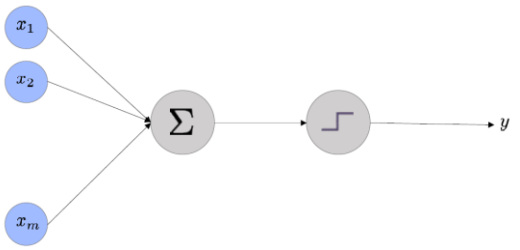
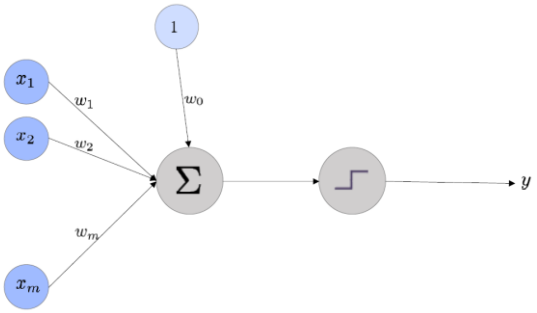
Deep Neural Networks

Single Layer Perceptron

1. **McCulloch & Pitts Neuron Model (1943)**
   - One of the earliest formal models of a neuron, introduced by McCulloch & Pitts.
   - Inputs: $x_1, x_2 \dots, x_m$ (binary inputs).
   - Summation unit ($\sum x$): sums all the inputs values.
   - Activation function: threshold function (step function).
     - Output $y = 1$ if the sum of inputs $> \theta$ (threshold).
     - Outputs $y = 0$ otherwise.
   - No weights or bias term involved.
   - Basic logic gate emulation (AND, OR) .



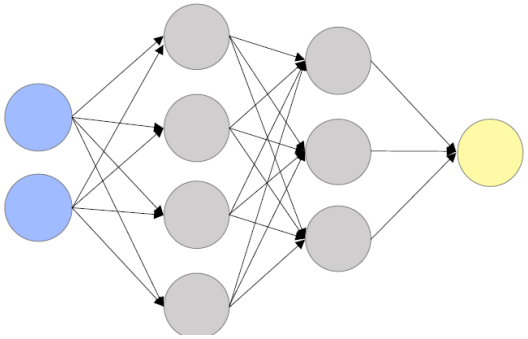
2. **Towards the Single Layer Perceptron**
   - Extension of McCulloch-Pitts model to a weighted model.
   - Inputs are now multiplied by weights $w_1 w_2, \dots, w_m$.
   - Adds a bias term w_0 with constant input $x_0 = 1$.
   - Still uses a threshold activation function.
   - Uses a vector notation $w^T[1, x]$ to include the bias.
   - Output is $y = 1$ if weighted sum $> \theta$, 0 otherwise.



3. **Non-linearity Activation Functions**
   - Non-linearity is essential for learning complex patterns.
   - Activation function: transforms a scalar to another scalar.
   - Common non-linear functions:
     - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$: Squishes input to $[-1,1]$.
     - $\sigma(x) = \frac{1}{1+e^{-x}}$:Smooth map input to $[0,1]$.
     - $\text{ReLU}(x) = \max(0, x)$: output 0 if input$< 0$, otherwise input.
     - $f(x) = \begin{cases} ax & if\ x < 0 \\ x & otherwise \end{cases}$: Allows small negative slope to mitigate “dead” units.

Multi-Layer Perceptron

1. **Neural Network Structure**
   - NNs are formed by chaining neurons.
   - Three types of layers:
     - Input layer (blue): where data enters the network.
     - Hidden layer (gray): one or more layer where intermediate computation happens.
     - Output layer (yellow): produces the final results.
   - Fully connected: each neuron is connected to all the neurons in the next layer.



2. **Width & Depth**
   - Width: the number of neurons in the hidden layer.
   - Depth: number of hidden layers.
   - Greater width and depth gradually allow modeling more complex functions but increases computational cost and risk overfitting.

3. **Forward & Backward Propagation: Neural Networks have two phases**
   - Forward Propagation:
     - Input flows though the layers.
     - A cost (or error) is calculated based on output vs. target.
   - Back Propagation:
     - The error is propagated backward through the network.

- Gradient of weights are computed for optimization (typically via gradient descent).

Forward Propagation

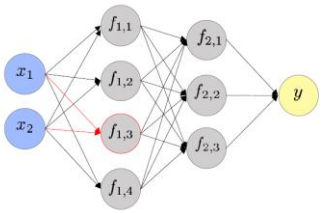
1. **Forward Propagation Overview**
   - Inputs x_1, x_2 are passed into a Neural Network.
   - Each neuron ($f_{1,1}, f_{1,2}, \ldots$) computes a weighted sum of its inputs followed by an activation function.
   - The signal flows from input layer -> hidden layers -> output layer y.
   - This process forms a compositional function, where each layer applies a transformation based on the output of previous layer.

2. **Weighted Notation**
   - Highlights the role of a single connection: from x_2 to $f_{1,3}$.
   - Weights are denoted as $W(layer, in, out)$, e.g., $W(1,2,3)$ is the weight from 2nd input to the 3rd neuron in the first layer.
   - Reinforces the idea that each edge (connection) has its own parameter.

3. **First Layer Node Computation**
   - Focuses on one neuron's computation:

$$f_{1,3}(x) = \sigma\left(w_{(1,:,3)}^{\top} x\right)$$

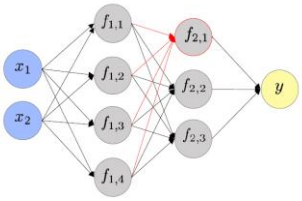
     - Where $w_{(1,:,3)}$ includes the weight for neuron $f_{1,3}$.
   - Shows how to retrieve weights : $[w_{(1,1,1)}, w_{(1,2,3)}]$.
   - Emphasizes forward step: dot product followed by nonlinearity.



4. **Second Layer Node Computation**
   - Now evaluates $f_{2,1}$using the outputs from previous layer:

$$f_{2,1} = \sigma\left(w_{(2,:,1)}^{\top} f_{(1,:)}\right)$$

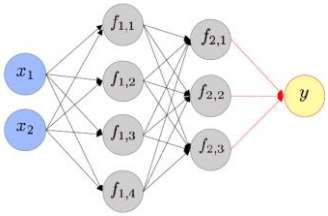
   - Where $f_{(1,:)} = [f_{1,1}, f_{1,2}, f_{1,3}, f_{1,4}]$.
   - Shows a layer-wise computation continues recursively across the network depth.



5. **Output Layer Computation**
   - Final prediction:

$$y = \sigma\left(w_{(o,:)}^{\top} f_{(2,:)}\right)$$

   - Output depends on all second-layer neurons activations $f_{2,1}, f_{2,2}, f_{2,3}$.
   - All computations use same core formula structure: weighted sum → activation.



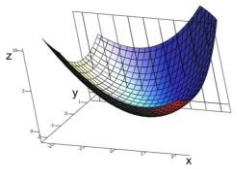
6. **Parameter Count**
   - **The function** f_w maps $R^2 \rightarrow R$ in this example.
   - The total parameters (weights + biases) $23 + 8$.
     - 23 total weighted connections.
     - 8 total biases (one per neuron).
   - This indicates that even a small network can have many parameters to learn.

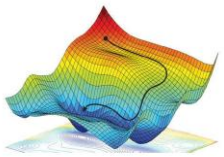
Training a Neural Network

1. **What it means**
   - Training = learning the parameters (weights and biases) that define the function f_w.
   - We seek to minimize the loss: a measure of how far our prediction is from the actual output.
   - Loss function l: $\mathcal{Y} \times \mathcal{Y} \to [0, +\infty)$
     - Takes a predicted output and true output.
     - Outputs a non-negative number representing error.

   - Depending on the task:
     - Binary Classification: $f_w: R^m \to \{-1,1\}$
     - Regression: $f_w: R^m \to R$
     - Multi-class classification: $f_w: R^m \to \{0, \dots, M-1\}$

2. **Training Landscape**
   - Simple convex function like least square or logistic regression.
   - Complex non-convex function real training landscape of Deep networks.
   - Real training involves:
     - Multiple local minima.
     - Saddle points.
     - Risk of getting stuck in suboptimal regions.





3. **Training Objectives**
   - We minimize the average loss over the dataset:

$$w^* = \arg\min_{w \in R^P} \left(\frac{1}{n} \sum_{i=1}^{n} \mathrm{l}\left(f_w(x^i), y^i\right) + \lambda \Psi(f_w) \right)$$

     - P: total number of model parameter (weights + biases)
     - Ψ: optional regularization term.
   - Training set : $S = \{(x^i, y^i,)\}_{i=1}^{n}$
   - Output: the optimal parameter W^*

4. **Optimization Strategy**
   - To find W^*, we use iterative optimization:

$$\boldsymbol{w_t} = \boldsymbol{w_{t-1}} - \gamma \nabla J(S;\, w_{t-1})$$

     - γ: learning rate (step size). It controls the size of each update (too big instability, too small, slow convergence)
     - ∇J: gradient of loss function.
   - Initialization: $\boldsymbol{w_0} = 0$ or random values.
   - Gradient Descent (GD): we iteratively descent the loss surface using local gradient info.
   - The gradient is a vector of partial derivatives over all parameters.

Backpropagation

1. **Introduction:**
   - Backpropagation (1960s): method to optimize the network by updating weights and biases.
   - It uses the gradient of cost function with respect to each parameter.
   - The goal: partial derivative of cost function J.

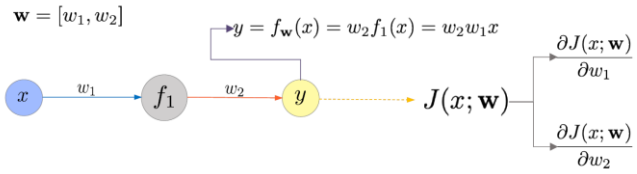
2. **Chain Rule:**
   - Core Idea: chain rule of calculate allows decomposition of complex derivatives paths.
   - Example: $F = f \circ g \circ h \circ u \circ v$

$$\frac{\partial F(x)}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial u} \cdot \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial x}$$

   - This enables computing gradient across multiple layers.
   - The gradient of the entire network is thus built as a product of local gradients using the chain rule.

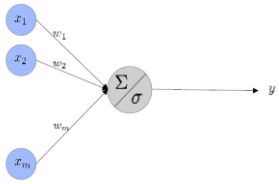
3. **Backprop Example 1**
   - Simple network with:

$$\boldsymbol{y = f_w(x) = w_2 f_1(x) = w_2 w_1 x}$$



   - Compute gradients:
     - $\boldsymbol{\frac{\partial J(x;w)}{\partial w_2} = \frac{\partial J}{\partial f_w} \cdot \frac{\partial f_w}{\partial w_2}}$
     - $\boldsymbol{\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial f_w} \cdot \frac{\partial f_w}{\partial f_1} \cdot \frac{\partial f_1}{\partial w_1}}$
   - This illustrates how each weight affects the final loss.
   - Show explicitly how local derivatives are chained in reverse order through the network.

4. **Backprop Example 2**
   - Single neuron with activation function σ, using square loss:



$$\boldsymbol{J(S; w) = \frac{1}{n} \sum_{i=1}^{n} \left(y^i - \sigma(w^\top x^i) \right)^2}$$

   - Focus on a single training sample for simplicity:

$$\boldsymbol{J^i(w) = \left(y^i - \sigma(w^\top x^i) \right)^2}$$

   - Key simplification: treat the cost for one sample first, generalize later.
   - Reinforce the notion that optimization occurs over a function composed of nonlinearities and weighted sums.
   - Apply the chain rule to compute $\frac{\partial J^i(w)}{\partial w_k}$:

$$\boldsymbol{-2 \left(y^i - \sigma(w^\top x^i) \right) \cdot \sigma'(w^\top x^i) \cdot x_k^i}$$

   - Final results:

$$\boldsymbol{\frac{\partial J^i(w)}{\partial w_k} = -2 \left(y^i - f_\sigma(w^\top x^i) \right) f'_\sigma(w^\top x^i) x_k^i}$$

   - This decomposition shows how each weight's update is called by:
     - The prediction errors.
     - The derivative of the activation function.
     - The input feature value x_k.

Gradient Descent Algorithms

1. **Core Optimization Objective:**

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w} \in R^P} J\,(S;\, \boldsymbol{w})$$

- $\boldsymbol{w} \in R^P$: $\boldsymbol{w}$ is parameter vector with P dimensions. The optimization searches over all real-valued vectors of length P.
- $J(S;\, \boldsymbol{w})$: Objective function (cost or lost function), it depends on training dataset S and model parameters W. It evaluates how the model with parameters W fits the dataset S.
- $\arg\min_{\boldsymbol{w} \in R^P} J$: finds the value of W that minimizes the objective function J.
- $\boldsymbol{w}^*$: this is optimal parameter vector, the one that minimizes the objective function.
- We are searching for the best weights $\boldsymbol{w}^*$ in R^P that minimizes the loss function J, evaluated over the dataset S.

2. **Batch Gradient Descent (GD):**

$$w_t = w_{t-1} - \gamma \nabla J(S;\, w_{t-1})$$

- Uses the entire dataset per update.
- Smooth convergence but computationally expensive.

3. **Stochastic Gradient Descent (SGD):**

$$w_t = w_{t-1} - \gamma \nabla J\left(x^i;\, w_{t-1}\right)$$

- Uses one random sample per update.
- More noise, but better exploration and often escapes local minimum.

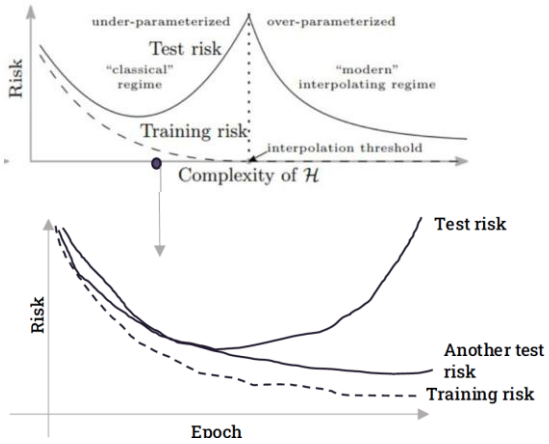
4. **Mini-batch Gradient Descent:**

$$w_t = w_{t-1} - \gamma \nabla J\left(x^{i:i+B};\, w_{t-1}\right)$$

- Compromise between two.
- B: Batch size (16, 32, 64, …).
- SGD may require more iterations but often converges to better minima, while mini-batches offer stability and speed used in practical Also noted that optimization isn't convex, so strategies matter greatly.

5. **Terminology epoch, Batch, iteration**
   - Epoch: One full pass though the training dataset.
   - Batch size: number of samples per gradient updates.
   - Iteration: one update step.
   - Hyperparameters:
     - Batch size
     - Number of epochs
   - You rarely train a model using one epoch; multiple epochs are need to converge.
   - Batch size affect memory and training stability.
   - Early epochs allow larger learning rate, while smaller rates help refine later steps.

6. **Model Selection in Deep Learning**
   - Classical method: tune hyperparameters using cross-validation.
   - In practice:
     - Fix network structure and complexity beforehand.
     - Tune using validation error over epochs.
   - Risk Vs Epoch:
     - Training risk decreases with more training.
     - Test risk might increase after a point → overfitting.
   - Complexity (H) = number of layers, neurons, and model depth/width.
   - Deep models are harder to evaluate because:
     - Training the computationally expensive.
     - Complexity can be high even for small networks.
   - Model complexity is implicit in the network architecture (number of layers, units) which influences generalization ability.



7. **Risk/Complexity plot (Double Descent):**
   - Training Risk (Dashed line): As the model complexity increases, the training Risk decreases, A more complex model has greater capacity to fit training data eventually reaching zero or near-zero when the model perfectly memorizes the training dataset.
   - Test Risk (Solid line):
     - Classic Regime: The test risk decrease, aligning with classical bias and variance trade-off.
     - Peak at interpolation threshold: As the model complexity reaches a point that it can perfectly fit the training data, (zero training error) the test risk often picks.
     - Modern Interpolation Regime(over-parameterized): Surprisingly, as the model capacity increases passing the interpolation threshold into a highly over-parameterized region (where the number of parameters is much larger than the number of training samples), the test risk starts to decrease again. This "second descent" is key characteristic of double descent phenomenon.

8. **Risk/epoch Plot (Empirical Observation):**
   - Represents the actual behavior of DNNs.
   - Training Risk: continues to decrease as the epoch increases.
   - Test Risk: the classical U-shape (The model begins to memorize the training data and fails to generalize).
   - Another Test Risk: Indicates better generalization and resistance to overfit.

9. **Preventing Overfitting:**
   - Classical methods are still useful:
     - Regularization term (L2 norm penalties on weights).
     - Early stopping based on validation loss trends.
   - Dropout:
     - Randomly deactivating units during training.
     - Prevents co-adaptation of neurons and forces redundancy in representation.
     - Overfitted DNN tend to suffer from a problem of co-adaptation:
       - Model's weights are adjusted co-linearly to learn the training data too well so the model doesn't generalize.

10. **Bagging and Dropout:**
    - Bagging (Bootstrap Aggregation):
        - Involves training multiple sub-models.
        - Each sub-model is trained independently.
    - During Inference of Bagging:
        - Predictions are aggregating via majority voting (classification) or averaging (regression).
    - It reduces the variance and helps generalization.
    - It is computationally expensive especially in DNN.
    - Dropout: A practical approximation of bagging.
        - At each training step, a random fraction of neurons is dropped (set to zero).
        - This is done independently across training iterations.
        - It avoids needing to train multiple complete models.
        - It injects noise during training, which forces the network to be more robust.
    - Dropout prevents units from co-adaptation and encourages redundancy in the learned representation.
    - At test time, all neurons are activated, but the outputs are scaled to account for dropout during training.
    - Why Dropout Works:
        - Encourages neurons to be useful independently.
        - Prevents over-reliance on specific connections or co-adapted units.
    - Dropout acts as an implicit regularizer:
        - No need for explicit $l1, l2$ penalties.

More details about deep networks training

1. **Loss Function – Regression vs Classification**
    - Regression: Square Loss (for problems with real valued targets):

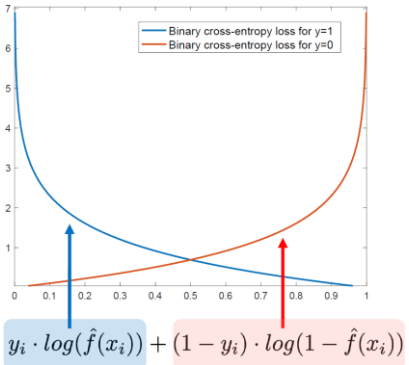
$$\mathrm{l}\left(y_i, \hat{f}(x_i)\right) = \left(y_i - \hat{f}(x_i)\right)^2$$

        - Penalizes large deviations quadratically.
    - Binary Classification: Cross-Entropy Loss:
        - For $y_i \in \{0,1\}$, we use:

$$\mathrm{l}\left(y_i, \hat{f}(x_i)\right) = -y_i \log\left(\hat{f}(x_i)\right) - (1 - y_i) \log\left(1 - \hat{f}(x_i)\right)$$

        - Strongly penalizes wrong confident predictions:
        - If $y_i = 1$ and $\hat{f}(x_i) \to 0$: $loss \to \infty$.
        - If $y_i = 0$ and $\hat{f}(x_i) \to 1$: $loss \to \infty$.
    - The Classification Loss must be asymmetric and bounded below but unbounded above to effectively penalize wrong predictions.
    - Cross-entropy is preferred over square loss in classification because it better aligns with probabilities interpretation and gradient behavior.

2. **Visualizing Binary Cross-Entropy Loss**
    - The loss curves for:
        - $y = 1$: decreases as $\hat{f}(x_i) \to 1$.
        - $y = 0$: decreases as $\hat{f}(x_i) \to 0$.
    - The plot shows a sharp increase in loss as predicted probabilities diverge from the true label.
    - Confidence calibration is critical. The more confidence you have on wrong direction, the higher the penalty and important feature minimizing misclassification risk.



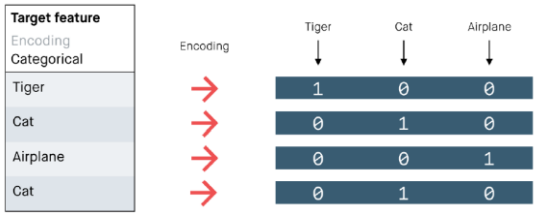
3. **Multi-class Classification – Categorical Cross-Entropy**
   - For M classes and one-hot encoded target Y_i, the loss is:

$$l\left(y_i, \hat{f}(x_i)\right) = -\frac{1}{M}\sum_{k=1}^{M} y_i^k \, lo\, g\left(\hat{f}(x_i)^k\right)$$

   - Often simplified since only one $y_i^k = 1$, others are 0:

$$l = -\log\left(predicted\ probability\ for\ correct\ class\right)$$

   - Required the model output to be a probability distribution, enforced via softmax.



4. **Activation Function for Output Units**
   - Binary classification: Use sigmoid activation:

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

     - Squashes outputs to (0,1), interpretable as probabilities.
   - Multi-class Classification: Use Softmax

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{M} e^{z_k}}$$

     - Ensures the output is valid probability distribution (all positive, sums to 1).
   - Sigmoid ↔ Binary Cross-entropy
   - Softmax ↔ Categorical Cross-entropy

Convolutional Neural Networks

A Refresh

1. **Introduction and Motivation for CNNs:** Dense networks connect every input to every neuron in the subsequent layer — this means:
   - They require a large number of parameters.
   - They do not leverage any structure in the input data (like locality in images).
   - Flattening an image into a 1D vector destroys spatial relationships.
   - CNNs are designed specifically to handle grid-like topology in data, especially image, by preserving spatial structure and reducing parameter count.
2. **Two-part structure of CNNs:** The typical CNN architecture has two conceptually distinct parts:
   - Feature extractor (Representation Learning):
     - Composed of convolutional layers, activation functions, and often pooling.
     - Learns hierarchical features from the input data.
   - Task Learner (Classifier):
     - Usually made of fully connection (dense) layers followed by softmax (for classification).
     - This is where decision-making happens based on extracted features.
   - This separation helps us understand the learning pipeline better: first extract what's useful, then classify based on it.

3. Dense vs Convolutional Layers
   - Dense Networks:
     - Every neuron is connected to all inputs.

- Do not scale with high dimensional inputs like images (even a tiny 32×32×3 image leads to >3,000 weights per neuron).
- Do not exploit spatial locality.

- Convolutional Layers:
  - Learn local features using kernels (filters) that slide over the input.
  - Use parameter sharing: the same kernel is applied across entire input.
  - Require fewer parameters, are computationally cheaper, and capture spatial hierarchies in data.

Interlude: convolution

1. Mathematical Formulation of Convolution and Cross-Correlation

- Convolution (Mathematical definition):

$$s(i,j) = (K * I)(i,j) = \sum_m \sum_n I(m,n)\ K(i-m, j-n)$$

- Equivalently (Convolution with Flipped Kernel):

$$s(i,j) = \sum_m \sum_n I(i+m, j+n)\ K(m,n)$$

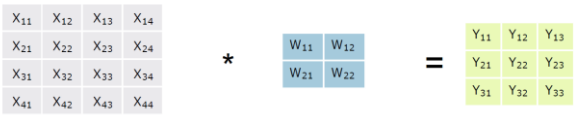
  - In convolution, the kernel is flipped along both axes before applying it.
- Cross-Correlation (used in practice):

$$s(i,j) = \sum_m \sum_n I(i-m, j-n)\ K(m,n)$$

  - In cross-correlation, we simply slide the kernel across the input and take the weighted sum (no flipping).
- While we often call it "convolution", what's actually implemented in CNN is cross-correlation, a simplified
- version where the kernel is not flipped.

2. Cross-Correlation (With an example).

- The image shows a 4×4 input matrix X, A 2×2 kernel W, and the resulting 3×3 output matrix Y.
- Operation: at each valid position of the kernel, we compute: $Y_{ij} = \sum_{m,n} X_{i+m,\, j+n}\ W_{mn}$.
- This Operation slides the kernel from top-left to bottom-right, computing a weighted sum at each location.
- Each kernel detects specific patterns.
- Output values represent activations: how strongly kernel matched that region of image.

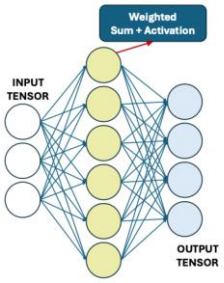


$$Y_{11} = X_{11}W_{11} + X_{12}\,W_{12} + X_{21}\,W_{21} + X_{22}\,W_{22}$$
$$Y_{12} = X_{12}W_{11} + X_{13}\,W_{12} + X_{22}\,W_{21} + X_{23}\,W_{22}$$
$$Y_{13} = X_{13}W_{11} + X_{14}\,W_{12} + X_{23}\,W_{21} + X_{24}\,W_{22}$$

3. A Sketch of Dense Layers:

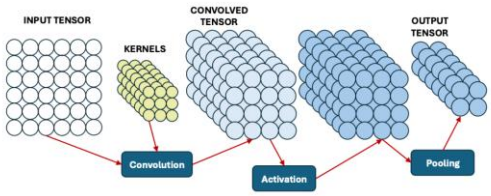
- Every input neuron connects to every neuron in the next layer.
- Each connection has a learnable weight.
- The output of each neuron is:

$$a = \phi\left(\sum w_i x_i + b\right)$$



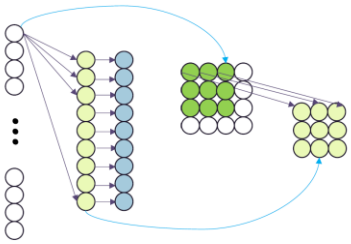
- Where ϕ is activation function (ReLU, Sigmoid).
- Problems: high parameter count, specially with high dimensional input like images.
- No spatial awareness: relationship between neighbors are lost after flattening.

4. **Sketch of Convolution layer (Flow Diagram)**



- Input Tensor (Image).
- Convolution: Apply kernel over the input.
  - Output of convolution is also called linear activation.
  - It highlights the presence of learned features.
  - The same kernel slides over all input region. Which enables translation equivariance the same pattern is detected regardless of its position (Parameter Sharing).
- Activation: Apply non-linearity (ReLU).
- Multiple feature detectors can be used to capture difference image properties(their number is called channels).
- Pooling: Downsample to reduce dimensionality.
- Output Tensor: the processed result.
- We don't use a single kernel; we use a bank of kernels and the output is cuboid.
- Each step serves a specific purpose:
  - Convolution: Feature extraction.
  - Activation: Non-linearity to model complex patterns.
  - Pooling: Size reduction + invariance to shifts.

5. **From Dense to Sparse Interaction**



- In dense layers: every output depends on all inputs.
- In convolution: each output depends on a small local region: this is called sparse interaction.
- With dense networks you are combining everything with everything. CNNs allow you to be more selective.
- This sparse is what makes CNNs computationally efficient, especially for large inputs like high-res images.

6. **Output Feature Size of Conv Layers**

$$O = \frac{W - K + 2P}{S} + 1$$

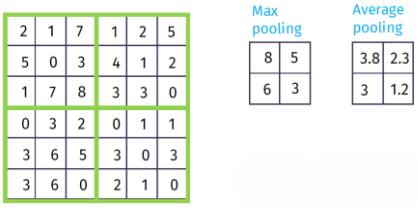
- Channels (K): number of kernels
- Stride (S): step size of kernel sliding
- Padding (P): how much border we add
- Input Width (W)

7. **ReLU Activation:**
- ReLU (Rectified Linear Unit) is the most commonly used activation:

$$f(x) = \max(0, x)$$

- ReLU introduces non-linearity, keeps the positive values, and sets negatives to zero.
- We apply activation function after weighted sum to introduce non-linearity.

8. **Pooling (Downsampling):**
- Pooling (typically max or average) is used to:
  - Reduce feature map size.
  - Make model robust to small shifts/transformations.
- Pooling helps reduce dimensionality and provides invariance to small shifts of the inputs.
- Pooling layer (especially average pooling) simplify backpropagation, they are linear and distributive.



Backpropagation in CNN

1. **Forward Flow:**
   - Input x_{ij} values from a 2D image patch.
   - Convolution:

$$z_{r,c} = \sum_{i=1}^{3}\sum_{j=1}^{3} w_{ij} \cdot x_{r+i-1,c+j-1}$$

     - Each output value $z_{r,c}$is a linear combination of inputs with filter weights.
   - Activation:

$$v_{s,t} = f_{\sigma}(z_{s,t})$$

     - Apply a non-linearity, like ReLU, to get activated feature map.
   - Pooling:

$$v = \frac{1}{4}\sum_{s=1}^{2}\sum_{t=1}^{2} v_{s,t}$$

     - Simple average pooling reduces dimensionality.
   - Classification:

$$\hat{y} = w \cdot v$$

     - Fully connected classifier maps the pooled features to predictions.

   - Loss:

$$J(\boldsymbol{w}) = \frac{1}{N}\sum_{k=1}^{N}(\widehat{y_k} - y_k)^2$$

     - We use mean square error (MSE) in this example.
   - So when we're talking about learning in CNNs, we mean learning the weights in both convolution filters and fully connected layers, and we do that by minimizing the loss using backpropagation.

2. **Gradient Calculation.**
   - Goal: compute gradient of loss weights:

$$\nabla J_k(W) = \begin{bmatrix} \frac{\partial J_k}{\partial w_{1,1}} \\ \vdots \\ \frac{\partial J_k}{\partial w_{3,3}} \end{bmatrix}$$

   - The update is based on chain rule:

$$\frac{\partial J_k(w)}{\partial w} = \frac{\partial J_k}{\partial \widehat{y_k}} \cdot \frac{\partial \widehat{y_k}}{\partial v}$$

   - Each step decomposes how loss changes propagate backwards from output to pooled features to kernel weights.
   - In other words once we compute the total risk, the only way to optimize this complicated function is by resorting to an iterative procedure like gradient descent. We initialize the kernel randomly then forward and backward passes update the weights step by step.

3. **Full Backpropagation Chain for Kernel Weights**
   - Kernel weight $w_{r,c}$:

$$\frac{\partial J_k(w)}{\partial w_{r,c}} = \frac{\partial J_k}{\partial \widehat{y_k}} \cdot \frac{\partial \widehat{y_k}}{\partial v} \cdot \sum_{s=1}^{2}\sum_{t=1}^{2} \frac{\partial v}{\partial v_{s,t}} \cdot \frac{\partial v_{s,t}}{\partial f_\sigma(z_{s,t})} \cdot \frac{\partial f_\sigma(z_{s,t})}{\partial z_{s,t}} \cdot \frac{\partial z_{s,t}}{\partial w_{r,c}}$$

     - $\frac{\partial J_k}{\partial \widehat{y_k}}$: how much the output affects the loss.
     - $\frac{\partial \widehat{y_k}}{\partial \boldsymbol{v}}$: Derivative of linear layer (just $\boldsymbol{w}$).
     - $\frac{\partial \boldsymbol{v}}{\partial v_{s,t}}$: from pooling, average pooling means this $= \frac{1}{4}$.
     - $\frac{\partial v_{s,t}}{\partial f_\sigma(z_{s,t})}$: identity of $f = ReLU$.
     - $\frac{\partial f_\sigma(z_{s,t})}{\partial z_{s,t}}$: ReLU $=1$ if $z > 0$, else 0.
     - $\frac{\partial z_{s,t}}{\partial w_{r,c}}$: This depends on input x it's $x_{r+i,c+j-1}$. This is core mechanism: CNN kernel gradient are based on the input values they overlap.

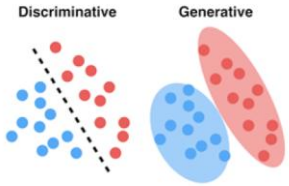
   - The function is complicated a composition of many operations. But all steps are differentiable. So we apply chain rule and update weights through backpropagation.
   - First we backpropagate from the loss to the classifier weight, then to the pooled feature, then to each activation, and finally to the kernel weights.
   - Each value inside the kernel is update using the gradient computed from the loss.
   - In other words:
     - Kernels extract local patterns.
     - Pooling adds spatial invariance.
     - Fully connected layers classify.
     - Backpropagation ties it all together.
   - Every operation has a derivative. Training is just applying the chain rule repeatedly to minimize the loss.
   - Convolutional backpropagation can be expressed using convolutions, allowing the reuse of optimized convolution operation frameworks like Pytorch.

Autoencoders

Introduction

1. **Changing the framework: From Supervised to Unsupervised Learning**
   - In this section, we start with a conceptual shift, from supervised to unsupervised learning.
   - In supervised learning, each point in the training data is associated with a corresponding output or label, enabling the model to learn a mapping from inputs to outputs. This is ideal for tasks like classification and regression where label data is available.
   - However, in real-world there are many scenarios where obtaining labeled-data is expensive, time-consuming, or simply impractical.
   - In this situations unsupervised learning is one alternative method where the dataset consists solely of inputs X and the goal is to understand or model the underlying pattern or distribution of data itself without relying on labels.

2. **Generative vs. Discriminative models:**
   - Discriminative models estimate $p(Y|X)$, the probability of a label Y given input X. These are typically used for supervised tasks.
   - Generative models estimate $p(X)$, the probability distribution of the inputs themselves. They aim to understand how data is generated.
   - In generative models we're interested not just in classifying, but in understanding the underlying distribution of the input data, often with goal of generating new data resemble the original.



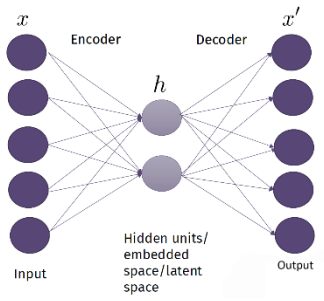
3. **Goals of Generative Modeling**
   - Density Estimation: Learn the probability distribution of the input data.
   - Representation Learning: Discovering lower-dimensional embeddings that captures the essence of the data.
   - Data Generation: Producing novel data points from the learned distribution.
   - Generative models like GAN, diffusion models, and autoencoders are powerful for these goals.

Understanding Autoencoder

1. **What is an Autoencoder:**
   - Autoencoders are a class of unsupervised neural networks that aim to learn a compressed, lower-dimensional representation (called the latent space) of input data and then reconstruct the original data from this representation.
   - This process involves no labels just raw data x.
   - The autoencoder consist of:
     - Encoder $h = f(x)$, maps the input to a latent space.
     - Decoder $x' = g(h)$, reconstruct the input from the latent representation.
   - Although this method is unsupervised, it mimics supervised training because the target output is the input itself. Thus the network can be trained using the usual forward propagation, loss computation, and backpropagation strategy.

2. **Motivation and Applications**
   - Historically autoencoders were used for:
     - Dimensionality reduction.
     - Feature learning.
   - Recently they evolved to support:
     - Generative modeling, where the decoder is used to produce new data.
   - Autoencoders can serve as a backbone for other applications like transfer learning, style transfer, and more. The latent space encodes only the essential information, which is the key to both understanding and generating data.

3. **Visual Structure and Flow**



- The architecture of a basic autoencoder can be summarized as the diagram.
- the goal is to make the reconstruction x' as close as possible to the original input x.
- This leads us to define a reconstruction loss, commonly mean square error (MSE):

$$L(x, x') = |x - x'|^2 = \sum_i (x_i - x_i')^2$$

  - This loss guides the training via forward and backward propagation.

4. **Connection to PCA**
   - If we remove non-linearities and use MSE as a loss function, autoencoders resemble Principal Component Analysis (PCA).
   - However, there are key differences:
     - PCA enforces orthogonality among the learned directions (components), while autoencoders do not.
     - PCA directions maximize variance, whereas autoencoders may not impose this constraint unless explicitly designed.
   - PCA transforms the data into a new basis maximizing variance, but autoencoders learn this basis implicitly, often non-linearly, without assuming orthogonality or equal variance.

5. **Under complete Autoencoders: Compressing for insight**
   - To encourage the autoencoder to learn meaningful, compact representations, we often impose a bottleneck by reducing the size of the latent space (making $dim(h) < \dim(x)$). This is known as undercomplete autoencoder.
   - This restriction:
     - Focuses the model to prioritize salient features.
     - Prevent trivial solutions like copying the input directly.
     - Promotes generalization and feature extraction.
   - If the latent space has to much capacity, the network may just memorize the input. With fewer dimensions, it is forced to extract the essential patterns to reconstruct the input.

6. **Beyond Deterministic Models: Probabilistic Encoders and Decoders**
   - Moving from deterministic mappings to probabilistic formulations helps extend autoencoders into generative models (like Variational Autoencoders – VAEs). Instead of computing exact mappings:
     - The encoder now defines a probabilistic distribution $p(h|x)$.
     - The decoder defines $p(x|h)$.
   - This means:
     - The latent code is a distribution, not a fixed vector.
     - Training involves maximizing the expected log-likelihood of the reconstruction:

$$\text{Loss} = E_{p(Z \mid X)}[\log P\ (x \mid z)]$$

   - Modeling each sample as a point in latent space without regularization can cause issues: nearby points my decode into completely different outputs. This motivates introducing regularization in probabilistic autoencoders, to ensure continuity and structure in latent space.

7. **Limitation without Regularization:**
   - Without explicit regularization, two nearby points in the latent space may:
     - Not decode similar outputs.
     - Map to meaningless outputs if the region in latent space was never "visited" during training.
   - This leads to:
     - Poor interpolation quality.
     - Unreliable data generation from random latent samples.

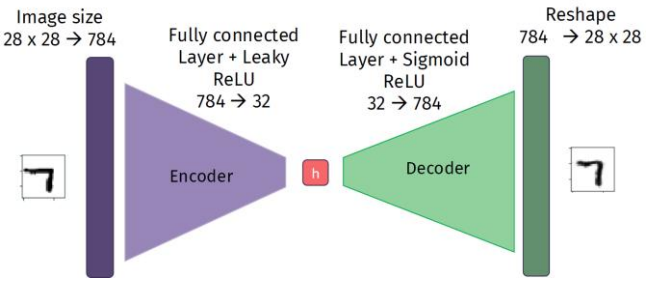
- Sampling from randomly structed latent space (with no constraints) often yields garbage outputs. We need more structured generative models.

How can we use Latent Space

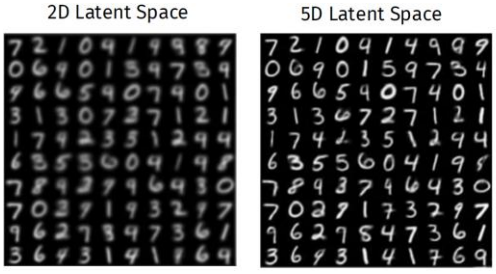
1. **Core Idea:**
   - Autoencoders are more than just reconstruction tools they are powerful representation learners.
   - Once trained, the latent space (also called embeddings) becomes a compressed, information-rich encoding of the input.
   - Instead of using the full input, we can now operate on the latent representation.
   - What we need is not reconstruction, but the representation here.
   - The latent space captures the essential aspects of the data and becomes a new input for downstream task.

2. **Practical Use of Embeddings:**
   - As features for classical machine learning: After training we can discard the decoder and use the encoder as a feature extractor. The latent vector $h = f(x)$ can now serve as input for:
     - SVMs, Logistic Regression, KNN, etc.
     - Tasks like classification, clustering, or similarity retrieval.
   - Transfer Learning: we can train an autoencoder on a large amount of unlabeled data (easy to collect), then use the encoder as a backbone and fine-tune it on a smaller labeled dataset.
     - This is similar to what's done with ImageNet-pretrained CNN in vision tasks.
   - Visualization: For very low-dimensional latent spaces (2D, 3D), the embeddings can be visualized, revealing structure in data.
     - Similar inputs are close in latent space.
     - Can be plotted and interpreted.

3. **Working Example: MNIST Autoencoder**
   - The reconstruction digits are usually close to the originals proof that the network learned a good latent representation.
   - Even with a significant reduction (from 784 to 32), the reconstruction was surprisingly accurate, showing the compression power of the autoencoder.



4. **Latent Space Dimensionality and Reconstruction Quality:**
   - 2D: More blur, loss of detail.
   - 5D: sharper reconstruction, better digit definition.
   - Smaller latent space: greater compression, more loss.
   - Larger latent space: better reconstruction, risk of overfitting.
   - If the latent space has too much capacity, the network might just memorize the inputs. But the with tight compression, it must focus on what matters the most.



5. **Convolutional Autoencoder:**
   - Better suited for image data.
   - The encoder learns spatially-aware filters via CNNs.
   - The decoder uses transpose convolutions (deconvolution) to reconstruct.
   - The output retains finer image features compared to basic fully connected layers.

6. **Regularized Autoencoders:**
   - Sometimes the latent space might be:
     - Too large (overcomplete).
     - Or not structured enough for downstream use.

- This motivations regularization, which modifies the loss function to enforce structure.

7. **Two common Regularization:**
   - Sparsity Regularization: encourages most neurons in h to be inactive (zero):

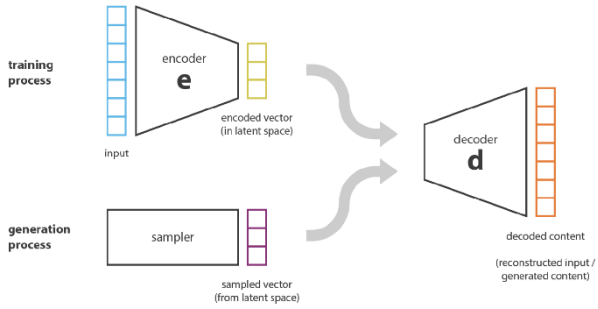
$$L\left(x, g\left(f(x)\right)\right) + \lambda|h|_1$$

     - This pushes the model to use only the most essential features.
   - Smoothness Regularization: penalizes sudden changes in latent space with respect to small input variations.

$$L\left(x, g\left(f(x)\right)\right) + \lambda \sum_i |\nabla_x(h_i)|^2$$

     - This unforces the encoder to produce stable, smooth embeddings.
   - These regularization help structure the latent space, so similar inputs yield similar latent vectors, and interpolations behave predictably.

8. **Autoencoders as Generators:**
   - Process:
     - Train encoder-decoder pair as usual.
     - Discard encoder.
     - Sample random vectors from latent space.
     - Feed these into the decoder to generate new data.
   - Challenge: this doesn't work well why?
     - Because latent space has no imposed structure.
     - Sample points may be far from real encoded input: output might be garbage.



   - You can't just sample arbitrary points from latent space. The decoder doesn't know what to do with them unless the latent space is properly structured.

Variational Autoencoder

1. **Motivation: Why move beyond classic Autoencoders? In traditional autoencoders:**
   - The latent space lacks explicit structure.
   - Sampling from it (for generation) often fails to produce coherent data.
   - There's no guarantee that nearby points in latent space will decode a similar or even valid samples.
   - Sampling arbitrary vectors from an unstructured laten space give us junk. There's no organization and no guarantee of meaning.

2. **Solution: Variational Autoencoders (VAEs):**
   - Introducing structure to the latent space.
   - Modeling latent representation as probability distribution instead of fixed vectors.
   - Use regularization to shape the latent space (Usually resembles a unit Gaussian).

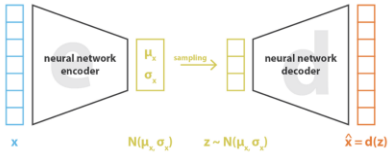
3. **What's the key difference?**
   - Classic Autoencoders:

$$x \to f(x) = h \to g(h) = x'$$

     - Deterministic Encoding.
     - Latent space is arbitrary.
   - Variational Autoencoders:

$$x \to distribution\ p(z \mid x) = \mathcal{N}(\mu_x, \sigma_x^2) \to z \sim p(z \mid x) \to x' = d(z)$$

   - Encoder outputs a mean μ_x and variance σ_x^2.
   - A latent variable z is sampled from distribution.
   - The decoder reconstructs the input from z.
   - The encoder learns to predict a region in latent space instead of a single point. This gives use both generativity and continuity.



4. **The VAE Loss Function:**

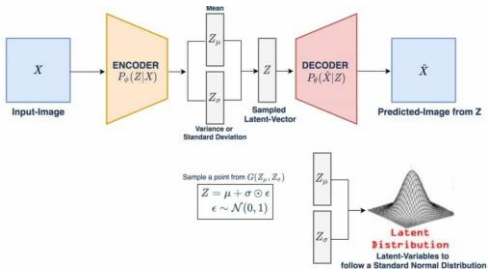
$$\mathcal{L}(x, x') = |x - x'|^2 + \mathbf{KL}\big(\mathcal{N}(\mu_x, \sigma_x^2) \,|\, \mathcal{N}(0,1)\big)$$

   - Reconstruction loss: ensures decoded output is similar to input.
   - KL divergence: forces the learned distribution to resemble the standard normal distribution.
   - This regularization discourages the encoder from using arbitrary cluster in latent space. Instead, it aligns latent codes with a known, smooth distribution.

5. **Visualizing the Architecture**
   - Encoder learns μ_x and σ_x^2.
   - A reparameterization trick is used:

$$z = \mu_x + \sigma_x \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0,1)$$

     - This trick allows backpropagation through sampling step.
   - Decoder uses z to reconstruct the input.



6. **What does the KL term do? Regularization through KL divergence enforces two properties:**
   - Continuity:
     - Similar latent codes decode into similar outputs.
     - Helps in smooth interpolation.
   - Completeness:
     - Every point in latent space corresponds to meaningful outputs.
     - Prevents holes in latent space where decoding fails.
   - Without KL, there's no guarantee that interpolation between latent points will produce coherent transitions.

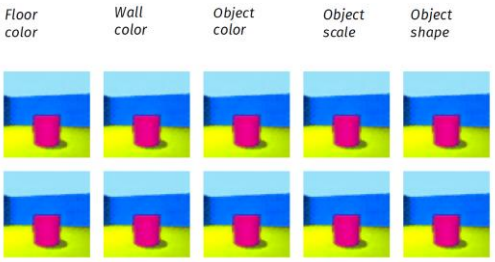
7. **Comparison: Autoencoder vs VAE Latent Representations**
   - Autoencoder: Latent space is scattered and unorganized.
   - VAE: Latent clusters are compact, well-separated, and centered—suitable for interpolation and generation.
   - This difference emerges entirely due to the KL regularization.

Disentanglement Learning

1. **What Are Disentangled Representations?**
   - It aims to separate out the independent and meaningful factors of variation (FoVs) in the data. The goal is to learn a latent representation h such as:
     - A change in a single factor in the data leads to a change in only one component of h.
   - For example if an image changes only in object color, the corresponding latent feature changes, while all others remain constant.

2. **Visual Intuition:**

FoV	What Varies
Floor color	Background floor hue
Wall color	Background wall hue
Object color	Object hue
Object scale	Object size
Object shape	Geometry (e.g., cube vs. sphere)
Camera position	Perspective

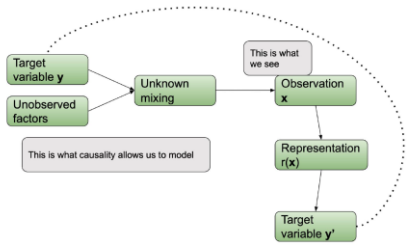


   - In a perfectly disentangled latent space, each of these factors would correspond to a distinct dimension in h.

3. **Properties of Disentangled Representations**
   - Modularity: Each factor affects only one part of the representation.
   - Compactness: Ideally, each factor controls one dimension.
   - Completeness: All factors of variation are present and represented.
   - This structure makes the representation interpretable and reusable in multiple tasks.

4. **Why do we care? (Motivation and Causality)**
   - Disentanglement aligns closely with casual reasoning, in casual models:
     - Variables are independently manipulable (Interventions).
     - Disentangling corresponds to isolating causal modules.

5. **A Parenthesis on Causality (Diagrammatically):**
   - We observe x (mixed influences from latent variables and unknown mixing).
   - Disentangled Representation $r(x)$ allows us to recover independent causal factors, useful for predicting outcomes or adapting to new tasks.
   - This improves:
     - Generalization across tasks.
     - Sample efficiency, because modules can be reused.



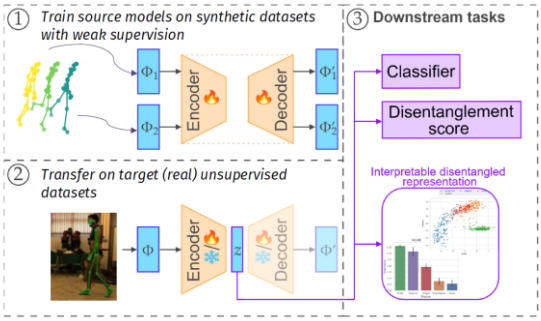
6. **Beta-VAE: enforcing Disentanglement**
   - The Beta-VAE is a key model to achieve disentanglement. It modifies VAE objectives:

$$\mathcal{L}(x, x') = E_{q(z \mid x)}[\log p\,(x' \mid z)] - \beta\,\mathbf{KL}\big(q(z \mid x) \mid \mathcal{N}(0,1)\big)$$

   - $\beta > 1$: emphasizes regularization over reconstruction accuracy.
   - Higher β leads to simpler, more factorized representation, favoring disentanglement.
   - The higher the β, the more pressure you put on learning independent latent factors. We sacrifice reconstruction fidelity for interpretability and modularity.
   - Trade-off: Reconstruction vs. Disentanglement
     - $\beta = 1$: Standard VAE, good reconstruction, less disentanglement.
     - $\beta > 1$: Prioritize structure in latent space, better disentanglement, worse reconstruction.
     - Overly high β may result in a too compact representation, missing finer details in reconstruction (blurred digits).

7. **Example: Human Pose Representation**
   - Input: Skeletal human pose data (joint positions).
   - Goal: Learn a representation where each latent feature controls a body part (arm, leg, …).
   - We fix all features in z and alter only one to see what it controls.
   - If latent space is disentangled, only the related body part moves.
   - Some features may appear "silent" (no effect); a sign of overcomplete representation or redundancy.

8. **How to train Disentangled Representation?**
   - Loss Function:
     - Reconstruction loss + weighted KL divergence.
     - Optionality, include explicit disentanglement penalties.
   - Weak supervision:
     - Use image pairs that differ in only one FoV.
     - Design contrastive or triple-based loss.
   - Transfer Learning Approach:
     - Train a synthetic data where factors are known.
     - Transfer to real, unsupervised domains and preserve disentanglement.

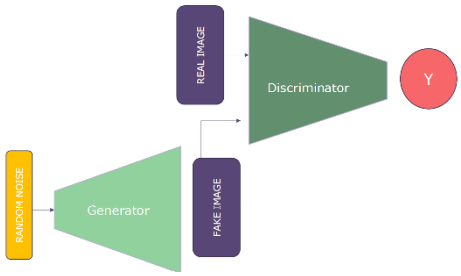


GAN

Introduction

1. **Generative Adversarial Networks (GANs):**
   - GANs are a class of models whose core purpose is to create new, synthesis data that resembles the given training data.
   - At the heart of GANs lies a dual between two neural networks: the generator and discriminator, engaging in a dynamic, adversarial training process.
   - The generator learns to produce data starting from random noise.
   - The discriminator acts as a critic, it evaluates incoming data and predicts whether its "real" or "fake".
   - We learn that this process is supervised for the discriminator real or fake samples are labeled accordingly during training, however the generator does not have direct access to the real data; it improves solely through the feedback provided by the discriminator via backpropagation.
   - The ideal end state equilibrium is achieved when the discriminator can no longer distinguish between real and fake samples, resulting in classification accuracy of 50%. This implies that the generator has successfully lean the true data distribution.

2. **GAN Architecture**



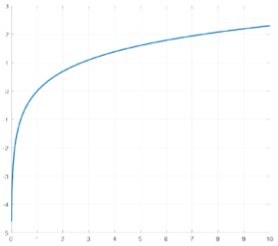
- Random Noise → Generator:
  - The generator takes in a random noise vector.
  - It transforms this noise into synthetic image.
  - At beginning of training, this output is nonsensical due to random weights.
- Generator Output → Discriminator:
  - The fake image, alongside the real image from the dataset are fed into the discriminator.
  - The discriminator attempts to classify each to real or fake.
- Discriminator Output : a label (real/fake) with a probability score (y).
- Adversarial Training Objective:
  - The discriminator tries to maximize the classification accuracy (maximize log $(D(x))$ and log $(1 - D(G(z)))$.
  - The Generator tries to minimize the discriminator's ability to tell fake from real (minimize log $(1 - D(G(z)))$), fool it.

Loss Function and Convergence

1. **GAN Objective Function and training procedure**

$$\min_G \max_D V(D,G) = E_{x\sim p_{\text{data}}(x)}[\log D(x)] + E_{z\sim p_z(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

- This is the core loss function of GANs. It defines the interaction between generator G and the discriminator D as a two-player minmax game:
  - Discriminator's goal: maximize the value function $V(D,G)$.
    - Assign high confidence to real samples: $D(x) \to 1$.
    - Assign low confidence to fake samples: $D(G(x)) \to 0$.
  - Generator's goal: minimize the same function:
    - Fool the discriminator: make $D(G(x)) \to 1$.
- Thus the generator indirectly maximizing $\log D(G(x))$, even though this expression is wrapped inside a minimization in the value function.
- Visual Insight:
  - The curve illustrates the function $\log(1 - D(G(x)))$. It shows when $D(G(x))$ is near zero, the discriminator is confident its fake.
  - The gradient is strong providing useful feedback to the generator. But as $D(G(x)) \to 1$, gradient vanish (saturate), which makes training harder.



2. **GAN Convergence and Training Dynamics:**
   - Key point at convergence: the generator's samples are indistinguishable from real data, and the discriminator's outputs 0.5 everywhere.
   - At this points the discriminator is no better than a coin flip; it cannot distinguish between real and fake data.
   - This marks the Nash Equilibrium of the GAN game: no player (generator or discriminator) can improve its outcome.
   - Theoretical Interpretation: Convergence is reach when the actions of one of the players do not changing depending on the actions of the other.

Training GAN

1. **Initial Training Phase:**

- Generator: Three orange dots represent the generator's initial outputs. These are randomly distributed because both the input noise and the generator's parameters are randomly initialized.
- Discriminator: Yellow (generated), Blue (real data), Bars (Discriminators confidence at each point).
- At this early stage, the generator is far from the real data distribution. It samples are scattered
- discriminator has an easy job. Confidently assigns high probabilities to blue points and low probabilities to orange ones.

2. **First Update:**
   - The generator hasn't changed much yet. Outputs are still random and far from the real data distribution.
   - The discriminator adjusts based on feedback. Bars shift slightly, reflect updated confidence scores, however it is still easy to separate real from fake.
   - This step shows how initial feedback starts flowing back to the generator. The discriminator continues to sharpen its predictions. This process is what triggers the gradient signals necessary for improving the generator.

3. **Generator Improves:**
   - Fake samples (orange) start clustering around real samples (blue), indicating that the generator has begun learning the correct data distribution.
   - Confidence bars now begin to align across real and fake samples. The discriminator's certainty is challenged—this is visible through the flattening of bars.
   - Now it [the generator] knows where the real samples are… it tries to approximate better the places where the real data are expected to be.

4. **Advanced Generator Performance**
   - The orange (generated) and blue (real) points now visibly overlap.
   - Discriminator becomes uncertain. All confidence bars start clustering around a probability of 0.5—indicating it is unsure whether a sample is real or fake.
   - This moment is crucial: the generator is effectively fooling the discriminator. The quality of fake data has drastically improved, and the discriminator's role becomes harder. This exchange is central to the adversarial training paradigm.

5. **Equilibrium (Convergence)**
   - Generator's outputs (orange) now almost exactly overlap with the real data points (blue).
   - The bars vanish—meaning the discriminator cannot confidently classify anything as real or fake. It assigns probabilities close to 0.5 across the board.
   - This reflects the ideal outcome of GAN training: a Nash equilibrium where neither the generator nor the discriminator can improve unilaterally. The discriminator achieves only chance-level performance—indicating success.