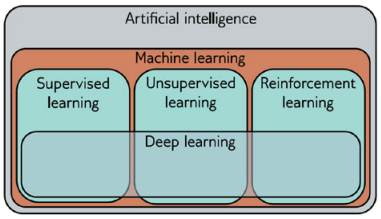
# Artificial Intelligence

AI is the science of solving complex problems and the simulation/ mimicking of intelligent behaviour with a computer, without being explicitly programmed. The automation of things and activities that we associate with human thinking and behaviour.

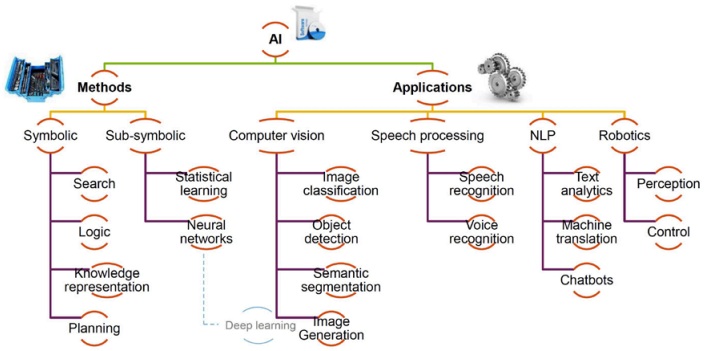
AI = Probabilistic tech, ML = Specific methods, Metrics: Time, size, compute




**Supervised:** The goal is to learn a function that maps input features to output labels.

**Unsupervised:** The goal is to model the underlying distribution of the data. **generative models.**

**Reinforcement Learning:** The goal is to learn a policy that maps states to actions.

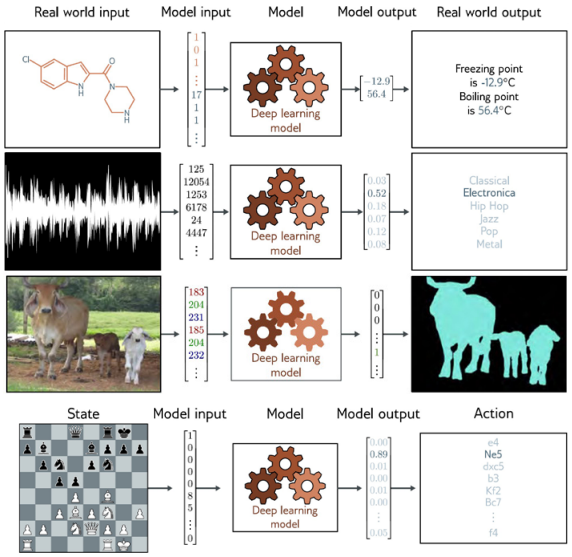



Applications use Methods. NLP = Natural Language Processing

## Basics

**Model:** Function that maps input to ouput = mapping of things. A family of patameterized functions for **regression and classification** problems.

Examples:




## Latent Space / Laten Variables

Latent means hidden or underlying features. Refers to a compressed, abstract and lower-dimensional representation of data where the model encodes the features in a vector it has learned.

**Interpolation and Manipulation:** You can smoothly transition between concepts by moving through latent space (e.g., morphing images)

**Efficiency:** Working in latent space is computationally cheaper than working with raw high-dimensional data

**Generalization:** The structure of latent space often reveals meaningful relationships the model has learned

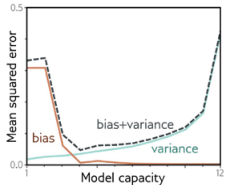
## Performance

**Accuracy:** Proportion of samples that are correctly classified. $\text{accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$

**Recall:** High recall means many correct classifications but may have many false positive. $\text{recall} = \frac{TP}{TP+FN}$

**Precision:** High precision means most classifications are correct but may have many misses. $\text{precision} = \frac{TP}{TP+FP}$

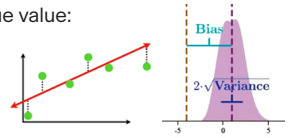
## Bias-Variance Trade-Off




Bias and variance is a function of model capacity.

**Bias:** Systematic error, due to an under-complex model, the inhability to capture the true relationship.
how far the estimate is on average from the true value:

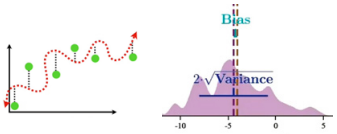
$$\text{Bias}(\hat{X}) = \mathbb{E}[\hat{X}] - X$$




High bias → systematically wrong estimation
Low bias → correct on average

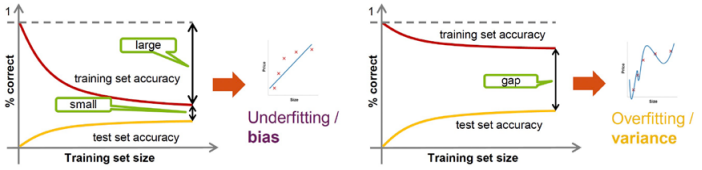
**Variance:** Dependence on specific samples, measures how much the estimate changes with different samples.
Shows the sensitifity of the model to the training data.
High variance means high sensitivity to small input changes.

$$\text{Var}(\hat{X}) = \mathbb{E}\left[(\hat{X} - \mathbb{E}[\hat{X}])^2\right]$$




High variance → unstable estimates
Low variance → consistent estimates

You want low bias and low variance, but reducing one usually increases the other.




**High bias,** low variance → **underfitting**
Low bias, **high variance** → **overfitting** (high sensitivity)
low bias and low variance → good generalization

### Error Decomposition (only for MSE)

$$E_{MSE} = \underbrace{\text{systematic error}}_{\text{Bias: average prediction's deviation from the truth}} + \underbrace{\text{dependence on specific sample}}_{\text{Variance: sensitivity of prediction to specific training sample}} + \underbrace{\text{random nature of process}}_{\text{Irreducible error or Bayes' rate: due to "noise"}}$$

**Fix Overfitting** (high varianve): Get more training examples, Try smaller sets of features, Try more regularization, Build ensembles.

**Fix Underfitting** (high bias): Try getting additional features, Try adding polynomial features, Try less regularization. Increase model capacity.

## Inductive Bias

Refers to the set of **assumptions** a machine learning model makes that **allows it to generalize** from limited training data to unseen examples. Its about assumptions that guide learning/generalization. It's the model's prior beliefs about what kinds of patterns are likely to exist in the world. Examples:

**Convolutional Neural Networks (CNNs):** Assume spatial locality and translation invariance, that nearby pixels are related

**Recurrent Neural Networks (RNNs):** Assume sequential dependencies, that order matters and past information influences future predictions.

**Decision Trees:** Assume the data can be split into hierarchical regions with axis-aligned boundaries

**Transformers:** Assume attention mechanisms can capture relevant relationships, with less built-in bias about structure (more flexible but requires more data)

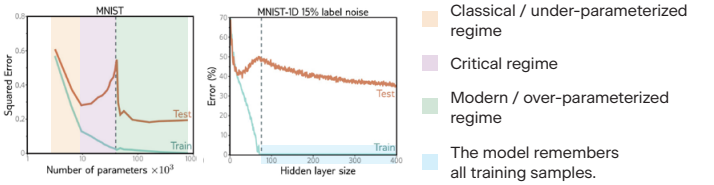
**Difference to Ontological Fit: Inductive Bias:** Focuses on the *how to learn* Assumptions guiding generalization. **Ontological Fit:** Focuses on *what exists.* Does the model's structure match reality?

## Varia

**ML Development Process:** ML development is an empirical science. Experiment, do explorative data analysis, rapid prototyping, verify decision experimentally, build ensembles / bagging of finaml model, provide enough training data.
**No Free Lunch:** There's no universally best model across all problems.
**Curse of Dimensionality:** Irrelevant features do harm.
More dimension are costly. Training data needs to grow exponentially.
**Choosing Model Capacity:** Empirically. Model with high capacity may lead to overfitting. But we can use tuning (hyperparameter optimization algorithms) to do systematic sampling, e.g. Bayesian Optimization, Genetic algorithms, etc.
**Invariance:** The quality of remaining unchanged, regardless of changes in conditions or transformations. E.g. the CNN model predicts the same class, regardless of the position of the object.
**Variance:** Sensitivity to small input changes
**Cross-Validation:** Typically we divide data into: training (to learn the model parameters), validation (to choose the hyperparameters) and test data (to estimate the final performance). But when data is limited (if the number of training examples is comparable to the model capacity), then the variance will be large. solves the problem of reliable model evaluation and hyperparameter selection when you have limited data. The hyperparameters are based on the average validation performance.
**Ockham's Razor:** A principle of simplicity. States that, among equally effective explanations or models, the simplest one should be preferred.
**Ontological Fit:** Its about correct representation of the domain. A model works well when its internal representation matches the actual entities, relationships, and causal structure of the domain. Determines if the model can represent reality accurately. Ontological Fit is about the right kind of model structure. Correct ontology → lower bias without increasing variance.
**Sparse vs Coarse: Sparse:** Data where most values are zero or absent. Opposite: Dense (most values are non-zero). **Coarse:** Low resolution or lack of fine detail, few discrete levels. E.g. bounding box (= coarse) instead of per pixel (= dense, semantic segmentation).
**Intelligence:** The ability to acquire, understand, use knowledge, and to solve problems (this requires to recognise the problem in the first place).
**Hypothesis Space:** All functions that can be represented by a model. hypothesis = candidate function that maps inputs to outputs. Example: The Hypothesis space of a Linear Regressionare are all possible straight lines.
**Inductive Learning:** When a model infers a general function from training data that can predict outcomes for unseen data. The goal is generalization. *Inductive* refers to reasoning from specific examples to general rules. Searching the hypothesis space is guided by inductive bias.

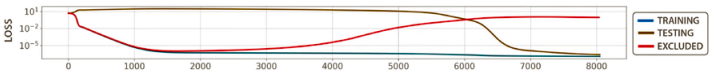
## Double Descend

When the test error does not show the expected bias-variance trade-off but continues to decrease even after the model has memorized the dataset.



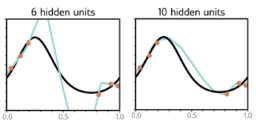

Over-parameterization makes Double Descend possible.
The model remembers all training samples.

**Softmax Collapse:** When both metrics are flat for prolonged time, the assumption is, that the model is done learning and has settled into a stable solution. But, because of floating-point precision in Softmax, gradients become 0, stopping the learning process (no gradient, no learning).Then it increases the magnitude of the logits (input to the softmax function), so gradients become non-zero again. This hidden metric is called **Excluded Loss** (*Progress Measures for Grokking Via Mechanistic Interpretability*, 2023). This happens before grokking takes place.




Softmax Collapse is key obstacle to generalization, and that overcoming it is a major step to allowing grokking to happen. (*Grokking at the Edge of Numercial Stability*, 2025)

Adding capacity and implicit regularization (weight initialization and training algorithm) leads to smoother interpolation between data points.



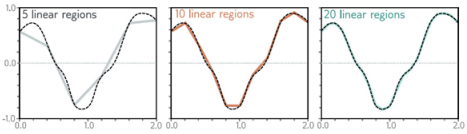

**Grokking:** When a model generalizes after overfitting (memorizing) the training set.

# Deep Learning

## Universality Approximation Theorem (Hornik, 1991)

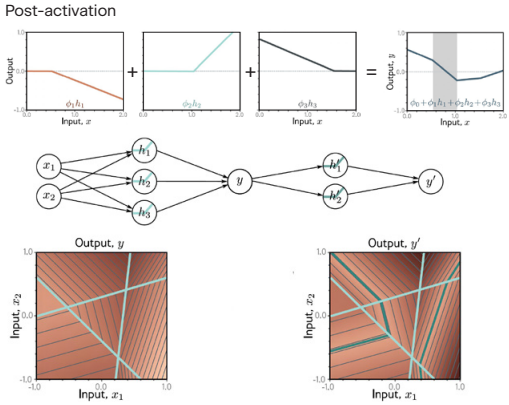
**A neural net is a (parametrizable) general function approximation.**

With enough hidden units, there exists a shallow neural network (shallow neural nets have only one hidden layer) that can approximate any continuous function to arbitrary precision. SVM, Decision Trees, etc. have their limitations. Neural nets are **piecewise linear functions.**




## Depth Efficiency

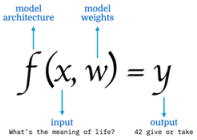
The number of linear regions increases with network depth, which means more capacity with less parameters. Shallow nets have an upper region bound of x·n + 1 (n = number of hidden neurons, x = number of input layers).



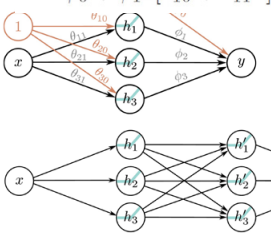

## Notation

$\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$

$\mathbf{y}$ = Output values
$\mathbf{x}$ = Input values
$\boldsymbol{\phi}$ = (phi) Parameters (weights and biases)
$\mathbf{f}$ = Neural Network can be seen as a function
$\mathrm{a}$ = Activation function




$$= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x] + \ldots$$



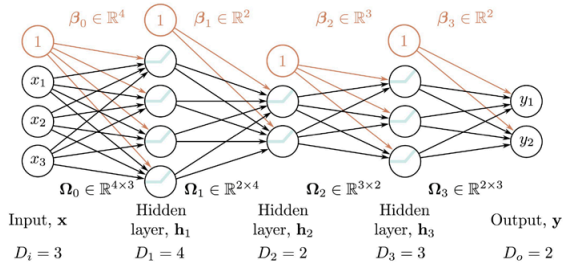
$n$ = Neuron (post-activation)
$\mathrm{a}$ = Activation function
$\theta, \phi, \psi$ = Weight or bias (theta, phi, psi)



$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \end{aligned} \qquad \begin{aligned} h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3] \end{aligned}$$

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$

## General Formulation




$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\ &\vdots \\ \mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\ \mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K. \\ &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{a}\left[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{a}\left[\ldots\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{a}\left[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}\right]\right]\ldots\right]\right] \end{aligned}$$

$\mathbf{h}$ = Layer
$\mathbf{a}$ = Activation function
$\boldsymbol{\beta}$ = Biases (Beta)
$\boldsymbol{\Omega}$ = Weights (Omega)

**Model Capacity** = all weights and biases

## Training

Training is about **finding the network parameters** $\phi$ that **minimize the negative log-likelihood loss function** ***L*** over the training dataset pairs (*xi*, *yi*)

1. Get a batch from the data loader
2. Forward: Obtain the predictions from the model for the batch
3. Calculate the loss based on the difference between predictions and labels
4. Backpropagation: calculate the gradients for every parameter with respect to the loss
5. Update the parameters of the model in the direction of the gradients

```
def train_model(model, optimizer, data_loader, loss_module, num_epochs=9):
  # Set model to train mode. Some models might perfom different
  # foward steps during training than turing testing
  model.train()
  # Training loop
  for epoch in tqdm(range(num_epochs)):
    for data_inputs, data_labels in data_loader:
       # Move input data to device (e.g. gpu)
       data_inputs = data_inputs.to(device)
       data_labels = data_labels.to(device)
       # Step 1: Forward pass. Run the model on the input data
       preds = model(data_inputs)
       preds = preds.squeeze(dim=1) # Output is [Batch, 1], we want [Batch]
       # Step 2: Calculate the loss
       loss = loss_module(preds, data_labels.float())
       # Step 3: Perform backpropagation
       # zero_grad() clears old gradients from the last step
       # This is important because gradients by default adds up;
       # we don't want to mix up gradients between mini-batches
       optimizer.zero_grad()
       # Perform backpropagation
       loss.backward()
       # Step 4: Update the parameters (weights)
       optimizer.step()
# Finaly, train the model
train_model(model, optimizer, train_data_loader, loss_module)
```

## Loss Functions *L*

Sometimes called Cost Function.

A loss function *L*[ϕ] returns a single number describing the mismatch between the model predictions and corresponding ground-truth.

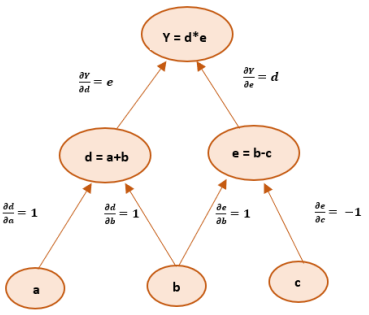
### Distributions for differents Output Types

| Data Type | Domain | Distribution | Use | |
|---|---|---|---|---|
| univariate, continuous, unbounded | $y \in \mathbb{R}$ | univariate normal | regression | MSE |
| univariate, continuous, unbounded | $y \in \mathbb{R}$ | Laplace or t-distribution | robust regression | |
| univariate, continuous, unbounded | $y \in \mathbb{R}$ | mixture of Gaussians | multimodal regression | |
| univariate, continuous, bounded below | $y \in \mathbb{R}^+$ | exponential or gamma | predicting magnitude | |
| univariate, continuous, bounded | $y \in [0, 1]$ | beta | predicting proportions | |
| multivariate, continuous, unbounded | $\mathbf{y} \in \mathbb{R}^K$ | multivariate normal | multivariate regression | |
| univariate, continuous, circular | $y \in (-\pi, \pi]$ | von Mises | predicting direction | |
| univariate, discrete, binary | $y \in \{0, 1\}$ | Bernoulli | binary classification | Cross-entropy |
| univariate, discrete, bounded | $y \in \{1, 2, \ldots, K\}$ | categorical | multiclass classification | Multiclass Cross-entropy |
| univariate, discrete, bounded below | $y \in \{0, 1, 2, 3, \ldots\}$ | Poisson | predicting event counts | |
| multivariate, discrete, permutation | $\mathbf{y} \in \text{Perm}[1, 2, \ldots, K]$ | Plackett-Luce | ranking | |

## Computational Graph

Visual representation of mathematical operations as a directed graph. The graph shows the computation of gradients using the chain rule.

Example: Y = (a+b) · (b-c)




## Maximum Likelihood

Is a way to **find the parameters** of a model (e.g. linear regression) that make the observed data most probable (likely to observe).

Model fitting: Finding the parameter that best describes the data's distribution.

**Identical Identity Distribution Assumption:** We assume the data is independent and identically distributed. When events are independent, the probability of them all happening together is the product of their individual probabilities:

$$Pr(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_I | \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_I) = \prod_{i=1}^{I} Pr(\mathbf{y}_i | \mathbf{x}_i)$$

We want to minimize the **negative log-likelihood.** We dont use *arg**max**[Pr(y|f)]* because of numerical stability, and because multiplying something by 0 ruins everything.

$$\begin{aligned}
\hat{\phi} &= \underset{\phi}{\text{argmax}} \left[ \prod_{i=1}^{I} Pr(\mathbf{y}_i | \mathbf{f}[\mathbf{x}_i, \phi]) \right] \quad \leftarrow \textbf{Maximum Likelihood Criterion} \\
&= \underset{\phi}{\text{argmax}} \left[ \log \left[ \prod_{i=1}^{I} Pr(\mathbf{y}_i | \mathbf{f}[\mathbf{x}_i, \phi]) \right] \right] \\
&= \underset{\phi}{\text{argmin}} \left[ -\sum_{i=1}^{I} \log \left[ Pr(\mathbf{y}_i | \mathbf{f}[\mathbf{x}_i, \phi]) \right] \right] = \underset{\phi}{\text{argmin}} \left[ L[\phi] \right]
\end{aligned}$$

*I* or *N* = training examples

**Inference** = maximum of the distribution. Our network no longer directly predicts outputs *y* but a probability distribution over *y*. But we want a point estimate rather than a distribution, so we return the maximum of the distribution = mean of gaussian distribution

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\text{argmax}} \left[ Pr(\mathbf{y} | \mathbf{f}[\mathbf{x}, \hat{\phi}]) \right]$$

## Designing a Loss Function

### Binary Classification (Binary Cross Entropy, BCE)

Negative log-likelihood BCE is a bernoulli distribution.

$$Pr(y|\lambda) = (1-\lambda)^{1-y} \cdot \lambda^y$$

$$Pr(y|\mathbf{x}) = (1 - \text{sig}[\text{f}[\mathbf{x}, \phi]])^{1-y} \cdot \text{sig}[\text{f}[\mathbf{x}, \phi]]^y$$

$$P(y|x) = (1 - f(x))^{1-y} \cdot f(x)^y$$

$\log[P(y|x)] = \log[(1 - f(x))^{1-y} \cdot f(x)^y]$ Take the logarithm

$\log[P(y|x)] = (1-y)\log(1 - f(x)) + y \log(f(x))$ Using logarithm properties:

$$L[\phi] = \sum_{i=1}^{I} -(1 - y_i) \log \left[ 1 - \text{sig}[\text{f}[\mathbf{x}_i, \phi]] \right] - y_i \log \left[ \text{sig}[\text{f}[\mathbf{x}_i, \phi]] \right]$$

y = labels, x = inputs, sig = sigmoid

The **sigmoid** is used to constrain the linear output to 0 and 1, giving us valid probability values.

Its called Cross Entropy because maximizing likelihood (i.e., negative log-likelihood criterion) and mimimizing distance between model and observed data (i.e., cross-entropy criterion) are equivalent.

### Multiclass Classification

$$Pr(y = k|\mathbf{x}) = \text{softmax}_k \left[ \text{f}[\mathbf{x}, \phi] \right]$$

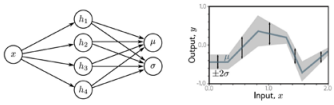
$$\begin{aligned}
L[\phi] &= -\sum_{i=1}^{I} \log \left[ \text{softmax}_{y_i} \left[ \text{f}[\mathbf{x}_i, \phi] \right] \right] \\
&= -\sum_{i=1}^{I} \left( \text{f}_{y_i}[\mathbf{x}_i, \phi] - \log \left[ \sum_{k'=1}^{K} \exp[\text{f}_{k'}[\mathbf{x}_i, \phi]] \right] \right)
\end{aligned}$$

where $\text{f}_{y_i}[\mathbf{x}, \phi]$ and $\text{f}_{k'}[\mathbf{x}, \phi]$ denote the $y_i^{th}$ and $k'^{th}$ outputs of the network

The transformed model output represents a categorical distribution over possible classes y.

### Heteroscedasic Regression

Predicts mean $\mu$ and variance $\sigma$.



$$\hat{\phi}, \hat{\sigma}^2 = \underset{\phi,\sigma^2}{\operatorname{argmin}} \left[ -\sum_{i=1}^{I} \log \left[ \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left[ -\frac{(y_i - \mathrm{f}[\mathbf{x}_i, \phi])^2}{2\sigma^2} \right] \right] \right]$$

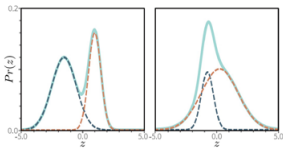
### Univariate Linear Regression

Least squares resp. Mean Squared Error (MSE)

$$L[\phi] = \sum_{i=1}^{I} \big(y_i - \mathrm{f}[\mathbf{x}_i, \phi]\big)^2 \qquad \hat{\phi} = \underset{\phi}{\operatorname{argmin}} \left[ \sum_{i=1}^{I} (y_i - \mathrm{f}[\mathbf{x}_i, \phi])^2 \right]$$

### Multimodal Distribution

If there is more than one valid prediction for a given input → weighted sum of normal distributions = mixture of Gaussians. Example, a mixture of two Gaussians $\boldsymbol{\theta} = \{\lambda, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2\}$:



$$Pr(y|\lambda, \mu_1, \mu_2, \sigma_1^2, \sigma_2^2) = \frac{\lambda}{\sqrt{2\pi\sigma_1^2}} \exp \left[ \frac{-(y-\mu_1)^2}{2\sigma_1^2} \right] + \frac{1-\lambda}{\sqrt{2\pi\sigma_2^2}} \exp \left[ \frac{-(y-\mu_2)^2}{2\sigma_2^2} \right]$$

## Evaluation

$$acc = \frac{\#\text{correct predictions}}{\#\text{all predictions}} = \frac{TP + TN}{TP + TN + FP + FN}$$

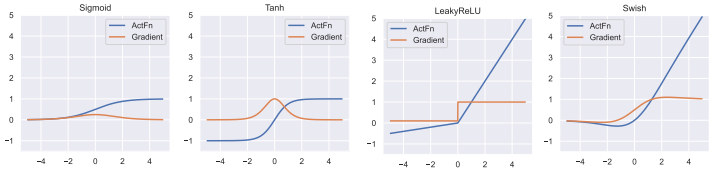
TP = True positives, TN = True negatives
FP = False positives, FN = False negatives

## Training using pre-trained Models

Pre-trained models have already learned useful features and patterns from large datasets. Instead of starting from scratch, you're building on top of knowledge the model has already acquired. By starting with a pre-trained model, you dramatically save on computational resources.

## Activation Functions

Activation functions add non-linearity to neural networks.
This makes it possible to approximate non-linear functions.




### ReLU

Popular because its derivative makes computation easy and fast.

One drawback of the ReLU activation is the occurrence of **dead neurons.** We cannot train the parameters of this neuron in the previous layer to obtain output values besides zero. Dead neurons increase with the depth of the layer in an **untrained network.** But that is **not a problem** because dead neurons in later layers can potentially become alive/active again as the input to later layers is changed due to updates to the weights of previous layers.

When implementing your own neural network, it is recommended to start with a ReLU-based network and select the specific activation function based on the properties of the network.

---

Basic Batch Gradient Descent Example:

```
h = lambda x, theta = np.array([1, x, x**2]) @ theta.T # 0.0 + 0.1 * x + 0.2 * x**2
J = lambda theta, X, Y: np.sum((h(X, theta) - Y)**2) / (2 * len(X)) # MSE
# partial derivative of J
phi(X) = np.column_stack([np.ones(len(X)), X, X**2])
gradient_J = lambda theta, X, Y: phi(X).T @ (h(X, theta) - Y) / len(X)

def batch_gradient_descent(X, Y):
    theta = np.zeros(3)
    current_cost = 999999.0
    while True:
        last_cost = current_cost
        for i in range(len(theta)): # Compute gradient for each parameter
            gradient = gradient_J(theta, X, Y, i)
            theta[i] = theta[i] - learning_rate * gradient
        current_cost = J(theta, X, Y) # Calculate cost, e.g. MSE
        if abs(last_cost - current_cost) < 0.001: # Stopping criterion
            break
    return theta
```

# Loss Optimization

An optimizer is responsible to update the network's parameters given the gradients.

Problem: Many local minima and saddle points. No guarantee that global minimum has been reached. Final destination is determined by starting point.

## Gradient Descent

Implicit regularization technique. Iterative optimization that updates model parameters by moving in the direction opposite to the gradient of the loss function in discrete step sizes. It iteratively adjusts parameters by taking small steps in the direction that most steeply decreases the loss.

### SGD (Stochastic Gradient Descent)

Samples (shuffles) a **random subset** of training data **(batch)** → each batch creates a different loss function and results in a different update. This adds noise but improves generalization (= regularization). Tends to overshoot, but escapes local minima. Updates parameters by multiplying the gradients with a small global constant (same for all parameters), called learning rate (usually 0.1). We move slowly towards the (local) minima.

$$\phi_{t+1} \longleftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

$\alpha$ learning rate
$\phi$ weight
**Gradients** of all batches

The loss per **epoch** (a forward and backward pass completed for all batches) is usually the **average of all batch losses.**

### SGD with Momentum

Gradient step is a **weighted sum (= exponential average = momentum)** of **all previous gradients,** which accelerates gradient descent.

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \\ \phi_{t+1} &\leftarrow \phi_t - \alpha \cdot \mathbf{m}_{t+1}, \end{aligned}$$

$\mathbf{m}$ Momentum term
$\beta$ controls degree of smoothing [0, 1)

Problem: Large steps when gradients are large (might be risky).
Small steps when gradients are small (should explore faster).

### Adam (Adaptive moment estimation)

The quasi-standard. **Adaps the learning rate for each parameter individually** by using **momentum.** Normalizes the gradients (sign) to move at fixed distance (governed by the learning rate and the momentum).

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}$$

$\beta, \gamma$ momentum coefficients [0, 1) controls degree of smoothing [0, 1)
$\mathbf{v}$ direction of gradients (normalized gradients). $v_{t+1}$ is Used to **normalize the gradient magnitude**, all that remains is each direction (sign).
$\varepsilon$ small constant for numerical stability. Prevents division by zero. Can stabilize training if gradients are tiny.

$$\tilde{\mathbf{m}}_{t+1} \leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \qquad \tilde{\mathbf{v}}_{t+1} \leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}$$

At the start

$$\mathbf{m}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi} \qquad \mathbf{v}_{t+1} \leftarrow \left( \frac{\partial L[\phi_t]}{\partial \phi} \right)^2$$

then

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \qquad \mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma) \left( \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \right)^2$$

**$\beta$ controls momentum:**

High $\beta$ → long memory (smooth gradient updates).
More stable but slower to adapt to sudden changes in the gradient.
Low $\beta$ → short memory (updates react more to the most recent gradient).
Faster adaptation but noisier updates, which can destabilize training.

**$\gamma$ controls adaptive learning rate:**

High $\gamma$ → slow update, smoother, more stable learning rate adaptation.
More stable, but may stall early training
Low $\gamma$ → reacts quickly to recent gradient magnitudes (more adaptive).
Faster convergence early, but training can become unstable

Should we always use Adam and never look at SGD anymore?
No. In certain situations, SGD (with momentum) generalizes better (but with slower convergence) where Adam often tends to overfit and get stuck in local minima. Because Adam converges faster, it should be used early training, and then switched to SGD (with momentum).

Adam requires two extra variables per parameter:
– First moment estimate (mean of gradients): $m_i$
– Second moment estimate (uncentered variance): $v_i$

## Parameter Initialization

Optimal initialization properties:

1. The mean of the activations should be zero
2. The **variance of the activations** should stay the same across every layer

In other words, the goal is that the variance of the activations in every layer is the same as the input and that the mean is zero. The input (x) should also have a mean of zero → normlization/standardization

Activation Distribution = Distribution of output values from neurons
Gradient Distribution = Distribution of gradient values (derivatives of the loss)

### Xavier Initialization

Normal

$$W \sim \mathcal{N}\left(0, \frac{2}{d_x + d_y}\right)$$

Uniform

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}}\right]$$

$d_x$ input dimension
$d_y$ number of output neurons

A small benefit of using a uniform instead of a normal distribution is that we can exclude the chance of initializing very large or small weights.

Xavier initialization works well for **Tanh** networks.

### He Initialization (Also called Kaiming Initialization)

Standard

$$\sigma_\Omega^2 = \frac{2}{D_h}$$

For both fan-in and fan-out

$$\sigma_\Omega^2 = \frac{4}{D_h + D_{h'}}$$

$D_h$ = dimension input layer
$D_{h'}$ = dimension output layer

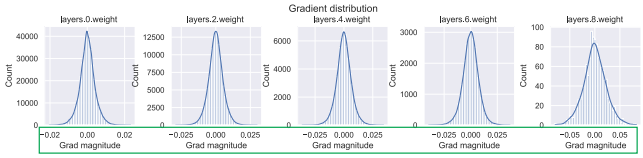
He initialization **ensures that the expected variance remains unchanged** after a linear plus ReLU layer. He initialization works well for **ReLU** networks because ReLU sets half of the inputs to 0. Second version is used when you want a compromise between forward and backward pass variance preservation. Example:

```
D_in  = 128  # Number of input units
D_out = 64   # Number of output units
# He Initialization: variance = 2/D_in
variance = 2.0 / D_in
std_dev = np.sqrt(variance)
# Initialize weights
W = np.random.randn(D_out, D_in) * std_dev
```

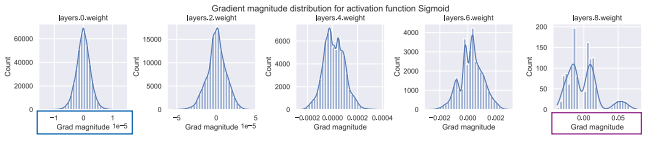
### Vanishing Gradient Problem and Gradient Flow

One important aspect of activation functions is how they propagate gradients through the network. If we have many hidden layers, and if all partial derivatives (gradients) are between –1 and 1, then multiplying them makes them exponentially small. Changes in the first layers will diminish since the gradient becomes extremely small. This results in very slowly changing weights and thus very long training times. If the gradient through the activation function is larger than 1, the gradients exponentially increase and might explode.

Optimally, we want **similar gradient norms** across all layers:

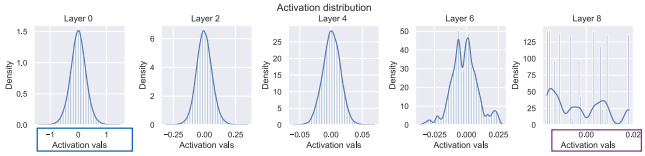



Example: The **sigmoid** activation function shows a clearly undesirable behaviour. While the gradients for the **output layer** are **very large**, the **input layer** has the **lowest gradient norm** across all activation functions with only 1e-5.




This is due to its **small maximum gradient** of 1/4, and finding a suitable learning rate across all layers is not possible in this setup.

Example of small gradients: the **variance of the activation** (post-activation values) becomes smaller and smaller across layers:




**Criterias for fast and feasible Training:** Efficient computation/ implementation, clever memory usage and proper initialization.

**1. Forward pass:** Saves the values, e.g. pre-/activation, we need for backpropagation.

**2. Backward pass:** Calculate each partial derivative. It starts at the outputs and moves backwards through the network. Uses previously stored pre-/ activations. Avoids Redundant computations by reusing prior computations. Gradients add at branches. Example 1:

Forward pass

$$\begin{aligned} \mathbf{f}_2 &= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2 \\ \mathbf{h}_3 &= \mathbf{a}[\mathbf{f}_2] \\ \mathbf{f}_3 &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3 \\ \ell_i &= \mathbf{l}[\mathbf{f}_3, y_i], \end{aligned}$$

f = pre-activation
h = post-activation
l = loss
β = bias
Ω = weights

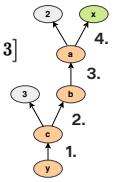
Backward pass: (chainrule)

$$\frac{\partial \ell_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} \frac{\partial \ell_i}{\partial \mathbf{f}_3}$$

How loss $l_i$ changes w.r.t. changes in $\mathbf{f}_2$.

Example 2:

$$y = \frac{1}{|x|} \sum_i \left[(x_i + 2)^2 + 3\right]$$




$$\mathbf{x} = [0, 1, 2]$$

$$\frac{\partial y}{\partial x_i} = \overset{1.}{\frac{\partial y}{\partial c_i}} \overset{2.}{\frac{\partial c_i}{\partial b_i}} \overset{3.}{\frac{\partial b_i}{\partial a_i}} \overset{4.}{\frac{\partial a_i}{\partial x_i}} = [1.333, 2.0, 2.667]$$

$$\frac{\partial a_i}{\partial x_i} = 1 \qquad \frac{\partial b_i}{\partial a_i} = 2 \cdot a_i \qquad \frac{\partial c_i}{\partial b_i} = 1 \qquad \frac{\partial y}{\partial c_i} = \frac{1}{3}$$

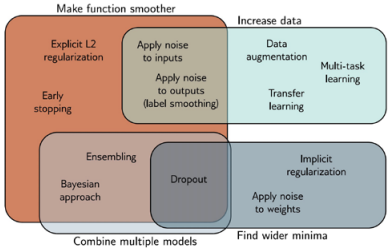
youtube.com/watch?v=d14TUNcbn1k&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=5

## Regularization

Solves the problem of **overfitting.** Adds constraints or penalties to the learning process and removes reduces model complexity. Model becomes simpler and generalizes better.

**Overfitting:** Model learns the training data too well (performs very well on training data) but poorly on unseen (test/validation) data.

Constrain the degree of flexibility of a model, e.g. remove parameters or penalize higher degree polynomial factors. Any method that limits the expressiveness of the hypothesis space by adding constraints to learning; e.g., pruning decision trees. Combats overfitting by penalizing high flexibility. A regularizer biases a model toward a subset of the solutions with similar training loss.




### Explicit Regularization

Add an additional term to the loss function that puts a high penalty on less desirable parameters.

**L2, L1, L0 Regularization:**

**L2:** Penalizes the sum of squares of the paramters: (= weight decay). Smaller weights create smoother functions with more bias. Also called **Ridge** Regression.

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[ \mathrm{L}[\boldsymbol{\phi}, \{\mathbf{x}_i, \mathbf{y}_i\}] + \lambda \sum_j \phi_j^2 \right]$$

**L1:** Penalizes the absolute values of the weights. Produces sparser (fewer weights are non-zero)solutions than L2, much easier to optimize than L0.

$$argmin_\phi[L[\phi, \{\mathbf{x}_i, \mathbf{y}_i\}] + \lambda \sum_j |\phi_j|]$$

**L0:** Adds a constant value (penalty) to every non-zero weight.

Elastic net penalty: If both L2 and L1 terms are added/used.

### Implicit Regularization

Any heuristic approach: early stopping, ensembling, dropout, data augmentation, etc.

**Gradient Descent:** Iterative optimization that updates model parameters by moving in the direction opposite to the gradient of the loss function in discrete step sizes.
**Early Stopping:** Stop training before loss on validation set increases again. Forfeit double descend and ultimative performance.
**Ensembling:** Build a group of several models with different initializations and combine their predictions. Model errors are independent and cancel out.
**Transfer- and Multi-Task Learning:** Pre-train model on related task *A*, then continue on original task *B*. If we have limited data on task *B*, but a lot of data for task *A*. E.g. train model on segmentation, then switch to depth layer.

**Bagging (Bootstrap and Aggregation):** Optain different training sets by resampling with replacement from original set.
**Dropout:** Set a random subset of hidden neurons to zero in each SGD iteration. Network becomes less dependent on any hidden neuron and encourages smaller weights and shared responsibility. **Monte Carlo Dropout:** Run the network multiple times with dropout and combine the result.
**Data Augmentation:** Data augmentation reduces the risk of overfitting and helps the network to generalize better. Data augmentation should only be applied to training set, but not validation, nor testing set. Data augmentation reduces the risk of overfitting and helps CNNs to generalize better.
**Apply Noise:** Add noise to input data, weights or labels. Smooths function and discourages overconfident behaviour. Too much noise hurts tho.
**Self-supervised Learning:** Learning from unlabeled data by creating its own supervision signal from the data itself.
**Generative:** Mask/remove a part of each sample and predict the missing part (e.g., missing words, pixels, etc.)
**Contrastive:** Create two augmented views of same image. Learn that these views are similar (positive pairs) vs. different images (negative pairs).
No information removed: Both views have full information, just transformed.

### Factors for Training Success

Data, Regularization (improves loss surface), **Overparameterization** (crucial), **Activation function** (crucial, affects training difficulty and gradient propagation), Initialization (shortens training time), Depth, learning rate and batch sizes (smaller bachtes and larger leraning rates generalize better).

### Lottery Ticket Hypothesis

For a given network, a smaller but equally performing one can be found by training or pruning weights (iteratively remove smallest weights) with lowest magnitude (as they are less important).

## Convolutional Neural Networks (CNN)

CNNs are very efficient because they use considerably less weights than fully connected networks. For instance, a 20-layer convolutional network outperforms a 5-layered network for image classification.

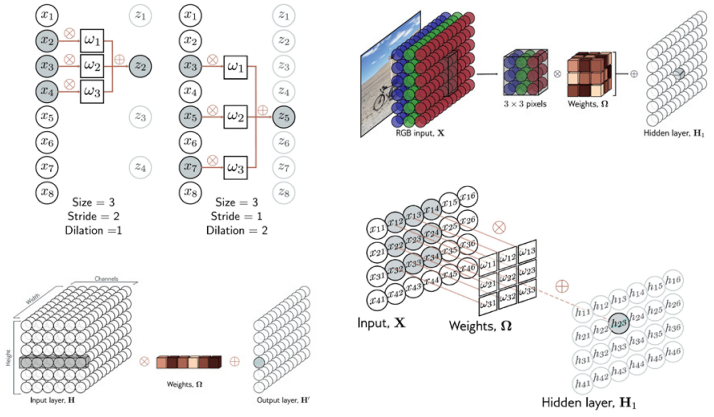
Convolution layers is **translation-equivariant,** means that when you translate (shift) the input, the output translates in the same way. This introduces a useful **inductive bias** and rules out many possible configurations).

**Equivariance:** A function $f$ is equivariant if applying the a transformation $T$ before or after the function gives equivalent results: $f(T(x)) = T(f(x))$.

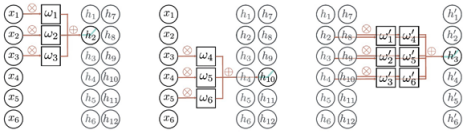
CNNs reuse model parameters (kernels) for spatial position to achieve translation equivariance. But CNNs do not handle rotations and scaling.

**Convolution (Faltung)** = Weighted sum of inputs.

**Kernel/filter:** A tensor (vector, matrix, volume) of weights.
**Padding:** How to handle edge cases, e.g. 0 padding
**Stride:** Kernel shift, e.g. 1, 2, ...
**Dilation:** Number of 0 weights in the kernel to cover larger input area.




A kernel can be viewed as a **feature extractor** (= patterns/templates). Convolution kernels are basically template matching. An object can be present in any location (as long its not scaled or rotated). Each filter creates a new channel. Example of two filters creating two channels:



Examples of learnt filter:

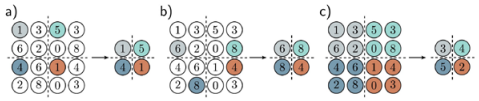

### Receptive Field

Refers to the region of the input that influences a particular neuron's activation. Each neuron in a CNN *sees* only a specific portion of the original input. The receptive field is the size of that visible region.

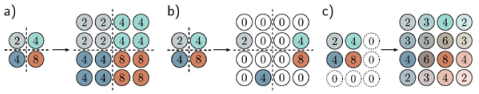
### Pooling

Used to down- (pooling) or up-sample (unpooling) convolutional layers. Pooling-layers reduce spatial resolution and but keep the most important signal in a local region. Pooling reduces sensitivity (= variance) and therefore **increase invariance.** Invariance comes from pooling layers that aggregate acticatations at the end of the network.

Pooling Examples:




a) Sub-sampling. b) Max pooling. c) Mean pooling




a) Nearest neighbour. b) Max unpooling. c) Bilinear Interpolation

### Tasks vor CNNs

Computer Vision:
Image classification: Predicting the correct label
Semantic Segmentation: Per-pixel class classification
Object Detection: Predicting bounding boxes and labes in an image (YOLO).

### Inductive Bias

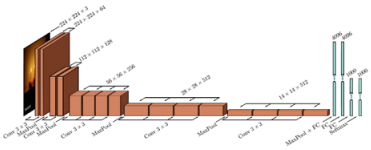
CNN's are optinionated. They process images in a very specific way (e.g. convolution). Models with strong inductive bias make strong assumptions about the nature of the input data, which limits the function space they can explore during training. We might be unwantedly exlude the best function spaces.

#### Assumtions in CCNs

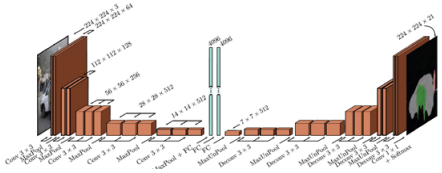
- **Locality:** Convolution size makes it prioritizes texture over the overall object structure.
- **Translation invariance:** Convolution kernels are basically template matching. An object can be present in any location.
- **Hierarchical struture:** Assume images are hierarchical (pooling-layers)

### VGG Network (Visual Geometry Group)

AlexNet but with increased depth.



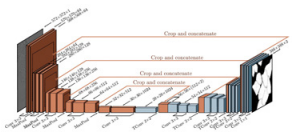
### Encoder-Decoder Networks



Used for **Semantic Segmentation.**

**Deconvolution (Transposed Convolution):** Upsamples the input via max unpooling, performing the reverse spatial transformation of a regular convolution. Used in: Image generation (GANs, VAEs), Semantic segmentation, Super-resolution and Autoencoders (decoder part).

#### U-Net



Specific type of encoder–decoder network with **skip connections** that preserve high-resolution information.

Encoder–decoder: general architecture pattern
U-Net: a particular encoder–decoder with skip connections

Encoder: compresses input into a latent representation
Decoder: reconstructs or maps that representation to an output

## Residual Architectures

Problem: Deep learning works better with more layers, but accuracy gets worse after a certain point of depth. **Residual architectures (networks with skip connections)** were designed to enable very deep neural networks.

New development: Hyper Connecitons (= parallel residual connecitons)

### Shattered Gradients

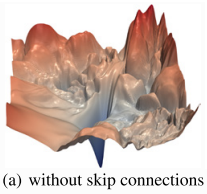
Adding more layers to a convolutional network does improve performance. However, image classification performance decreases again as further layers are added. Every layer depends fully on the output of the last layer. **Little change in input can cause large change in output. Final gradient reacts unpredictably** to changes in the processing of early layers. In very deep networks, gradients of nearby inputs tend to be uncorrelated (autocorrelation is close to zero). This means: the loss surface is not smooth at all. Solutions: **Residual Blocks (ResNet)** and **DenseNet**

### Residual Blocks (Residual Layers)

Skip connections enable depth and help overcome shattered gradients. Residual connections are highly effective as it **supports stable gradient propagation through the network.** Residual connections help overcome shattered gradients.

Standard feed-forward neural network:

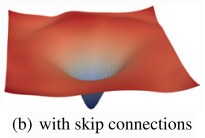
$$\begin{aligned}\mathbf{h}_1 &= \mathbf{f}_1[\mathbf{x}, \phi_1]\\ \mathbf{h}_2 &= \mathbf{f}_2[\mathbf{h}_1, \phi_2]\\ \mathbf{h}_3 &= \mathbf{f}_3[\mathbf{h}_2, \phi_3]\\ \mathbf{y} &= \mathbf{f}_4[\mathbf{h}_3, \phi_4] = \mathbf{f}_4\Big[\mathbf{f}_3\Big[\mathbf{f}_2\big[\mathbf{f}_1[\mathbf{x}, \phi_1], \phi_2\big], \phi_3\Big], \phi_4\Big]\end{aligned}$$



(a) without skip connections

**Residual** (also called **Skip Connections**):

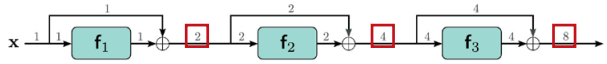
$$\begin{aligned}\mathbf{h}_1 &= \mathbf{x} + \mathbf{f}_1[\mathbf{x}, \phi_1]\\ \mathbf{h}_2 &= \mathbf{h}_1 + \mathbf{f}_2[\mathbf{h}_1, \phi_2]\\ \mathbf{h}_3 &= \mathbf{h}_2 + \mathbf{f}_3[\mathbf{h}_2, \phi_3]\\ \mathbf{y} &= \mathbf{h}_3 + \mathbf{f}_4[\mathbf{h}_3, \phi_4] = \mathbf{x} + \mathbf{f}_1[\mathbf{x}]\\ &\quad + \mathbf{f}_2[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]]\\ &\quad + \mathbf{f}_3\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]]\big]\\ &\quad + \mathbf{f}_4\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big] + \mathbf{f}_3\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]]\big]\Big]\end{aligned}$$



(b) with skip connections

Skip connections allow information to bypass one or more layers by creating direct paths from earlier layers to later layers. Each function $f_k$ learns an additive change to the current representation. Each additive combination is known as a residual block or residual layer = additive combination of input and output. This can be seen as an ensemble of sub-networks.
→ Gradient becomes more stable due to contribution of **short paths.**

Problem: **Exploding gradients** despite He initialization. Because the input of previous block is added back to the output, the variance doubles at each laxer → Solution: Batch Normalization




### Batch Normalization (BatchNorm)

Batch normalization **shifts** and **rescales** each activation *h* so that its mean *m* and variance *s* **across the batch** become values that are learned during training. Transforms the input to each layer to have unit variance, and with He initialization, the output variance will also be one. Helps to control the initial gradients in a residual network.

**→ Two new parameters to learn:**

δ Shift parameter (mean)
γ Scale parameter (variance)

$$m_h = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} h_i$$

$$s_h = \sqrt{\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}(h_i - m_h)^2}$$

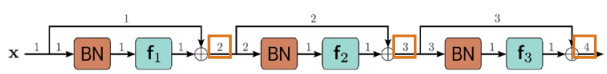
1. Standardize the batch activations to have mean zero and unit variance:

$$h_i \leftarrow \frac{h_i - m_h}{s_h + \epsilon}$$

2. The normalized variable $h_i$ is then scaled by γ and shifted by δ:

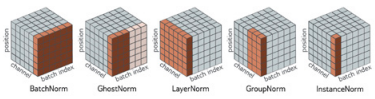
$$h_i \leftarrow \gamma h_i + \delta$$

The **variance** increases linearly with the number of residual blocks:




Loss surface becomes even more smooth and higher learning rates are possible. BatchNorm injects noise into training → regularization.

**Normalization Schemes: BatchNorm:** Normalizes across the batch dimension. For each feature, computes mean/std across all samples in the batch. Usecases: CNNs, computer vision task. **LayerNorm:** Normalizes across the feature dimension. For each sample, computes mean/std across all features. Usecases: Transformers and attention-based models, RNNs, NLP




### ResNet (Resembles AlexNet)

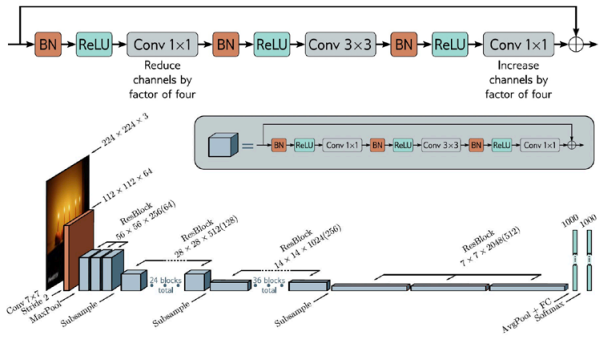
Architecture for enabling very deep neural networks using **Bottleneck residual block** to save parameters by reducing channels via conv1x1.

Bottleneck: Information is forced through a narrower representation than what came before (that can be fewer channels, lower resolution, etc.), often followed by an expansion back to higher dimensional space.

The ResNet paper (https://arxiv.org/abs/1512.03385) is one of the most cited AI papers ever. and has been the foundation for neural networks with more than 1000 layers.

#### Bottleneck Block

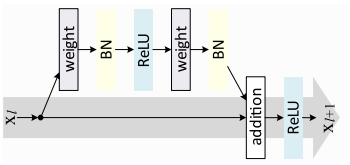
1×1 conv (reduce channels) → 3×3 conv (process) → 1×1 conv (expand ch)



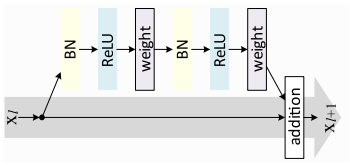

Instead of modeling $\boldsymbol{x}_{l+1} = \boldsymbol{F}(\boldsymbol{x}_l)$ we model $\boldsymbol{x}_{l+1} = \boldsymbol{x}_l + \boldsymbol{F}(\boldsymbol{x}_l)$

*F* is a non-linear mapping (usually a sequence of NN modules likes convolutions, activation functions, and normalizations)

Original ResNet block



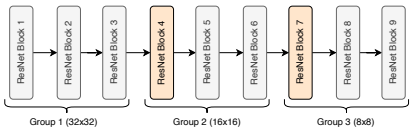

Pre-activation ResNet block




For very deep network the pre-activation ResNet has shown to perform better. Example code for original ResNet block:

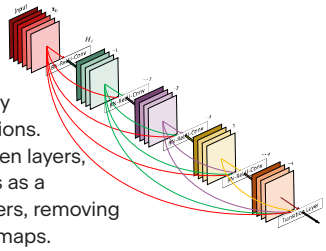
```
# Network representing F
net = nn.Sequential(
   # No bias needed as the Batch Norm handles it
   nn.Conv2d(c_in, c_out, kernel_size=3, padding=1, stride=1, bias=False)
   nn.BatchNorm2d(c_out),
   act_fn(),
   nn.Conv2d(c_out, c_out, kernel_size=3, padding=1, bias=False),
   nn.BatchNorm2d(c_out))
```

The overall ResNet architecture consists of stacking multiple ResNet blocks [3, 3, 3], of which some are downsampling the input:

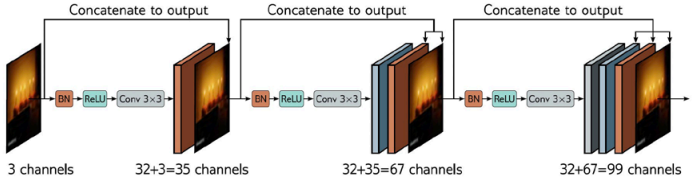



ResNet has been shown to produce **smoother loss surfaces** than networks without skip connection. **SGD** often leads to a slightly better accuracy on plain, shallow ResNets than **Adam.**

### DenseNet



Another architecture for enabling **very deep neural networks** and takes a slightly different perspective on residual connections. Instead of modeling the difference between layers, DenseNet considers residual connections as a possible way to reuse features across layers, removing any necessity to learn redundant feature maps.

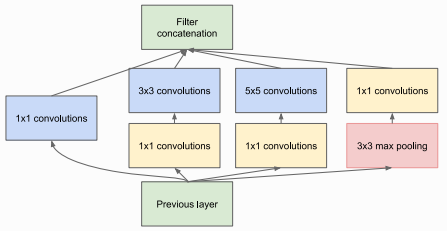



DenseNet provides an efficient way of reusing features by having each convolution depend on all previous input features.

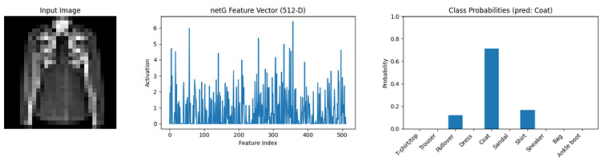
The last layer, called the transition layer, is responsible for reducing the dimensionality of the feature maps in height, width, and channel size.

### Inception Block

An Inception block applies four convolution blocks separately on the same feature map: a 1x1, 3x3, and 5x5 convolution, and a max pool operation. This **allows the network to look at the same data with different receptive fields.** The additional 1x1 convolutions before the 3x3 and 5x5 convolutions are used for dimensionality reduction (crucial step as the feature maps of all branches are merged afterward, and we don't want any explosion of feature size). As 5x5 convolutions are 25 times more expensive than 1x1 convolutions, we can save a lot of computation and parameters by reducing the dimensionality before the large convolutions.




### Splitting Feature Extractor and Classifier

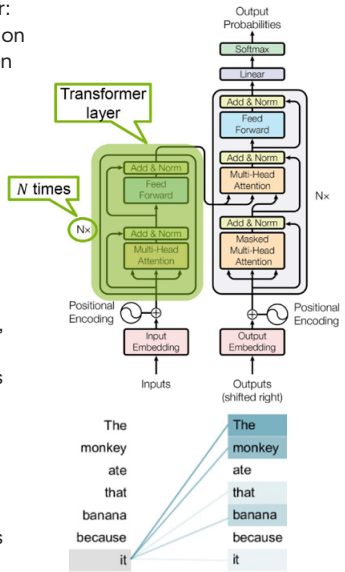



Using a pre-trained backbone (feature extractor net) and a small task-specific head (classifier net). Allows to freeze backbone and fine-tune classificator independently. Also allows to swap heads without touching the backbone.

## Transformers

Transformers solve the problem of modeling long-range dependencies (→ vanishing/exploding gradients) of **sequence processing,** being able to remember long contexts, efficiently and in parallel.

Compares to the CNN, the transformer: lower bias, higher variance. The attention mechanism is a superset of convolution (thats what the original google paper claims). It can implement convolution.

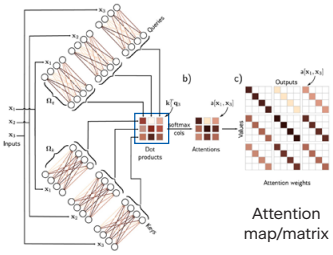



### Self-Attention Block

Enables parameter-sharing and context-dependency of meaning (e.g. meaning of words in its context). Self-attention is a mechanism that allows each element in a sequence to attend to (focus on) all other elements, learning which parts are most relevant to each other. Self-attention computes how much each element should "pay attention to" every other element. Instead of position-dependence, it achieves context-dependence routing (attention as routing). Hypernetwork architecture: A sub-network computes weights (attention matrix) for another.

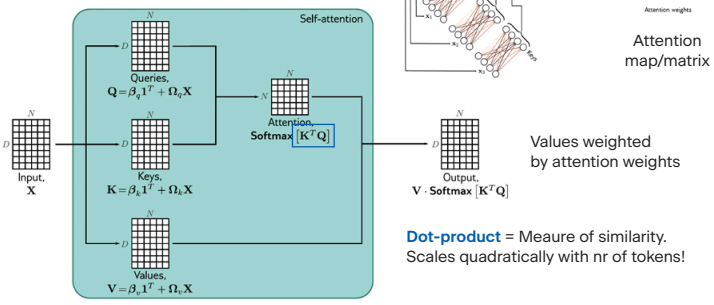



For the word "it" in: "The animal didn't cross the street because it was too tired" self-attention learns that "it" should pay high attention to "animal" (not "street").




Attention map/matrix

Example: *N* tokens of size *D*




Values weighted by attention weights

**Dot-product** = Meaure of similarity. Scales quadratically with nr of tokens!

**Scaled dot-product self-attention** for better gradient flow:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V}\cdot\mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right]$$

### Attention

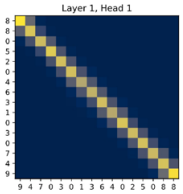
The attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements' keys. We want to dynamically decide on which inputs we want to "attend" more than others. It consists of:

**Query:** The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.

**Keys:** For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is "offering", or when it might be important.

**Values:** For each input element, we also have a value vector. This feature vector is the one we want to average over.

**Score function:** To rate which elements we want to pay attention to. It outputs the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product.




The **attention map/matrix** (right img) shows how much each token attends to every other token. Each row is a probability distribution over all tokens. A diagonal-ish attention map means: Token i mostly attends to itself or nearby tokens.

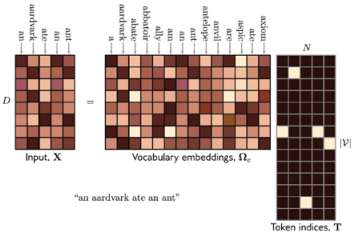
### Tokens and Embeddings

A **token** is a descrete unit of text. A text can't be fed directly to a model, it must first be broken into smaller units. Tokens can be words, subwords, or characters, depending on the tokenizer. Example: ["Hel", "lo", "wor", "ld"]

**Byte Pair Encoding (BPE)** is a subword tokenization algorithm used in NLP to split text into tokens in a way that balances vocabulary size and flexibility.
Idea: Common words → one token, Rare words → split into smaller pieces
Example: low, lower, lowest might become low, er, est
Advantages: No unknown tokens (words)

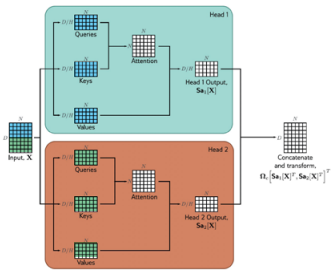
An **embedding** is a numerical representation of a token; a dense vector that represents a token in a high-dimensional space (vocabulary matrix). It converts discrete tokens into numbers that neural networks can process. Embeddings capture semantic relationships between tokens. Similar words have vectors that are close together. Embeddings (vocabularies) are are parameters of the neural network and are learned.
Example: Token "king" → Embedding [0.23, -1.2, 0.5, ..., 0.8]

Text → Tokens → Token IDs → Embedding
"Hello world" → ["Hello", ... ], "Hello" → 1542 → [0.23, -1.2, ..., 0.8]



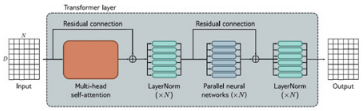

### Multihead Self-Attention

Self-attention applied in parallel for **efficient computation.** Splits tokens.




### Transformer Layer

Uses **LayerNorm** to improve activation/gradient flow and to provide small regularizatino.



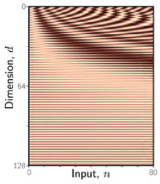

### Applications

Mainly: Text (Natural Language) and image processing. Originally designed from machine translations.

**Transformer Models**
**Encoder:** Learns vocabulary, supports NLP tasks, e.g. BERT
**Decoder:** Text/tokens generation (= generative model), e.g. GPT3
**Encoder-decoder: Sequence-to-sequence** tasks such as translation.

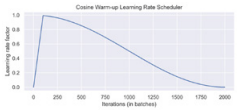
### Positional Encoding

Problem: The self-attention mechanism does not take into account the order of the inputs. The Multi-Head Attention block is permutation-equivariant. it cannot distinguish whether an input comes before another. Solution: A matrix $\Pi$, where each column is unique, is added to the input X that encodes positional information.



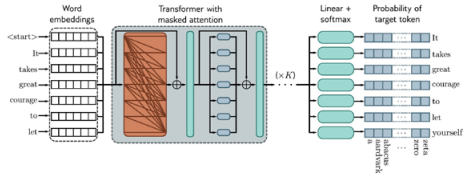

### Training

Commonly used technique for training a Transformer is learning rate warm-up: Gradually increase the learning rate from 0 on to our originally specified learning rate in the first few iterations. The iteratively applied Layer Normalization across layers can lead to very high gradients during the first iterations. Learning-rate warm-up is used in transformers to stabilize early training by preventing large gradient updates before attention weights and layer-normalization statistics have settled.




### Encoder Models

Autoregressive models (e.g. GPT3) learns the joint probability of tokens and to predict next token from previous token.




### Vision Transformer (ViT)

Tackled the problem of image resolution by dividing the image into 16×16 patches. It uses supervised pre-training (unlike BERT). Each of these is projected via a learned linear transformation to become a patch embedding. These patch embeddings are fed into a transformer encoder network, and the <cls> token is used to predict the class probabilities. They have global receptive field because all patches attend to each other.

### Comparison: Complexity per Layer

| Layer Type | Complexity per Layer | Sequential Operations | Maximum Path Length |
|---|---|---|---|
| Self-Attention | $O(n^2 \cdot d)$ | $O(1)$ | $O(1)$ |
| Recurrent | $O(n \cdot d^2)$ | $O(n)$ | $O(n)$ |
| Convolutional | $O(k \cdot n \cdot d^2)$ | $O(1)$ | $O(log_k(n))$ |
| Self-Attention (restricted) | $O(r \cdot n \cdot d)$ | $O(1)$ | $O(n/r)$ |

*n* is the sequence length, *d* is the representation dimension and *k* is the kernel size of convolutions.

## Generative Adversarial Networks (GAN)

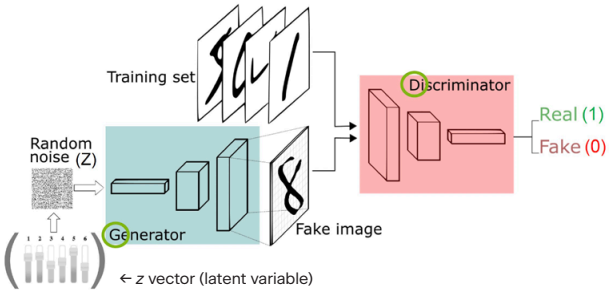
GANs learn a generator network that transforms random noise into data that is indistinguishable from a training set. Idea: Learn about high-dimensional probability distribution. The PDF (Probability Density Function, e.g. gaussian/ normal distribution) tells everything there is to know about the data without seeing it. Allows for sampling data from the underlying distribution. Process: 1. Estimate distribution. 2. Sample from distribution to generate data.

Flavors of Generative Models: Statistical: Directly model the PDF. Graph-based: Models with latent variables (e.g. Boltzmann machines) Autoencoders: NNs that reconstruct compressed representations of data.

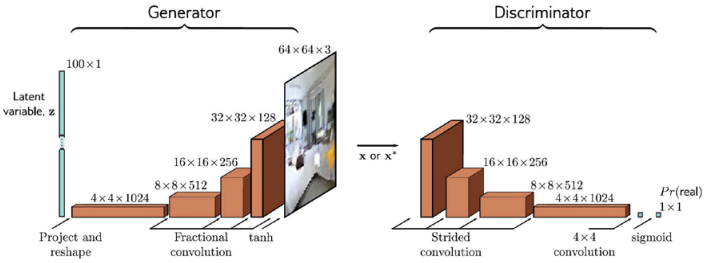
### Adversarial Nets

Adversarial = feindlich, gegnerisch

The generator *G* is trained using a *D* discriminator network that tries to distinguish real examples from generated samples. The generator *G* is then updated so that the data that it creates is identified as being more real by the discriminator *D*. The generator *G* tries to trick the discriminator *D*. Both *G* and *D* are FCNs. A Fully Convolutional Network is a network that uses only convolutional (and related) layers, no dense (fully connected).




**Deep Convolutional GAN (DCGAN),** specialized for generating images: The input to the generator g[z, θ] is 100D a latent variable z sampled from a uniform distribution. Strided convolution used in *D* instead of (max) pooling. Fractionally-strided (Dilation) convolution used in *G* to -up-sample input.




### Training

1. Sample z and Generate fake sample: $x_{\text{fake}} = G(z), \quad z \sim \mathcal{N}(0, I)$
2. Discriminator tries to distinguish x_fake from real data
3. Generator learns to make x_fake more realistic

The discriminator D is a binary classifier that tries to maximize its ability to correctly classify real and fake samples:

$$L[\phi] = \sum_j -\log\Big[1 - \text{sig}[\text{f}[\mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}], \boldsymbol{\phi}]]\Big] - \sum_i \log\Big[\text{sig}[\text{f}[\mathbf{x}_i, \boldsymbol{\phi}]]\Big]$$

real data, fake data

The objective of *G* is to fool *D*:

$$L[\boldsymbol{\theta}] = \sum_j \log\Big[1 - \text{sig}[\text{f}[\mathbf{g}[\mathbf{z}_j, \boldsymbol{\theta}], \boldsymbol{\phi}]]\Big]$$

Training is not guaranteed to converge. $D$ and $G$ play a game-theoretic (minimax) game against each other. Gradient descent isn't meant to find the corresponding Nash Equilibria (a Nash equilibrium is a strategy profile in which no player can improve their payoff by unilaterally deviating from their strategy. Everyone is stuck doing what they're doing, changing alone only makes things worse).

### *z* Vector (latent variable)

The z vector (latent variable) in GANs is a crucial component that represents the input to the generator, a random code from which the generator creates synthetic data. z is a random vector sampled from a simple prior distribution (usually standard normal or uniform), which the generator transforms into realistic-looking data. Different z vectors produce different outputs. Without z, the generator would always produce the same output (no diversity). The z vector describes the image features. The z vector is a random input to the generator, it sampled from a simple distribution $z \sim \mathcal{N}(0, I)$. This seed determines what gets generated

#### Properties of the Latent Space:

**Continuity and Interpolation:** Nearby z vectors should produce similar outputs. You can smoothly morph between samples.

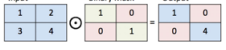
**Latent space arithmetic:** In well-trained GANs, you can perform vector arithmetic. Example: man_w_glasses - man + woman = woman_w_glasses


### Use Cases

Image impainting, generative images from text, image segmentation,

#### Image impainting (reconstruction)


Find *z* such that $x_{reconstructed} = M \odot x_{corrputed} + (1 - M) \odot G(z)$




$\odot$ = element-wise product
$M$ = binary mask (0 for missing pixels)

$$L_{contextual}(z) = \|M \odot G(z) - M \odot x_{corrupted}\|_1$$
$$L_{perceptual}(z) = \log\big(1 - D(G(z))\big)$$
$$L(z) = L_{contextual}(z) + \lambda \cdot L_{perceptual}(z)$$

$$\text{Optimize } \hat{z} = \arg\min_z L(z)$$

Learning smooth approximations of complex probability density functions (PDF) enables us to sample previously unseen examples. That is, we can create new images, new music, etc.

## Reinforcement Learning

An agent learns to act by interacting (= sequential decision making) with a stochastic environment. The goal is to learn a policy (= probability distribution over all possible actions, represented by a neural network) that maps the current state to an action.

We first sample the environment (creating experiences) by creating episodes/trajectories (= many simulations) and perform a Monte Carlo Tree Search to build the value function (= maps state, and optionally action, to future reward).

**(Experience) Replay Memory:** Stores batches of transitions (= trajectories) in a buffer that the agent observes. We'll need trajectories in order to use **supervised learning** to train the neural network. By sampling from it randomly, transitions/episodes are less correlated. This stabilizes and improves the DQN training.

Blogpost: https://karpathy.github.io/2016/05/31/rl/

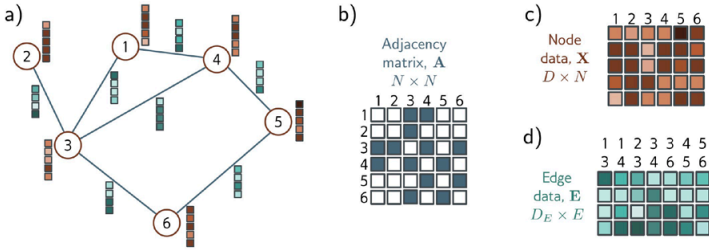
## Graph Neural Networks (GNNs)

Many objects in the real world naturally take the form of graphs. Graphs are very flexible are all-rounder to present data.

### Types of Graphs

Social networks, (word) relationships, point cloud, Recommendation System, Transportation Networks, molecules, Knowledge Graph (= relationships between entities), segmentation (dense prediction), text.

### Graph Representation




**Adjacency Matrix:** Defines which nodes are connected.
0 = no connection, 1 = connection (can be weigthed)

**Node embeddings (edge feature):** e.g. age, sex, job, etc.
**Edge embeddings:** relationship, e.g. friend, husband, etc.

A graph neural network is a model that takes the **node embeddings X** and the **adjacency matrix A** as inputs and passes them through a series of layers.

### Graph Tasks

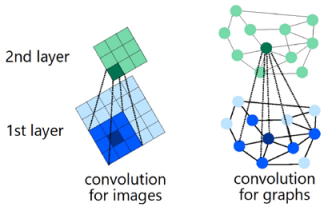
**Graph-level:** Graph classification, regression, generation, e.g. image classification. The network assigns a label or estimates one or more values from the entire graph, exploiting both the structure and node embeddings.

**Node-level:** Node classification, regresison, clustering. The graph assigns a label (classification) or values (regression) to each node of the graph.

**Edge-level:** Link prediction, link classification, e.g. recommendation. The network predicts whether or not there should be an edge between nodes.

### Message Passing

Can be seen as convolution:




```python
def forward(self, features, adj_mat):
    # message passing
    h = self.conv1(features, adj_mat)
    h = h.tanh()
    # ...
    return self.classifier(h), h
```

**Aggregation Function:** Determines how information from neighboring nodes is combined during message passing. An aggregation function combines messages from a node's neighbors into a single vector.
Example: mean, max or attentin-based aggregation.

**Aggregation vs pooling:**
Aggregation: Node-level (neighbors → node)
Pooling: Graph-level (nodes → graph)

### What does a GNN learn?

Weight matrices, Aggregation parameters, Attention weights (if used). These parameters define how information is propagated and transformed over the graph. It produces: Node, Edge and Graph embeddings.

## Responsible AI Development

Technology changes human behaviour.
AI, as any technology, is a **dual use,** it can be used for good and bad.
→ Design the system so it positively impacts human behaviour.

### FAT ML Principles

Helps developers to build algorithmic systems in publicly accountable ways. Accountability = The obligation to report, explain, or justify algorithmic decision-making and mitigate any negative social impacts or potential harms.

**5 Principles**

- **Responsibility:** Make somebody available who will take care of adverse individual/societal effects.
- **Explainability:** Explain any algorithmic decision in non-technical terms.
- **Accuracy:** Report all sources of uncertainty/error in algorithms and data.
- **Auditability:** Enable 3rd parties to probe and understand system behavior.
- **Fairness:** Ensure algorithmic decisions are not discriminatory.

### Threats through AI Systems

**Algorithmic Bias:** systematic and repeatableerrorsin a computer system that create *unfair* outcomes, such as *privileging* one category over another. Caused by learning algorithms picked up human biases from the training data.

**Overreliance and Intelligence Illusion:** The tendency of individuals to depending too heavily on AI, which can lead to poor decision-making.

**Oxytocin Hacking:** Hijacking attachment and relationships through AI systems. Enhances feelings of love, trust, and happiness, etc.

**Mass Unemployment (no proof):** AI automats tasks (not jobs) but can hardly automate the *glue* between these tasks. AI as a normal technology takes years to be really adopted.

### Impact Areas

Digital Security: Autotmate cyberattacks or social engineering
Physical Security: Drones or autonmous weapons
Political Security: Surveillance, Propagande, Deception (synthetic news)

Solutions: Promote a culture of responsiblity

## Varia

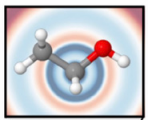
### About the Term «AI»

The term "Artificial Intelligence" (AI) was coined in 1956 during a workshop at Dartmouth College, where researchers aimed to explore how to create machines that could think and learn like humans. John McCarthy (American computer scientist and cognitive scientist), one of the attendees, suggested the name to describe their ambitious goal of developing intelligent machines.

### Equivariance

Equivariant Neural Networks and Invariance vs Equivariance: https://www.youtube.com/watch?v=2bP_KuBrXSc

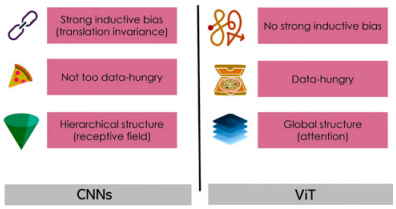
**Point Convolution,** see paper SchNet: A continuous-filter convolution neural network for modeling quantum interactions



Blogpost: https://maurice-weiler.gitlab.io/blog_post/cnn-book_1_equivariant_networks/

Euclidian neural network: https://github.com/e3nn/e3nn

### ViT vs CNNs




CLS token in ViT: The CLS token (class token) in Vision Transformers serves as a learnable summary representation of the entire image, used for classification. Its a special learned embedding (vector) that's prepended to the sequence of image patch embeddings.

Image patches: [patch$_1$, patch$_2$, ..., patch_N]
Add CLS token: [CLS$^{\downarrow}$, patch$_1$, patch$_2$, ..., patch_N]

### Important Models

AlexNet (2012) Deepness + GPUs win.

VGG16 (2014) Add more layers.

GoogLeNet (2014) Scale invariance/efficiency.

Seq2Seq Models (2014) NNs can translate.

ResNet models (2015) Learn the residuals.

DenseNet (2017) More direct/skip connections

Transformers (2017) Attention is all you need.

EfficientNet (2019) How to scale up a network

GPT-2 (2019) Scaling gives zero shot learning.

They all use Softmax (Multiclass-classification).

### Auto-Regressive Model

https://www.youtube.com/watch?v=zc5NTeJbk-k

In image generation: Generates the image per-pixel (output is one pixel).

### Foundation Model

Foundation models are large-scale models trained on broad data to learn general-purpose representations that can be adapted to many downstream tasks. They are pre-trained on massive datasets and serve as a base (foundation) for specialized models via fine-tuning or prompting ~ Swiss Army knife of AI models.

They use Self-supervised pretraining, e.g. they Learn from unlabeled data using objectives like predicting masked words, next tokens, or image patches. They can be adapted to new tasks with small labeled datasets.

### Diffusion Model

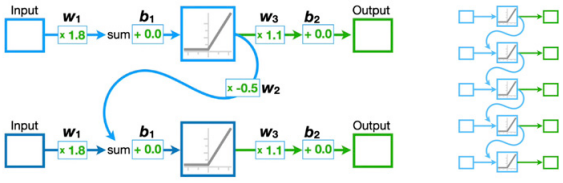
https://www.youtube.com/watch?v=zc5NTeJbk-k

In image generation: Learns to reduce noise for all pixels at once = noise reduction.

### Recurrent Neural Networks (RNNs):

RNNs accept different amounts of input lengths.

RNNs have feedback loops. It uses one input value but due to the feedback loop it is possible to use sequential input values. Think of a shift register. This is called **unrolling.**




Problem with unrolling: The more we unroll, the harder and unstable it becomes to train vanishing/exploding gradients) → **Long-Short-Term** Memory Networks (Transformers are even better).

Blogpost: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

### Large Language Models

An LLM is an instance of a foundation model applied specifically text-like things. «Large» refers to the size (amount of weights/parameters, tens of gigabytes in size) of the model and trained on enormous amount of data (in the order of petabytes). The architecture is usually a transformer (e.g. GPT).

### The Bitter Lesson

A blog post by Richard Sutton (one of the founders of RL):

Throughout AI history, general methods (e.g. learning and search) that exploit and scale with computation outperform clever algorithms built on human insights.

→ Scaling compute wins in the long run
→ Human-crafted knowledge doesn't scale well

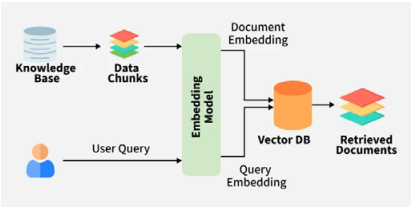
### Retrieval-Augmented Generation (RAG)

Problems of LLMs is that they are frozen in time. RAG solves this problem. It solves this by context injection (into the prompt).

AI framework that combines information retrieval with text generation models like GPT to produce more accurate and up-to-date responses. Instead of relying only on pre-trained data like traditional language models, RAG fetches relevant documents from an external knowledge source before generating an answer.

Static models rely on training data that may become outdated. RAGs dynamically retrieves latest information ensuring relevance and accuracy in real time with high contextual relevance.

Retrieve: Find useful information
Augment: Add it to the Ai's knowledge
Generate: Create a better response (using LLM's)




**RAG vs. Long Context**
RAGs don't append the data sources directly to the prompt, but retrieve it from a vector database. RAGs hava a heavy stack and many parts: chunking strategy, embedding model to encode the data), vector db, re-ranker (to rank the results). Modern LLMs have much larger **context windows,** that allows them to append the data source directly to the prompt. However, the model might forget previously information when out of the context window. RAG is still better at retreiving old information.

**LangChain** is a library that allows you to to things like RAG.

```
prompt = PromtTemplate.from_template("""You are an AI assistant who
answers questions with the provided context.
Context:
{context}
Question:
{question}
Answer:""")
ollama = ChatOllama(model="llama3:70b")
chain = {"context": lambda q: fetch_data(), "question":
RunnablePassthrough()} | prompt | ollama | StrOutParser()
query = "Please make a summary."
print(chain.invoke(query))
```

### RAG vs. MCP

Retrieval-Augmented Generation (RAG) enhances LLMs by retrieving relevant information from external sources like documents or databases to improve response accuracy, while Model Context Protocol (MCP) allows LLMs to access live data and perform actions by connecting to APIs or tools. RAG is best for static, unstructured data, while MCP handles dynamic, structured data and tasks. RAG and MCP can be used together.

**RAG** is about text, it pulls information from documents and enhances your prompts with that knowledge. RAG makes a semantic search every single time a prompt is sent.

**MCP** can pull anything, databases, files, APIs, and it can even take actions. Yes, it's more versatile, like the Swiss Army knife of the AI world. MCP is on-demand. Better for agentic workflows.

### Vector Database and Vector Search

A vector DB stores unstructured data as vector embeddings that encodes semantic meaning and similar objects are close together in vector space, e.g. an image of a sunset and the word "sunset".

This allows to perform "similarity searches". It uses **Vector Indexing** and more specifically **Approximate Nearest Neighbour (ANN)** algorithm.

Contrastive learning is used to create the vector space embeddings (e.g. CLIP for images).

Similarity of vectors is usually measured by cosine distamce which is simply the angle between two vectors (but noth their lengths).

```
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformer/
all-mpnet-base-v2")
e1 = embeddings.embed_query("Why is the sky blue?")
e2 = embeddings.embed_query("The sky is blue due to scattering.")
e1e2 = cosine(e1, e2)
```

```
query = "...
results = vector_db.similarity_search(query, k = 10) # ret. 10 closest
results[0].page_content
```

### LangChain

Opensource orchestration framework (Python, Javascript) for the development of applications, that use LLMs.