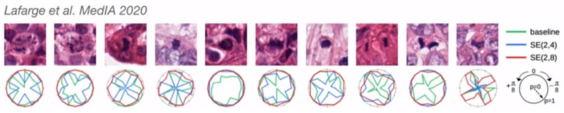
– Architecture for rotation invariant mitotic cell detection - Applying the conv kernel $k_1^{(2)}$ (with the conv2D operator) at different rotations, we get a set of transformed images $F_1^{(2)}$, we can then apply a group conv layer (without padding on the spacial axis) to convert this batch into a single point for each of the possible rotations $\theta$, we can then pool over the extra rotation axis $\theta$ to get a feature vector of values representing this entire patch. As you see the rotated image has the same output feature layer!




– Performance of Mitosis Detection - comparison to a model using data augmentation methods. For the G-CNN models, $N$ is the number of rotations. As you can see, the G-CNN out performs the standard approach (CNN + data augmentation). All models have the exact same amount of parameters, so to keep to this, the $N = 16$ model required so many more weights for the group convolutional kernals for the extra number of rotations that they had to reduced the number of channels encoded, which had a negative impact on task performance.




Here they show the prediction based on rotations of the same image - showing that SE(2,8), is more capable of correctly classifying the rotated images (prediction at the center point is healthy, prediction at outer point is a mitotic flag).




**Steerable Group Convolutions**

– Steerable Basis - A vector $Y(x) = \begin{pmatrix} \vdots \\ Y_l(x) \\ \vdots \end{pmatrix} \in \mathbb{K}^L$ with basis functions $Y_l \in \mathbb{L}_2(X)$ is steerable if

$$\forall_{g \in G}: \quad Y(gx) = \rho(g)Y(x) \tag{35}$$

Where $gx$ denotes the action of $G$ on $X$ and $\rho(g) \in \mathbb{K}^{L \times L}$ is a representation of $G$

i.e we can transform all basis function simply by taking a linear combination of the original basis functions.

–

–

–

Notes from Eric Bekkers:

### 4.3 EfficientNetB0-B7

https://arxiv.org/pdf/1905.11946.pdf –OG paper

Used in RT-1 (https://ai.googleblog.com/2022/12/rt-1-robotics-transformer-for-real.html: :text=Theand https://robotics-transformer.github.io/assets/rt1.pdf

Efficient ConvNN networks (EfficientNet B0 to B7) - B7 being the largest but best performing

## 4.4 Robot Transformer 1 (RT-1)

Found here:

https://robotics-transformer.github.io/assets/rt1.pdf

https://ai.googleblog.com/2022/12/rt-1-robotics-transformer-for-real.html: :text=The

# 5 Literature review

## 5.1 Video Prediction

**Temporally Consistent Video Transformer for Long-Term Video Prediction**

# 6 Memorisation Prompts

## 6.1 Machine Learning

——— **Linear Algebra and Calculus** ———

- **General Notation:**
  - Vectors
  - Matrix
  - Identity Matrix
  - Diagonal Matrix
- **Matrix Operations:**
  - Vector x Vector
  - Matrix x Vector
  - Matrix x Matrix
  - Transpose
  - Inverse
  - Trace
  - Determinant
- **Matrix Properties:**
  - Symmetric Decomposition
  - Norm
  - Linearly Dependence
  - Matrix Rank
  - Positive semi-definate Matrix
  - Eigenvalue and Eigenvector
  - Diagonalisation
  - Spectral Theorem

——— **Probabilities and Statistics** ———

- **Basics & Combinations**
  - Sample Space
  - Event
  - Axioms of Probability
  - Permutation and Combination
- **Conditional Probabilities**
  - Bayes Rule
  - Partition
  - Extended Bayes Rule
- **Random Variables**
  - Random Variable
  - Cumulative Distribution Function

——— **Deep Learning** ———

- **Neural Networks**
  - Architecture
  - Activation Function
  - Cross Entropy Loss
  - Learning Rate
  - Back Propagation
  - Updating the weights
- **Convolutional Neural Networks**
  - Conv Layer Requirements
  - Batch Normalisation
- **Recurrent Neural Networks**
  - Types of gates
  - LSTM
- **Reinforcement Learning and Control**
  - Markoc Decision Process
  - Policy
  - Value Function
  - Bellman Equation
  - Value Iteration Algorithm
  - Maximum Liklihood Estimation
  - Q-Learning

——— **Supervised Learning** ———

- **Introduction**
  - Type of Prediction
  - Types of Models
- **Notation and General Concepts**
  - Hypothesis
  - Loss Function
  - Cost Function
  - Gradient Descent
  - Stochastic Gradient Descent
  - Liklihood
  - Newtons Algorithm
- **Linear Regression**
  - Normal Equations
  - Least Mean Squares algorithm
  - Locally Weighted Regression
- **Classification and Logistic Regression**
  - Sigmoid Function
  - Logistic Regression
  - Softmax Regression
- **Generalize Linear Models**
  - Exponential Family
  - Assumptions of Generalised Linear Models
- **Support Vector Machines**
  - Optimal Margin Classifier

## 6.2 Deep Learning

——— **Recurrent Neural Networks** ———

- **Overview:**
  - Architecture of a traditional RNN
  - Typical RNN pros and cons
  - RNN application
  - Loss Function
  - Back Propagation
- **Long Term Dependencies:**
  - Common Activation Functions
  - Vanishing/Exploding Gradient
  - Gradient Clipping
  - Types of Gates
  - GRU/LSTM
  - RNN variants
- **Long Term dependencies:**
  - Learning word Representations
  - Learning word Representations
  - Word2vec
  - Skip-gram
  - Negative Sampling
  - GloVe
- **Comparing Words:**
  - Cosine Simiarity
  - t-SNE
- **Language Models:**
  - Overview
  - n-gram models
  - Perplexity
- **Machine Translation:**
  - Overview
  - Beam Search
  - Beam Width
  - Error Analysis
  - Bleu score
- **Attention:**
  - Attention Model
  - Attention Weight
- **Overview:**
  -

## 6.3 SOTA - Neural Networks

——— **G-CNNs** ———

- **Introduction:**
  - Motivation for G-CNNs
  - Inveriance
  - Geometric Guarentees (Equivariance)
  - Importance of Equivariance
  - Convolution Equivariance
  - Question
  - G-CNNs
  - Equivariant operators for invariant problems
  - Psychology of Vision: Recognition of Components
  - Symmetry
- **Group Equivariant Deep Learning:**
  - Geometric Guarantees
  - Inveriance
  - Geometric Guarentees (Equivariance)
  - Importance of Equivariance
  - Convolution Equivariance
  - Question
  - G-CNNs
  - Equivariant operators for invariant problems
  - Psychology of Vision: Recognition of Components
  - Symmetry
  - What is a Group
  - Example: Translation Group $(\mathbb{R}^2, +)$
  - Example: Roto-Translation Group $SE(2)$
  - Scale Translation Group $R^2 \times |\mathbb{R}^+$
  - Affine Groups
  - Translation Group $(\mathbb{R}^2, +)$
  - So... how to translate this to (G-)CNNs?
  - Representations
  - Left-regular Representations
  - Group Actions
  - Equivariance
- **Regular Group Convolutions:**
  - Cross-correlations
  - Equivariance
  - SE(2) Equivariant cross-correlations (roto-translation equivariant)
  - SE(2) G-correlations (roto-translation equivariant)
  - G-CNN Use
- **Basic Application and Findings:**
  - Architecture for rotation invariant mitotic cell detection
  - Performance of Mitosis Detection
  -
  -
  -
  -

# References


[1] Anil Kag and Venkatesh Saligrama. "Condensing CNNs with Partial Differential Equations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 610–619.

[2] Bart Smets et al. "PDE-based group equivariant convolutional neural networks". In: *Journal of Mathematical Imaging and Vision* (2022), pp. 1–31.