Week 11

Autoencoders & Variational Autoencoders

(Explained for Beginners)

Deep Learning for Perception — BSCS, FAST-NUCES

From Compression to Generation

The Big Picture - Your Learning Journey

Think about what you've learned so far:
Weeks 1-10: You learned how AI RECOGNIZES things

- CNN: "Is this a cat or dog?" (classification)
- Object detection: "Where are the objects?" (localization)
- RNN/LSTM: "What happens next?" (sequence prediction)

This Week: Learn how AI UNDERSTANDS and CREATES things!

- "What makes a '7' truly look like a seven?"
- "Can we compress images in a smart way?"
- "**Can AI generate completely NEW, realistic images?**"

Your Journey This Week:

1. **The Problem:** Why too many pixels is a nightmare (Curse of Dimensionality)
2. **Solution 1:** Autoencoders - Intelligent compression
3. **Limitation:** They can only copy, not create new things
4. **Solution 2:** VAE - Can actually GENERATE new data!
5. **The Magic:** How to make neural networks creative

Promise: By the end, you'll understand how AI:

- Generates realistic human faces that don't exist
- Creates new art and images
- Smoothly transforms between different images
- Compresses data better than traditional methods

Don't worry! We'll build everything step-by-step with lots of examples and analogies.

1 The Curse of Dimensionality

Why Does This Matter?

The Fundamental Question: "Why can't we just use all the pixels?"
You might think: "An MNIST image is just 784 numbers (28×28 pixels). My computer handles millions of numbers easily. Why is this a problem?"

The shocking answer: It's not about computing power. It's about the GEOMETRY of high-dimensional space. In 784 dimensions, space behaves in completely counterintuitive ways that make learning impossible!
Real impact: Without understanding this, you can't understand WHY autoencoders are needed, or HOW they work.

1.1 Let's Build Intuition Step by Step

Building Block

Start Simple: What is a "Dimension"?
Let's build from what you know:
1 Dimension (a line):

- Example: Temperature on a thermometer
- You need just ONE number to describe anything: 25°C
- Think: A point on a ruler

2 Dimensions (a flat surface):

- Example: Your location on a map
- You need TWO numbers: (latitude, longitude)
- Think: A point on a sheet of paper

3 Dimensions (our physical space):

- Example: Location of a drone
- You need THREE numbers: (latitude, longitude, height)
- Think: A point floating in a room

784 Dimensions (an MNIST image!):

- Each pixel is ONE dimension (brightness 0-255)
- 28 × 28 pixels = 784 dimensions
- You need 784 numbers to describe one image!

Mind-bending fact: We can't visualize 784D space, but mathematics still works there!

1.2 The First Problem: Space Explodes

Analogy (Make It Real!)

Analogy: Filling Boxes with Marbles
Imagine you want to fill a space with marbles, placing them 0.1 units apart:
1D (a line):

- To cover a 1-meter line: need 10 marbles
- Easy! You can hold 10 marbles in your hand

2D (a square):

- To cover a 1m × 1m square: need $10 \times 10 = 100$ marbles

- Still manageable. That's a small box of marbles

3D (a cube):

- To cover a 1m × 1m × 1m cube: need $10 \times 10 \times 10 = 1{,}000$ marbles
- Getting heavy! But still doable

10D (hard to imagine):

- Need $10^{10} = 10{,}000{,}000{,}000$ marbles
- That's 10 BILLION marbles!
- Would fill an entire warehouse

784D (MNIST image space):

- Need 10^{784} marbles
- This number is: 1 followed by 784 zeros!
- **More than all atoms in the universe!**
- **PHYSICALLY IMPOSSIBLE!**

The lesson: Space grows EXPONENTIALLY with dimensions. You run out of data FAST!

Concrete Example

Concrete Example: MNIST Dataset
What we have:

- 60,000 training images
- Each image: 784 dimensions (28×28 pixels)

What we need to fill the space:

- To properly "cover" 784-dimensional space with samples
- We'd need 10^{784} images (roughly)
- That's more images than atoms in the universe!

Reality:

- We have: 60,000 images
- We need: 10^{784} images
- Our data is like a few drops of water in an ocean
- Most of the space is EMPTY - no data at all!

Consequence:

- When predicting, model encounters regions it's NEVER seen
- It must guess blindly
- Learning becomes unreliable

This is called "data sparsity" - one of the curses!

1.3 The Second Problem: Everything Becomes Far Apart

Analogy (Make It Real!)

Analogy: Finding Your Friend
In an empty room (2D):

- You can easily tell: "That person is 2 meters away, that one is 10 meters away"
- Distance is meaningful
- You can find your close friends easily

In a city (sort of 3D):

- Still works: "This friend lives 1 km away, that one lives 10 km away"
- "Close" vs "far" makes sense

In a huge stadium (3D with many people):

- Everyone starts to seem equally far
- But you can still tell differences

In 784-dimensional space:

- ALL points become roughly the SAME distance apart!
- There's no "close friend" vs "distant acquaintance"
- Everyone is equally far!
- The concept of "nearest neighbor" breaks down

Why this matters:

- Many ML algorithms rely on "similarity" (distance)
- K-Nearest Neighbors: needs meaningful distances
- Clustering: needs meaningful distances
- In high dimensions: distances become meaningless!

Concrete Example

Concrete Example: Finding Similar Images
Low dimensions (works well):

- If images represented in 3D
- Distance between image A and B: 5 units
- Distance between image A and C: 20 units
- Clear conclusion: "B is more similar to A than C is"

High dimensions (breaks down):

- In 784D space
- Distance between image A and B: 148.2 units
- Distance between image A and C: 148.7 units
- Distance between image A and random noise: 148.5 units
- **Everything seems equally similar/dissimilar!**
- Algorithm can't tell which image is actually more similar

This is called "distance concentration" - another curse!

1.4 The Third Problem: Volume Hides in Corners

Analogy (Make It Real!)

Analogy: The Orange Mystery
Cut an orange in half (2D slice):

- Most of the juice (volume/mass) is in the CENTER
- The peel is thin, the middle is thick
- This matches our intuition

Now imagine a 100-dimensional orange:

- **ALL the juice is in the PEEL!**
- The center is basically EMPTY
- 99.9999% of volume is in the outer 1% of radius
- Your intuition from 2D completely fails!

Consequence:

- If you randomly pick a point in high-D space
- It will almost ALWAYS be on the boundary (in the "peel")
- Almost NEVER in the interior (the "center")
- Data naturally clusters at the edges, not the middle

Why this is weird:

- In 2D/3D: uniform distribution fills space uniformly
- In high-D: uniform distribution concentrates on boundary
- This makes sampling and learning very tricky

Stop and Think!

Pause and Reflect:
Before moving on, make sure you understand these three curses:
1. Space explodes:

- Number of "grid points" grows as 10^d (d = dimensions)
- We don't have enough data to fill high-D space

2. Everything becomes far apart:

- All distances become approximately equal
- "Nearest neighbor" becomes meaningless

3. Volume concentrates on boundary:

- In high dimensions, most volume is in thin outer shell
- Center is essentially empty

If you get these three, you understand why working with raw pixels is so hard!

1.5 The Hope: The Manifold Hypothesis

Why Does This Matter?

Wait, if the curse is so bad, how does deep learning work at all?
Great question! This is where the **Manifold Hypothesis** saves us.

Building Block

The Key Insight: Real Data is Special
The curse says: 784-dimensional space is impossibly huge.
The hypothesis says: Real images don't actually USE all 784 dimensions independently!
What does this mean?
Think about face photos:

- A face has thousands of pixels
- But faces don't vary in all possible ways
- They vary in just a few ways:

Ways faces actually vary:

1. **Whose face?** (identity - but limited number of people)
2. **Which way looking?** (angle left/right, up/down - maybe 2 dimensions)
3. **How lit?** (lighting from different angles - maybe 2 dimensions)
4. **What expression?** (happy/sad/neutral - maybe 1-2 dimensions)
5. **How close?** (zoom level - 1 dimension)

Total: Only about 5-10 "true" ways faces vary, NOT 784!
Technical term: These 10 variables control all 784 pixels together. The data lives on a low-dimensional "manifold" (surface) embedded in the high-dimensional space.

Analogy (Make It Real!)

Analogy: Earth's Surface
Embedding:

- Earth exists in 3D space
- But Earth's SURFACE is only 2D!
- You need only 2 numbers: latitude and longitude
- You can't go "inside" Earth or fly into space (for our purposes)

Similarly for images:

- Images exist in 784D space
- But actual images live on a 10D "surface" (manifold)
- You can't make arbitrary changes to pixels
- Most random 784D points don't look like real images!

The manifold:

- The set of all "realistic-looking" images
- Is a low-dimensional surface

- Embedded in high-dimensional pixel space

Concrete Example

Concrete Example: MNIST Digits
Ambient space: 784 dimensions (28×28 pixels)
Actual variation:

- Which digit? (0-9, but that's discrete)
- How thick? (pen stroke width - 1 dimension)
- How tilted? (rotation angle - 1 dimension)
- Where positioned? (x,y offset - 2 dimensions)
- How stretched? (aspect ratio - 1 dimension)
- Writing style? (different people's handwriting - maybe 2-3 dimensions)

Total intrinsic dimensions: Maybe 5-10, NOT 784!
This means:

- We don't need to learn all 10^{784} possible images
- We only need to learn the 10-dimensional manifold
- This is MUCH easier!
- Autoencoders are designed to find this manifold

Key Takeaway

Summary: Why the Curse Exists and Why We Can Still Succeed
The Curse:

- High dimensions make space impossibly large
- Data becomes sparse, distances meaningless, volume weird
- Working with raw 784D pixels seems impossible

The Hope:

- Real data doesn't use all dimensions independently
- Data lives on a low-D manifold (10D) in high-D space (784D)
- We only need to learn the manifold, not the whole space

The Solution (coming next):

- **Autoencoders** automatically find this low-D manifold
- They compress 784D $\rightarrow$ 20D $\rightarrow$ back to 784D
- By forcing compression, they find what's truly important!

Self-Test (Check Your Understanding!)

Check Your Understanding:
Q1: Why can't we just use all 784 pixels as features directly?
Answer: Data becomes too sparse - we'd need 10^{784} samples to fill the space, but we only have 60,000. Most of the space is empty!
Q2: What does it mean that "all distances become equal" in high dimensions?
Answer: In high-D space, nearest and farthest neighbors are roughly the same distance

apart, making "similarity" meaningless.
Q3: If images are 784-dimensional, how can they "really" only be 10-dimensional?
Answer: The 784 pixels aren't independent - they're controlled by 10 underlying factors (angle, lighting, etc.). Images live on a 10D "manifold" in 784D space.
Q4: What will an autoencoder do about this?
Answer: Find the low-D manifold by compressing to 20D and back, forcing the network to learn what's truly important!
If you got 3-4 correct, you're ready to move on!

2 Autoencoders: Intelligent Compression

Why Does This Matter?

The Question: "Now that we know high dimensions are a problem, what can we do?"
The Idea: What if we could:

- Take a 784-dimensional image
- Compress it to just 20 dimensions (97% reduction!)
- But in a SMART way that keeps what's important
- Then reconstruct the original from those 20 numbers

That's exactly what an autoencoder does!
Why it matters:

- Finds the low-D manifold automatically
- Learn features without labels (unsupervised!)
- Can compress data, denoise images, detect anomalies
- Foundation for generative models (VAE coming later!)

2.1 The Big Idea (Before the Details)

Analogy (Make It Real!)

Analogy: Taking a Photograph
Think about photography:
Step 1 - Encoding (Camera):

- You see a 3D scene (real world)
- Camera projects it onto 2D photograph
- Lost information: depth (how far things are)
- Kept information: colors, shapes, positions

Step 2 - The Compressed Representation:

- The 2D photo is a compressed version of 3D reality
- Much smaller (went from 3D to 2D)
- But still contains essential visual information

Step 3 - Decoding (Looking at photo):

- Your brain sees the 2D photo
- Reconstructs mental model of 3D scene
- Not perfect (depth is guessed)
- But recognizable!

Autoencoder does the same:

- Encoder: 784D image $\rightarrow$ 20D code (like camera: 3D $\rightarrow$ 2D)
- Code: Compressed representation (like 2D photo)
- Decoder: 20D code $\rightarrow$ 784D reconstruction (like brain imagining 3D)

Key insight: Forcing compression makes the network learn what's truly important!

2.2 Building the Architecture Step by Step

Building Block

Let's Build an Autoencoder Piece by Piece
What we want to do:

- Input: Image with 784 pixels
- Output: Reconstructed image with 784 pixels
- Goal: Output should match input as closely as possible
- Constraint: Must pass through narrow bottleneck (20 dimensions)

The trick: The bottleneck forces compression!
Three main parts:
Part 1 - Encoder (The Compressor):

- Takes 784-dimensional input
- Gradually reduces dimensions: $784 \to 512 \to 256 \to 128 \to 20$
- Like a funnel getting narrower
- Must decide: "What information is essential?"
- Output: 20-dimensional code z

Part 2 - Bottleneck (The Code):

- Just 20 numbers!
- This is the compressed representation
- Contains essential information about the image
- Like a "summary" of the image

Part 3 - Decoder (The Reconstructor):

- Takes 20-dimensional code
- Gradually expands dimensions: $20 \to 128 \to 256 \to 512 \to 784$
- Like an inverse funnel getting wider
- Must recreate image from summary
- Output: 784-dimensional reconstruction

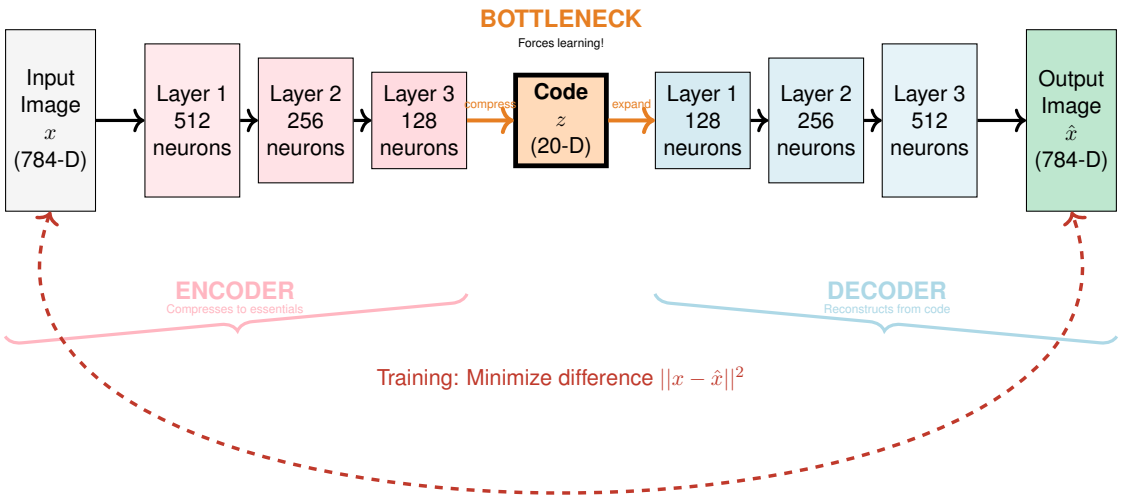


Figure 1: Autoencoder architecture: Input → Compress → Bottleneck → Expand → Output

2.3 How Does Training Work?

Building Block

The Training Process (Step by Step)
Step 1 - Forward Pass:

- Feed image x into encoder
- Encoder outputs code z (20 numbers)
- Feed code z into decoder
- Decoder outputs reconstruction $\hat{x}$ (784 numbers)

Step 2 - Compute Loss:

- Compare original x with reconstruction $\hat{x}$
- Loss = how different they are
- Common choice: Mean Squared Error (MSE)
- Loss $= \frac{1}{784}\sum_{i=1}^{784}(x_i - \hat{x}_i)^2$
- Lower loss = better reconstruction

Step 3 - Backpropagation:

- Compute gradients of loss w.r.t. all weights
- Update encoder weights to create better codes
- Update decoder weights to reconstruct better

Step 4 - Repeat:

- Process entire dataset
- Multiple epochs (passes through data)

- Network gradually learns to compress intelligently

What the network learns:

- Encoder: "How to extract essential features"
- Decoder: "How to reconstruct from features"
- Together: "What information is truly important"

Concrete Example

Concrete Example: MNIST Digit Compression
Setup:

- Input: Image of digit "7" (784 pixels)
- Autoencoder: Compresses to 20 numbers
- Output: Reconstructed "7"

What the 20-number code might represent:

1. Which digit? (but this is learned, not given!)
2. Stroke thickness: 0.7 (normalized)
3. Tilt angle: 0.2 (slightly tilted right)
4. Vertical position: 0.5 (centered)
5. Horizontal position: 0.5 (centered)
6. Width/height ratio: 0.8
7. Crossbar height: 0.6 (where horizontal line crosses)
8. Curve at top: 0.3 (how curved)
9. ... (12 more learned features)

Process:

- Encoder sees all 784 pixels
- Extracts these 20 features automatically
- Decoder uses these 20 features
- Reconstructs all 784 pixels
- Result: Looks like original "7" (maybe slightly blurry)

The magic:

- Nobody told the network what features to extract!
- It learned by itself through training
- Just by trying to minimize reconstruction error
- Bottleneck forced it to learn compressed representation

2.4 Why the Bottleneck is Crucial

Stop and Think!

Thought Experiment: What if there was NO bottleneck?
Scenario: Network with 784 $\rightarrow$ 784 $\rightarrow$ 784 dimensions

- No compression needed
- Network could just copy pixels directly: $\hat{x} = x$
- Achieves perfect reconstruction (loss = 0)
- But learns NOTHING useful!
- It's just an expensive copy machine

With bottleneck: Network with 784 $\rightarrow$ 20 $\rightarrow$ 784

- MUST compress (can't fit 784 numbers in 20 slots!)
- Must decide what's important
- Must learn meaningful features
- Can't achieve perfect reconstruction
- But learns USEFUL representation!

The bottleneck is like:

- Forcing you to explain a movie in 3 sentences
- You must focus on the essential plot
- You learn what's truly important
- Same for autoencoder!

Common Mistake - Don't Fall For This!

Common Mistakes Students Make:
Mistake 1: "The encoder is just compressing like a ZIP file"

- Wrong! ZIP uses generic compression (works for any file)
- Autoencoder learns SEMANTIC compression (understands images)
- ZIP can't "understand" that nearby pixels are similar
- Autoencoder learns that "this looks like a digit" matters

Mistake 2: "We could just use PCA instead"

- PCA only finds LINEAR relationships
- Autoencoders can learn NON-LINEAR patterns
- Image manifolds are highly non-linear!
- Example: Rotating an image is non-linear in pixel space

Mistake 3: "Bigger bottleneck is always better"

- Too small: Can't capture enough information (underfit)
- Too large: Doesn't force meaningful compression (overfit)
- Sweet spot: 20-100 for MNIST (experiment to find!)

Mistake 4: "The code will look meaningful to humans"

- The 20 numbers don't have obvious meaning!

- Not like "number 3 is tilt angle"
- They're learned, abstract representations
- But the decoder knows how to use them!

2.5 What Can We Do With Autoencoders?

Building Block

Practical Applications

1. Dimensionality Reduction (Like PCA but better):

- Compress 784D to 2D for visualization
- Plot all MNIST digits in 2D plane
- See clusters: all "7"s group together
- Non-linear, so better than PCA

2. Feature Learning (Unsupervised):

- Train autoencoder on unlabeled data
- Use learned encoder as feature extractor
- Feed features to supervised classifier
- Often works better than training from scratch!

3. Denoising:

- Train with: Input = noisy image, Target = clean image
- Network learns to remove noise
- Code doesn't have space for noise details
- Reconstruction is clean!

4. Anomaly Detection:

- Train on normal data only
- Normal data: low reconstruction error
- Anomaly: high reconstruction error (never seen before!)
- Application: Find defects in manufacturing

5. Data Compression:

- Store only 20 numbers instead of 784
- Save 97% space!
- Lossy compression (some quality lost)
- But learned for specific type of data

2.6 The Big Limitation (Leads to VAE)

Why Does This Matter?

What Autoencoders CANNOT Do:
They cannot GENERATE new data!
Why not?

- You can encode an existing image: $x \to z$
- You can decode it back: $z \to \hat{x}$
- But you CANNOT sample new z and decode!

The problem:

- What values can z take?
- No idea! There's no distribution defined
- The 20 numbers could be anything
- Random z probably decodes to garbage

Example:

- Train autoencoder on MNIST
- Get codes: $z_1, z_2, \ldots, z_{60000}$ for all training images
- Want to generate NEW digit
- Sample $z_{new} = ?$ How? From where?
- No distribution! Just discrete points scattered around
- Random vector likely produces nonsense

This is a FUNDAMENTAL limitation!
The solution? Variational Autoencoders (VAE) - coming next!

Key Takeaway

Summary: Autoencoders
What they are:

- Neural networks: Input $\to$ Compress $\to$ Bottleneck $\to$ Expand $\to$ Output
- Goal: Output = Input (reconstruction)
- Bottleneck forces learning of compressed representation

What they learn:

- Encoder: Extract essential features (784D $\to$ 20D)
- Decoder: Reconstruct from features (20D $\to$ 784D)
- Together: The low-dimensional manifold of data

Applications:

- Dimensionality reduction, feature learning, denoising
- Anomaly detection, data compression

Big limitation:

- NOT generative - cannot create new data
- No distribution over codes

- Random codes decode to garbage

Next: VAE fixes this limitation!

Self-Test (Check Your Understanding!)

Check Your Understanding:
Q1: Why does the bottleneck force the network to learn useful features?
Answer: With only 20 slots, it can't memorize all 784 pixels. Must extract essential information to enable reconstruction.
Q2: What's the difference between autoencoder and ZIP compression?
Answer: ZIP is generic (works for any file), autoencoder learns semantic meaning (understands images specifically).
Q3: Can we use an autoencoder to generate new MNIST digits?
Answer: NO! No distribution over z. Random z produces garbage, not realistic digits.
Q4: Why can't we just use PCA instead of autoencoders?
Answer: PCA is linear only. Image manifolds are highly non-linear. Autoencoders learn non-linear mappings.
If you got 3-4 correct, you understand autoencoders! Ready for VAE?

3 Variational Autoencoders (VAE): The Generative Solution

Why Does This Matter?

The Big Question: "Can we fix the autoencoder so it CAN generate new data?"
Autoencoder problem recap:

- Can compress: $x \to z$ (encoding works)
- Can reconstruct: $z \to \hat{x}$ (decoding works)
- CANNOT generate: No distribution to sample z from!

What we want:

- Sample random code: $z_{new} \sim ?$ (from some distribution)
- Decode it: $x_{new} = \text{Decoder}(z_{new})$
- Get realistic new image!

VAE makes this possible!
The key innovations:

1. Make the encoder probabilistic (output distribution, not single point)
2. Force all codes to follow standard Gaussian: $z \sim \mathcal{N}(0, I)$
3. Now we CAN sample: just sample from Gaussian!

Result: A true generative model that can create new data!

3.1 The Core Idea (Intuition First)

Analogy (Make It Real!)

Analogy: Weather Forecast
Deterministic Forecast (like Autoencoder):

- "Tomorrow will be 25°C"
- Single number (point estimate)
- No uncertainty
- Can only predict for days you've seen

Probabilistic Forecast (like VAE):

- "Tomorrow will be 25°C $\pm$ 3°C"
- Mean: 25°C, Standard deviation: 3°C
- Represents uncertainty!
- Can sample: 27°C, 23°C, 24.5°C (all plausible)

VAE does the same:
Autoencoder:

- Encode image $\to$ Get single code: $z = [0.5, -0.2, 1.1, \ldots]$ (20 numbers)
- Deterministic: Same image $\to$ Same code always

VAE:

- Encode image → Get distribution: $\mu = [0.5, -0.2, \ldots]$, $\sigma = [0.1, 0.3, \ldots]$
- Sample code: $z \sim \mathcal{N}(\mu, \sigma^2)$
- Stochastic: Same image → Different codes (but all similar!)
- Can sample new codes from this distribution!

Building Block

VAE Step-by-Step (Building the Intuition)
Step 1: What changes in the Encoder
Old (Autoencoder):

- Input image → Encoder → Code z (20 numbers)
- Example: Image of "7" → $z = [0.5, -0.2, 1.1, \ldots]$

New (VAE):

- Input image → Encoder → μ (20 numbers) AND σ (20 numbers)
- Think: Encoder outputs MEAN and STD for each dimension
- Example: Image of "7" → $\mu = [0.5, -0.2, \ldots]$, $\sigma = [0.1, 0.3, \ldots]$
- Then SAMPLE: $z \sim \mathcal{N}(\mu, \sigma^2)$

Step 2: Why this helps
Multiple similar images now have overlapping codes:

- Image of "7" (thin): $\mu_1 = [0.5, -0.2, \ldots]$, $\sigma_1 = [0.1, 0.1, \ldots]$
- Image of "7" (thick): $\mu_2 = [0.6, -0.1, \ldots]$, $\sigma_2 = [0.1, 0.1, \ldots]$
- The distributions OVERLAP!
- Samples from both can be similar
- Decoder learns smooth function: similar codes → similar images

Step 3: Forcing Gaussian Structure
We add a constraint:

- All codes must be close to standard Gaussian: $\mathcal{N}(0, I)$
- This means: μ shouldn't be too far from 0, σ shouldn't be too far from 1
- Enforced by adding KL divergence to loss

Result:

- All codes cluster around origin with variance ≈ 1
- Creates smooth, continuous latent space
- NOW we can sample: $z_{new} \sim \mathcal{N}(0, I)$ and decode!

Concrete Example

Concrete Example: VAE on MNIST
Training:
Image 1: Thin "7"

- Encoder outputs: $\mu_1 = [0.3, 0.1, -0.5, \ldots]$, $\sigma_1 = [0.2, 0.1, 0.3, \ldots]$
- Sample: $z_1 = \mu_1 + \sigma_1 \odot \epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$
- Might get: $z_1 = [0.35, 0.08, -0.48, \ldots]$

- Decoder reconstructs thin "7"

Image 2: Thick "7"

- Encoder outputs: $\mu_2 = [0.5, 0.2, -0.4, \ldots]$, $\sigma_2 = [0.2, 0.1, 0.3, \ldots]$
- Distributions overlap with Image 1!
- Sample: $z_2 = [0.48, 0.25, -0.35, \ldots]$
- Decoder reconstructs thick "7"

Key observation:

- z_1 and z_2 are close to each other
- They might even overlap when sampled!
- Decoder learns: codes in this region $\rightarrow$ "7"-like images

Generation (after training):

- Sample: $z_{new} = [0.4, 0.15, -0.45, \ldots] \sim \mathcal{N}(0, I)$
- This is between z_1 and z_2!
- Decoder outputs: Medium thickness "7"
- It's a NEW digit, never seen before!
- But looks realistic because it's in the smooth learned manifold

3.2 The VAE Architecture

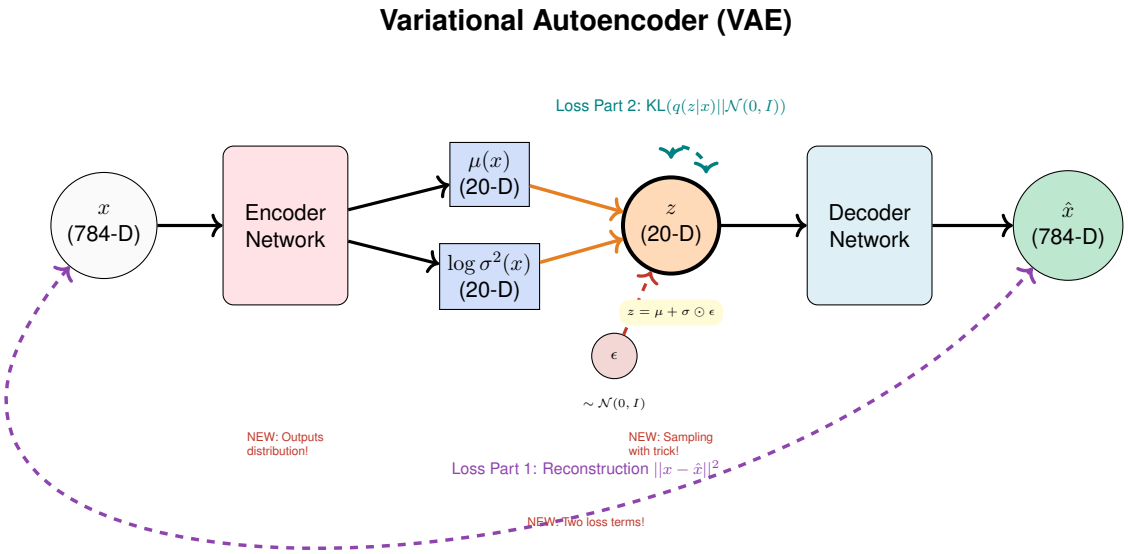


Figure 2: VAE architecture: Probabilistic encoder, reparameterization trick, probabilistic decoder

3.3 The Reparameterization Trick (The Clever Part)

Why Does This Matter?

The Problem We Need to Solve:
We want to:

1. Encoder outputs μ and σ
2. Sample $z \sim \mathcal{N}(\mu, \sigma^2)$
3. Decode z to get $\hat{x}$
4. Backpropagate to update μ and σ

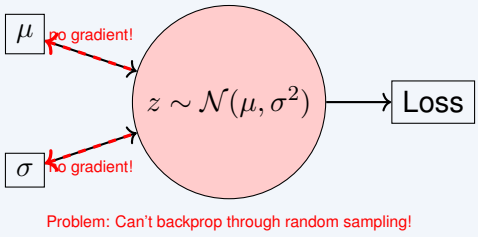
But there's a PROBLEM with step 2:

- Sampling is a random operation
- Random operations break backpropagation!
- Can't compute $\frac{\partial z}{\partial \mu}$ through sampling
- Gradients don't flow!

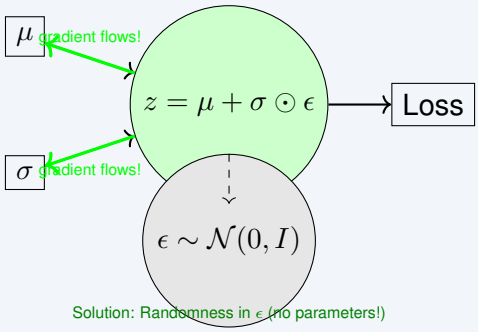
The reparameterization trick solves this!

Building Block

Understanding the Problem (Visual Explanation)
Naive approach (doesn't work):



Reparameterization trick (works!):



Building Block

The Trick Explained Step-by-Step
What we want: Sample z from $\mathcal{N}(\mu, \sigma^2)$
Original (non-differentiable):

$$z \sim \mathcal{N}(\mu, \sigma^2)$$

Problem: z is random, can't compute $\frac{\partial z}{\partial \mu}$!
Reparameterized (differentiable):

$$\boxed{z = \mu + \sigma \odot \epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0, I)}$$

Why this works:
1. Mathematically equivalent:

- If $\epsilon \sim \mathcal{N}(0, 1)$, then $\mu + \sigma\epsilon \sim \mathcal{N}(\mu, \sigma^2)$
- This is a standard property of Gaussians!
- Same distribution, different representation

2. Gradients flow:

- $\frac{\partial z}{\partial \mu} = 1$ (clear!)
- $\frac{\partial z}{\partial \sigma} = \epsilon$ (clear!)
- Both are well-defined
- Backpropagation works!

3. Randomness separated:

- Random part: ϵ (has no parameters, no gradients needed)
- Deterministic part: $\mu + \sigma \odot \epsilon$ (gradients flow)
- Best of both worlds!

Concrete Example

Concrete Numerical Example
Encoder outputs:

- $\mu = [0.5, -0.2, 1.0]$ (3-D for simplicity)
- $\sigma = [0.1, 0.3, 0.2]$

Sampling:

- Generate random: $\epsilon = [0.5, -1.2, 0.3] \sim \mathcal{N}(0, I)$
- Compute: $z = \mu + \sigma \odot \epsilon$
- $z_1 = 0.5 + 0.1 \times 0.5 = 0.55$
- $z_2 = -0.2 + 0.3 \times (-1.2) = -0.56$
- $z_3 = 1.0 + 0.2 \times 0.3 = 1.06$
- Result: $z = [0.55, -0.56, 1.06]$

Backpropagation:

- Suppose $\frac{\partial \text{Loss}}{\partial z} = [2.0, -1.0, 0.5]$ (from decoder)
- Then: $\frac{\partial \text{Loss}}{\partial \mu} = [2.0, -1.0, 0.5]$ (gradients flow directly!)
- And: $\frac{\partial \text{Loss}}{\partial \sigma} = [2.0 \times 0.5, -1.0 \times (-1.2), 0.5 \times 0.3]$
- $= [1.0, 1.2, 0.15]$
- Both μ and σ get updated!

The magic: Even though sampling is random, gradients still flow because randomness is in ϵ (no parameters)!

Common Mistake - Don't Fall For This!

Common Confusion About the Trick
Misconception: "We're removing randomness"

- Wrong! z is still random
- ϵ is random, so $z = \mu + \sigma \odot \epsilon$ is random
- Different ϵ each time $\rightarrow$ different z each time

Truth: "We're SEPARATING randomness"

- Random part (ϵ): Has no parameters
- Deterministic part ($\mu + \sigma\odot$): Has parameters
- Gradients flow through deterministic part
- Randomness is still there!

Think of it like:

- Rolling dice (random) vs computing sum (deterministic)
- We can't differentiate dice rolling
- But we CAN differentiate "2 + 3 × (dice result)"
- Same idea!

3.4 The VAE Loss Function

Why Does This Matter?

VAE needs TWO things to work:
1. Good reconstruction:

- Decoded image should match original
- Like regular autoencoder

2. Structured latent space:

- All codes should be Gaussian-distributed
- Centered around 0, variance around 1
- This allows sampling!

So we need TWO loss terms!

Building Block

VAE Loss = Reconstruction + Regularization
Total Loss:

$$\mathcal{L}_{VAE} = \underbrace{\mathcal{L}_{reconstruction}}_{\text{Make it accurate}} + \underbrace{\beta \cdot \mathcal{L}_{KL}}_{\text{Make it Gaussian}}$$

Term 1: Reconstruction Loss

$$\mathcal{L}_{reconstruction} = ||x - \hat{x}||^2 = \frac{1}{784}\sum_{i=1}^{784}(x_i - \hat{x}_i)^2$$

What it does:

- Measures: How different is reconstruction from original?
- Forces: Decoder to accurately reconstruct images
- Alone would lead to: Good reconstructions but chaotic latent space

Term 2: KL Divergence

$$\mathcal{L}_{KL} = D_{KL}(q_\phi(z|x)||\mathcal{N}(0, I))$$

For Gaussian encoder, this has a closed form:

$$\mathcal{L}_{KL} = \frac{1}{2}\sum_{j=1}^{20}(\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1)$$

What it does:

- Measures: How different is $q(z|x)$ from standard Gaussian?
- Forces: Encoder outputs to be close to $\mathcal{N}(0, I)$
- Means: μ should be near 0, σ should be near 1
- Alone would lead to: Ignoring input (everything maps to same Gaussian)

Analogy (Make It Real!)

Analogy: Packing Suitcases for a Trip
Reconstruction loss (Term 1):

- Like: "Pack everything you need"
- Wants: Bring all clothes, toiletries, books, etc.
- Alone would lead to: Huge, overstuffed suitcase

KL divergence (Term 2):

- Like: "Suitcase must be standard size and weight"
- Wants: Compact, organized, airline-approved
- Alone would lead to: Empty suitcase (too restrictive!)

Together:

- Balance: Pack essentials efficiently
- Result: Standard-size suitcase with important items
- Same for VAE: Structured latent space that still reconstructs well!

Building Block

Understanding the Balance
What each term wants:
Reconstruction term alone:

- Wants: Encode each image to unique, specific code
- Example: Image 1 $\rightarrow \mu = [10, 20, 30, \ldots]$, Image 2 $\rightarrow \mu = [-15, 8, -22, \ldots]$
- Problem: Codes spread everywhere, $\sigma \rightarrow 0$ (deterministic)
- Can't sample! (no structure)

KL term alone:

- Wants: All images $\rightarrow \mu = [0, 0, \ldots]$, $\sigma = [1, 1, \ldots]$
- Example: Every image maps to standard Gaussian
- Problem: Ignores input! (can't reconstruct)
- Can sample, but outputs are garbage

Together (the balance):

- Reconstruction: "Encode similar images to similar codes"
- KL: "Keep all codes near origin with reasonable variance"
- Result: Structured latent space + good reconstruction
- Similar images $\rightarrow$ overlapping Gaussians near origin
- Different images $\rightarrow$ separate Gaussians, but all near origin
- Can sample anywhere near origin $\rightarrow$ plausible images!

Concrete Example

Concrete Example: Training on Three Images
Image A: Thin "7"

- Reconstruction wants: Unique code, say $\mu = [5, 3, \ldots]$
- KL wants: μ close to $[0, 0, \ldots]$
- Compromise: $\mu = [0.3, 0.2, \ldots]$, $\sigma = [0.2, 0.15, \ldots]$

Image B: Thick "7"

- Should be similar to Image A
- Reconstruction wants: Different but nearby code
- KL wants: Also near origin
- Result: $\mu = [0.5, 0.25, \ldots]$, $\sigma = [0.2, 0.15, \ldots]$
- Overlaps with Image A! (good!)

Image C: "3"

- Very different from "7"
- Reconstruction wants: Distinct code
- KL wants: Still near origin
- Result: $\mu = [-0.4, 0.6, \ldots]$, $\sigma = [0.2, 0.15, \ldots]$
- Different region from "7"s, but still near origin

After training:

- "7" images cluster around $[0.4, 0.2, \ldots]$
- "3" images cluster around $[-0.4, 0.6, \ldots]$
- All clusters near origin (enforced by KL)
- Can sample $z \sim \mathcal{N}(0, I)$:
  - Near $[0.4, 0.2, \ldots] \rightarrow$ generates "7"
  - Near $[-0.4, 0.6, \ldots] \rightarrow$ generates "3"
  - Anywhere near origin $\rightarrow$ generates something digit-like!

Stop and Think!

Pause and Reflect:
Before moving on, make sure you understand:
1. Why we need probabilistic encoding:

- Single codes (AE) don't allow sampling
- Distributions (VAE) create smooth latent space

2. The reparameterization trick:

- $z = \mu + \sigma \odot \epsilon$ (where $\epsilon \sim \mathcal{N}(0, I)$)
- Separates randomness (ϵ) from parameters (μ, σ)
- Enables backpropagation through sampling

3. The two loss terms:

- Reconstruction: Make it accurate
- KL divergence: Make it Gaussian
- Together: Structured space + good reconstruction

If you get these three points, you understand VAE!

3.5 Normal AE vs VAE: The Critical Differences

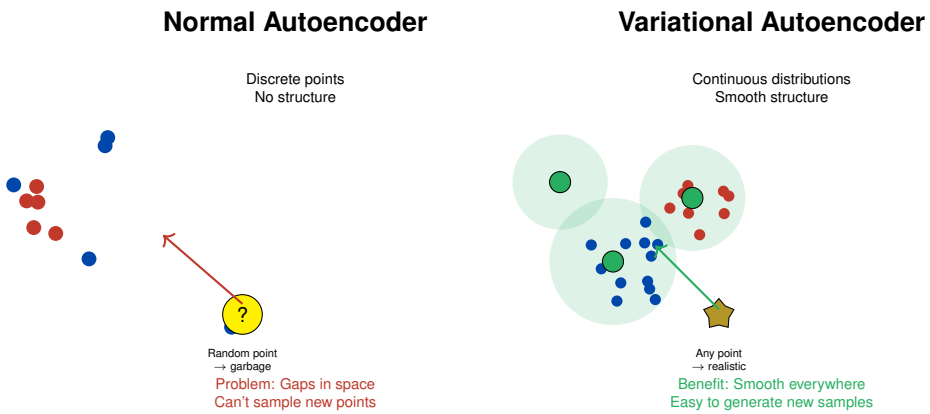


Figure 3: Latent space: Normal AE (discrete, gaps) vs VAE (continuous, smooth)

Building Block

Side-by-Side Comparison

Aspect	Normal AE	VAE
Encoder Output	Single code z (20 numbers)	Distribution: μ and σ (20 numbers each)
Latent Code	Deterministic: same input $\rightarrow$ same z	Stochastic: same input $\rightarrow$ different z (sampled)
Latent Space	Discrete points, gaps	Continuous Gaussians, smooth
Can Generate?	NO - don't know where to sample	YES - sample from $\mathcal{N}(0, I)$
Loss Function	Reconstruction only	Reconstruction + KL divergence
Interpolation	May produce garbage in gaps	Always produces realistic images
Use Case	Compression, features	Generation, modeling

The key difference: VAE models a DISTRIBUTION, not just mappings!

Key Takeaway

Summary: Variational Autoencoders
The Core Innovations:

1. **Probabilistic encoder:** Outputs μ and σ (distribution parameters)
2. **Reparameterization trick:** $z = \mu + \sigma \odot \epsilon$ (enables gradients)
3. **Structured prior:** Force $z \sim \mathcal{N}(0, I)$ (enables sampling)
4. **Two-part loss:** Reconstruction + KL divergence (balance accuracy and structure)

Why VAE works:

- KL term forces all codes near origin with variance ≈ 1
- Reconstruction term forces similar images to have overlapping distributions
- Result: Smooth, continuous latent space
- Can sample $z \sim \mathcal{N}(0, I)$ and decode to get NEW realistic images!

Applications:

- Generate new faces, digits, images
- Smooth interpolation between images
- Semi-supervised learning
- Anomaly detection
- Data augmentation

VAE vs Normal AE:

- AE: Learns mappings (compression/reconstruction)
- VAE: Learns distributions (can generate new data)
- VAE is a true GENERATIVE model!

Self-Test (Check Your Understanding!)

Final Self-Test: Do You Really Get It?
Q1: Why can't a normal autoencoder generate new images?
Answer: No distribution over codes. Don't know where to sample from. Random z likely produces garbage (not on the learned manifold).
Q2: What does the VAE encoder output?
Answer: Two sets of numbers: μ (mean, 20 numbers) and σ (std deviation, 20 numbers) - parameters of a Gaussian distribution.
Q3: Explain the reparameterization trick in simple terms.
Answer: Instead of sampling $z \sim \mathcal{N}(\mu, \sigma^2)$ (not differentiable), we compute $z = \mu + \sigma \odot \epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$ (differentiable). Separates randomness from parameters.
Q4: What do the two loss terms do?
Answer: Reconstruction ($||x - \hat{x}||^2$): make it accurate. KL divergence: force codes to be Gaussian $\mathcal{N}(0, I)$. Together create structured latent space with good reconstruction.
Q5: How do you generate a NEW image with a VAE?
Answer: Sample $z_{new} \sim \mathcal{N}(0, I)$ (20 random numbers from standard Gaussian), feed to decoder, get new realistic image!
Q6: Why does KL divergence create a smooth latent space?
Answer: Forces all codes near origin with variance ≈ 1. Makes distributions overlap. Decoder learns smooth function. Any point near origin decodes to something realistic.
Score:

- 6/6: Excellent! You understand VAEs deeply.
- 4-5/6: Good! Review the parts you missed.
- 2-3/6: Re-read the VAE section carefully.
- 0-1/6: Start again from autoencoders section.

End of Week 11 Notes

Deep Learning for Perception — FAST-NUCES

You now understand the foundation of generative AI!

What You've Learned

Congratulations! You've mastered:
1. The Problem: Curse of dimensionality (why high-D is hard)
2. First Solution: Autoencoders (intelligent compression)
3. The Limitation: Can't generate new data
4. Better Solution: VAE (true generative model)
5. The Magic: Reparameterization trick (how it works)
These concepts are fundamental to:

- Modern generative AI (DALL-E, Stable Diffusion use these ideas)
- Understanding GANs (next topic?)
- Diffusion models (cutting-edge)
- Any generative modeling task

You're now ready for advanced generative models!