Week 2

Deep Learning Fundamentals Review

[ECEA0649/ECE40049] Deep Learning for Image Processing | Spring 2026

Kihyun Na (Research Professor)

BK21 AI Project Group & Institute for Information and Communication Technology,
Handong Global University

HGU HANDONG GLOBAL UNIVERSITY

Announcements

Before we start — a few updates on course structure.

Updated Curriculum

** Adjusted based on survey results*

Week	Topic
Wk 1	OT + Introduction
Wk 2	DL Fundamentals Review (today)
Wk 3	CNN
Wk 4	RNN + Attention Mechanism
Wk 5	Transformer + ViT
Wk 6	Detection + Segmentation
Wk 7	Self-Supervised Learning
Wk 8	Review Literacy + Role Explanation
Wk 9	Foundation Models (CLIP, SAM) + Paper #1
Wk 10	Diffusion Models + Paper #2
Wk 11	Conditional Generation + Paper #3
Wk 12	Vision-Language Models + Paper #4
Wk 13	VLM Applications + Paper #5
Wk 14	Video Understanding + Paper #6
Wk 15	Embodied AI & Robot Vision + Paper #7
Wk 16	Miniconference (Final Project)

Lecture | Lecture + Paper | Miniconference

Challenge & Final Project

Challenge Participation

Team Formation
Self-organized, multiple team participation OK
Clearly define your role in each team

Instructor Involvement (your choice)
Include: Active idea/code/paper contribution
→ co-author on summary paper
Exclude: General feedback during class only
→ full autonomy

Timeline
This week: List up candidate challenges
Wk 3: Finalize teams + registration

Final Project (30%)

Format: Technical Report or Blog Post

Options
Challenge report (most students)
Research paper (discuss with instructor later)

Deliverable
Written report with Author Contributions section (who did what)

Presentation
Wk 16 Miniconference (team-based)

** Grading proportions (30/30/30/10) may be adjusted.*

Paper Presentation: Updated Roles

8 students → 7 units (undergrad pair = 1 unit) | 7 roles × 7 weeks

Author

Present the paper
+ Rebuttal

Area Chair

Synthesize reviews
Accept/Reject

Reviewer ×2 (Pro, Con)

Submit written review

Archaeologist

Prior work context briefing

Future Researcher

Limitations + follow-up ideas

Reproducibility Engineer

Reproducibility assessment

NEW

Reproducibility Engineer — Assess whether the paper provides enough detail to reproduce results. Is the code public? Are hyperparameters specified? How would you approach re-implementation? (Inspired by the "Hacker" role from Raffel & Jacobson's seminar model)

Weekly Class Structure

Wk 2–7: Foundations

Time	Activity
90 min	Lecture
~10 min	Break
40–50 min	Challenge Feedback & Discussion

Total: ~2h 30min

Wk 9–15: Seminar

Time	Activity
60–70 min	Lecture (topic overview)
~10 min	Break
~80 min	Paper Presentation Session

Total: ~2h 40min

Supplementary video lectures will be provided via LMS when topics need additional depth.

Today's Agenda

01

Images & Data

How computers see images, datasets, train/val/test split

20 min

02

Linear Classifiers → Neural Networks

From Wx+b to MLP — why we need depth and non-linearity

20 min

03

Loss & Optimization

Cross-Entropy, MSE, SGD → Adam, LR scheduling

20 min

04

Regularization & Training Tricks

Overfitting diagnosis, Dropout, BatchNorm, Data Augmentation

20 min

05

PyTorch Walkthrough

From DataLoader to training loop (self-study materials provided)

10 min

Part 1

Images & Data

How computers see images, and the data we use to teach them.

What a Computer Sees

Image Classification: A core computer vision task

Input: image

This image by Nikita is licensed under CC-BY 2.0

Output: Assign image to one of a fixed set of categories

cat
bird
deer
dog
truck

What a Computer Sees

Problem: Semantic Gap

This image by Nikita is
licensed under CC-BY 2.0

```
[[105 112 108 111 104  99 106  99  96 103 112 119 104  97  93  87]
 [ 91  98 102 106 104  79  98 103  99 105 123 136 110 105  94  85]
 [ 76  85  90 105 128 105  87  96  95  99 115 112 106 103  99  85]
 [ 99  81  81  93 120 131 127 100  95  98 102  99  96  93 101  94]
 [106  91  61  64  69  91  88  85 101 107 109  98  75  84  96  95]
 [114 108  85  55  55  69  64  54  64  87 112 129  98  74  84  91]
 [133 137 147 103  65  81  80  65  52  54  74  84 102  93  85  82]
 [128 137 144 140 109  95  86  70  62  65  63  63  60  73  86 101]
 [125 133 148 137 119 121 117  94  65  79  80  65  54  64  72  98]
 [127 125 131 147 133 127 126 131 111  96  89  75  61  64  72  84]
 [115 114 109 123 150 148 131 118 113 109 100  92  74  65  72  78]
 [ 89  93  90  97 108 147 131 118 113 114 113 109 106  95  77  80]
 [ 63  77  86  81  77  79 102 123 117 115 117 125 125 130 115  87]
 [ 62  65  82  89  78  71  80 101 124 126 119 101 107 114 131 119]
 [ 63  65  75  88  89  71  62  81 120 138 135 105  81  98 110 118]
 [ 87  65  71  87 106  95  69  45  76 130 126 107  92  94 105 112]
 [118  97  82  86 117 123 116  66  41  51  95  93  89  95 102 107]
 [164 146 112  80  82 120 124 104  76  48  45  66  88 101 102 109]
 [157 170 157 120  93  86 114 132 112  97  69  55  70  82  99  94]
 [130 128 134 161 139 100 109 118 121 134 114  87  65  53  69  86]
 [128 112  96 117 150 144 120 115 104 107 102  93  87  81  72  79]
 [123 107  96  86  83 112 153 149 122 109 104  75  80 107 112  99]
 [122 121 102  80  82  86  94 117 145 148 153 102  58  78  92 107]
 [122 164 148 103  71  56  78  83  93 103 119 139 102  61  69  84]]
```

What the computer sees

An image is just a big grid of numbers between [0, 255]:

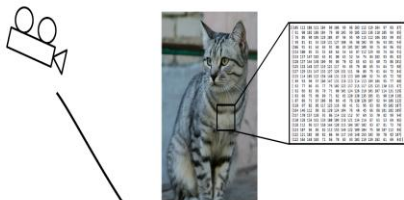
e.g. 800 x 600 x 3
(3 channels RGB)

An image is a 3D tensor of pixel values. Each pixel has R, G, B channels (0–255).

The Semantic Gap

Why is visual recognition hard for computers?

Viewpoint Variation



All pixels change when the camera moves!

Illumination

Lighting changes alter pixel values drastically

Deformation

Objects can bend, stretch, change shape

Occlusion

Objects can be partially hidden

Background Clutter

Object blends into surroundings

Intra-class Variation

Same class, vastly different appearance

Machine Learning: Data-Driven Approach

1. Collect a dataset of images and labels
2. Use Machine Learning to train a classifier
3. Evaluate the classifier on new images

```
def train(images, labels):
  # Machine learning!
  return model
```

```
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

Example training set

airplane

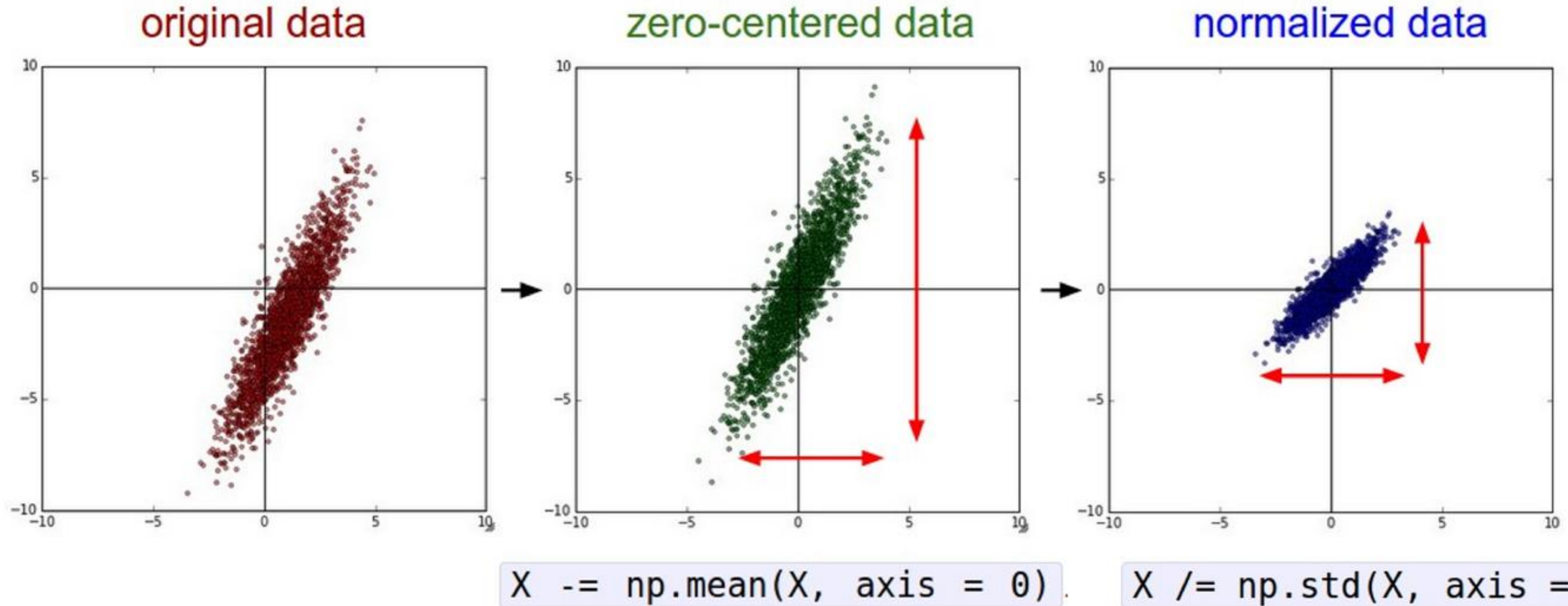
automobile

bird

cat

deer

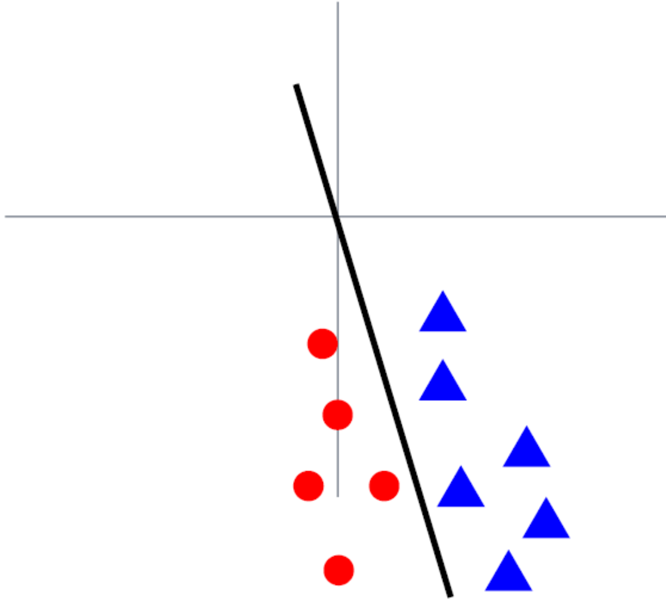
Data Preprocessing



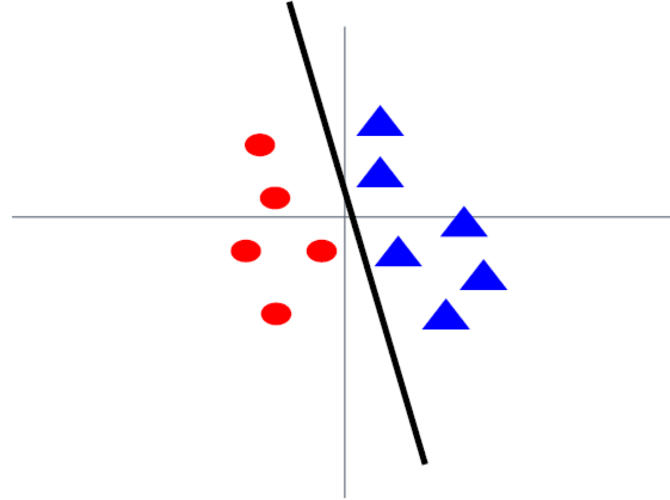
(Assume X [NxD] is data matrix, each example in a row)

Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data Preprocessing

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Not common to do PCA or whitening

Image Classification Dataset: MNIST

MNIST Dataset

Classes

10 classes (0~9)

Image Size

28 x 28 grayscale

Training / Test

60,000 / 10,000

Image Classification Datasets: CIFAR-100

CIFAR-100 Dataset

Classes

100 classes / 20 superclasses

Image Size

32 x 32 RGB

Training / Test

50,000 / 10,000

Image Classification Datasets: ImageNet

ImageNet (ILSVRC)

Classes

1,000 classes

Image Size

Variable size (resized to 256x256)

Training / Validation / Test

1.2M / 50K / 100K

Image Classification Datasets: MIT Places

MIT Places365

Classes

365 scene classes

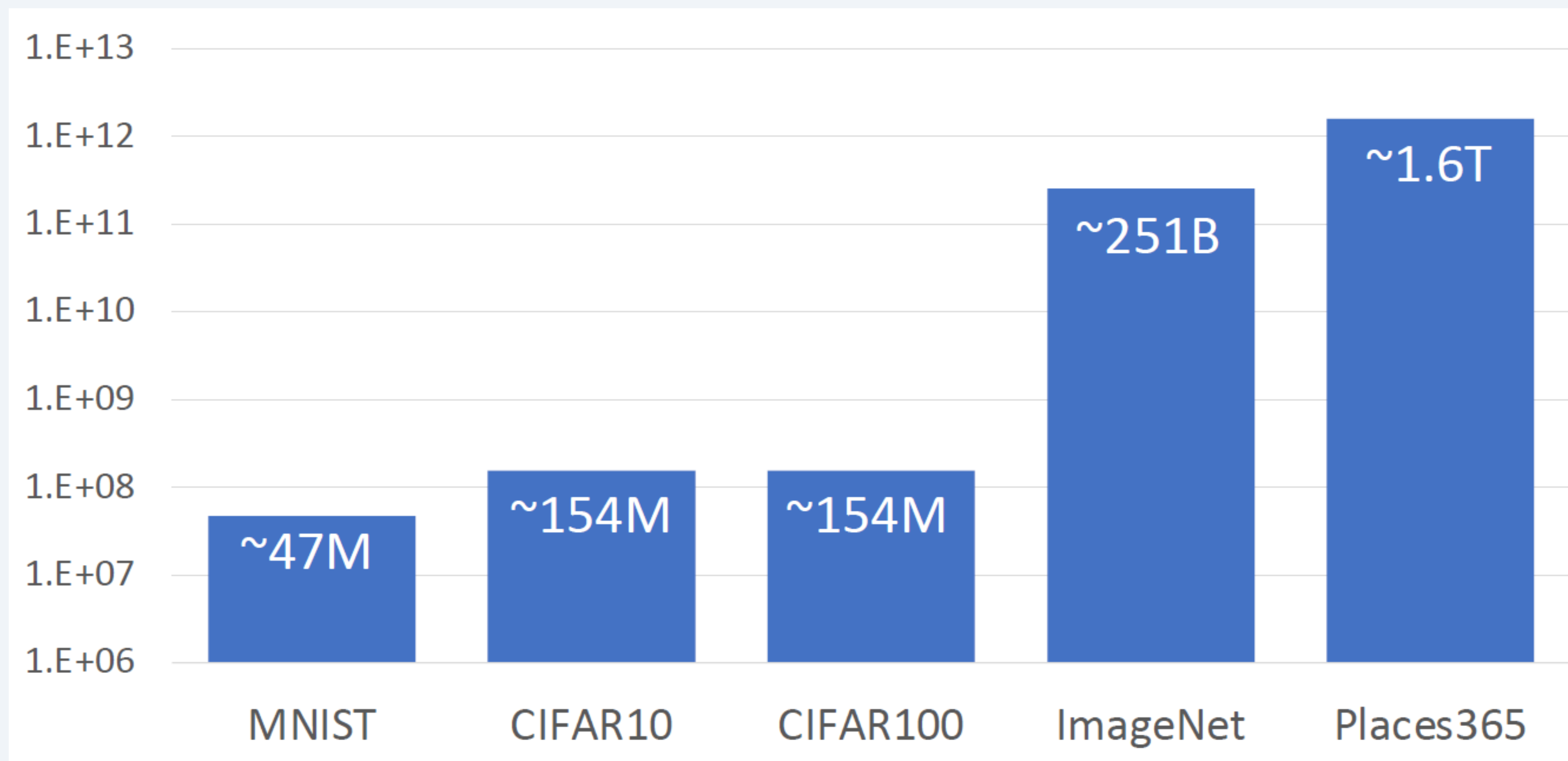
Image Size

256 x 256

Training / Validation / Test

~8M / 18.25K / 328.5K

Classification Datasets: Training Pixels



Hyperparameters

Key Questions

What learning rate should I use?
What regularization strength?
How many hidden units? How many layers?

Definition

These are **hyperparameters** — choices about the algorithm that are set before learning begins, not learned from the training data.

Key Takeaway: Very problem/dataset-dependent. Must try them all out and see what works best.

Train / Validation / Test Split

Training Set (70~80%)	Val (10~15%)	Test (10~15%)

Training Set

Model learns from this data. Weights are updated via backprop.

Validation Set

Tune hyperparameters. Monitor overfitting during training.

Test Set

Final evaluation only. Never used during training or tuning.

Common mistake: tuning hyperparameters on the test set → you're overfitting to it without knowing.

Dataset Split

1 **Idea #1:** Train on full data, evaluate on training set — BAD

K=1 always works perfectly on training data

2 **Idea #2:** Split data into train and test sets — BAD

test

No idea how algorithm will perform on new data

3 **Idea #3:** Split into train, validation, and test sets — Better!

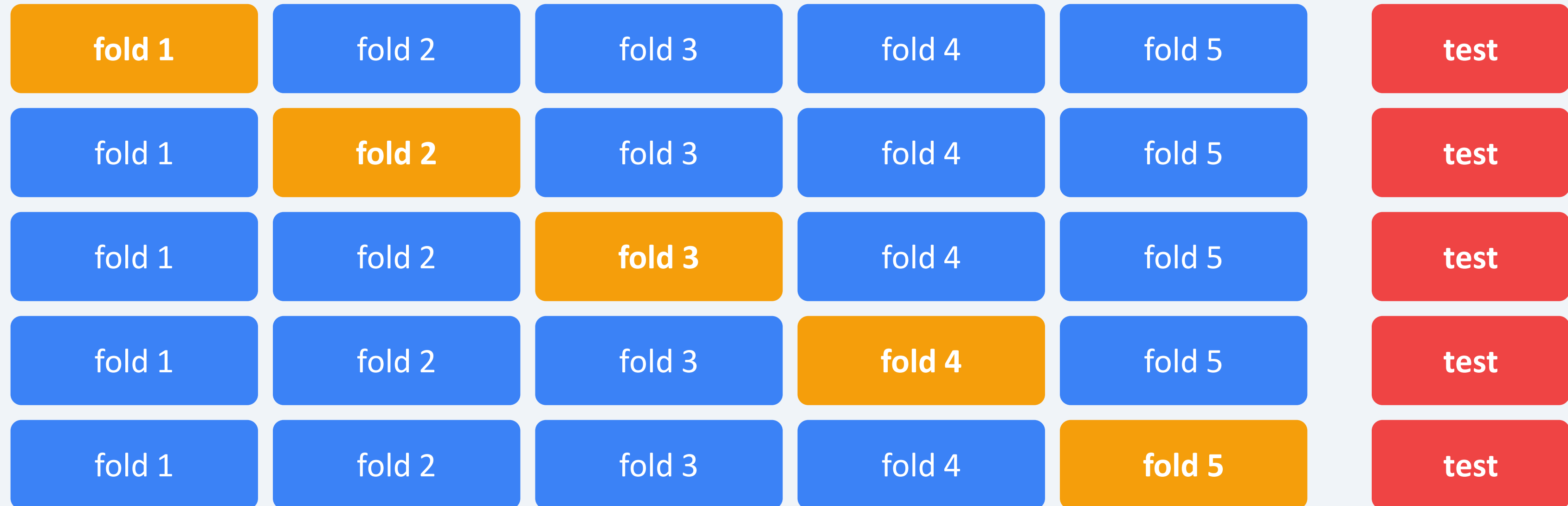
val

test

Choose hyperparameters on val, evaluate once on test

K-fold Cross-Validation

5-Fold Cross-Validation: Each fold takes turns being the validation set



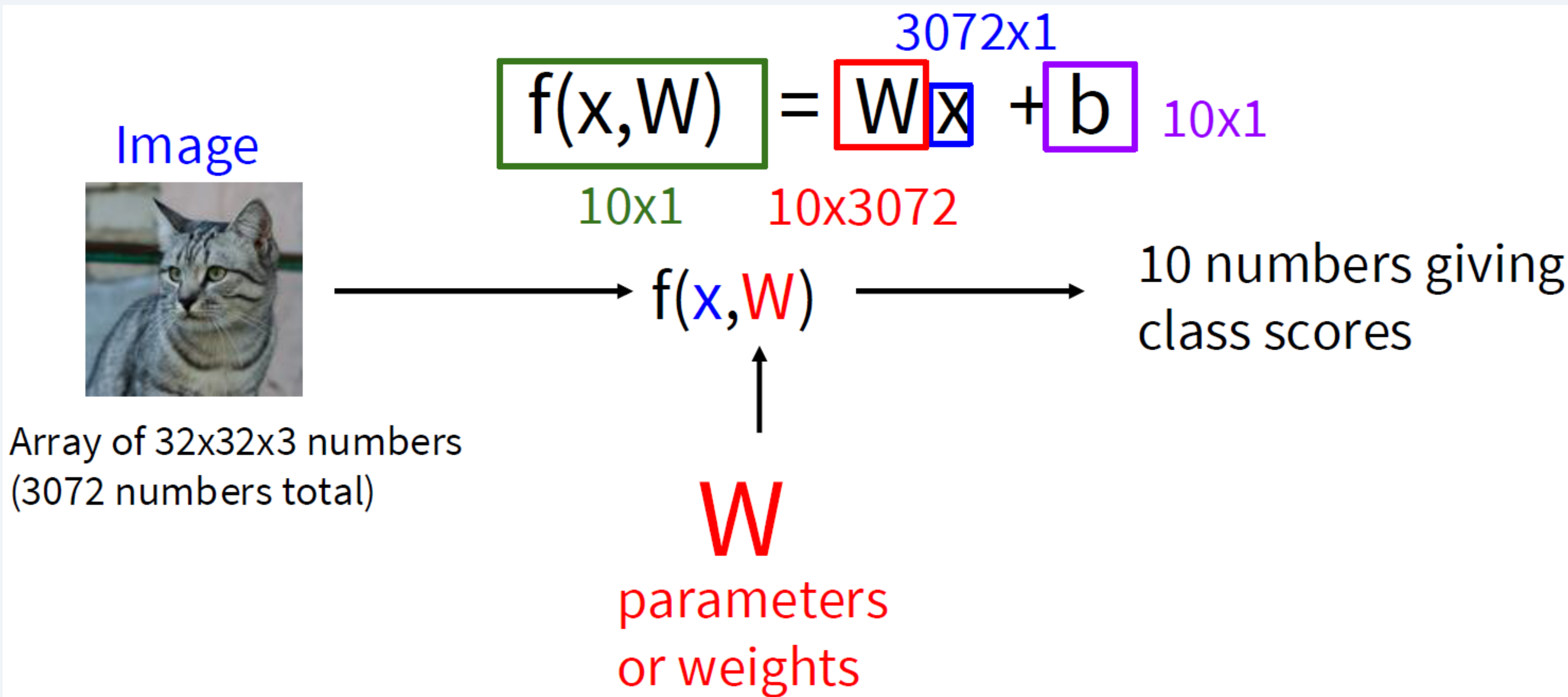
Useful for small datasets, but not used too frequently in deep learning.

Part 2

Linear Classifier to Neural Networks

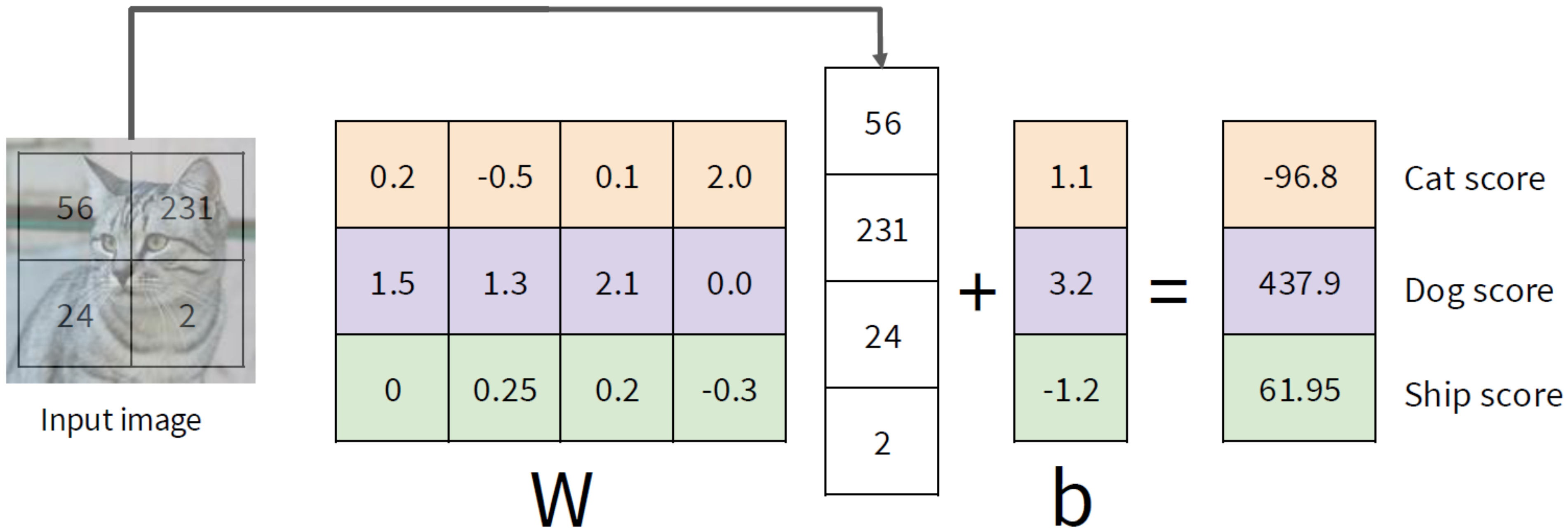
From Wx+b to MLP; why we need depth and non-linearity

Linear Classifier



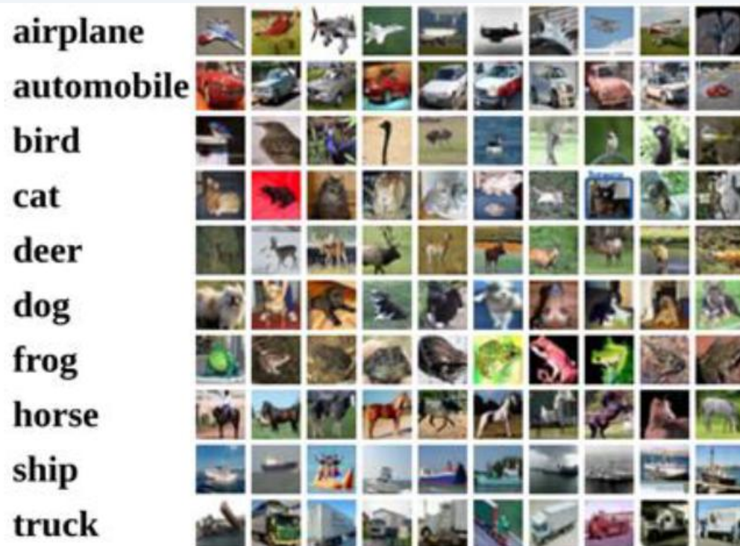
Key Idea: Image pixels are multiplied by weight matrix W and added bias b to produce class scores. W and b are learned parameters.

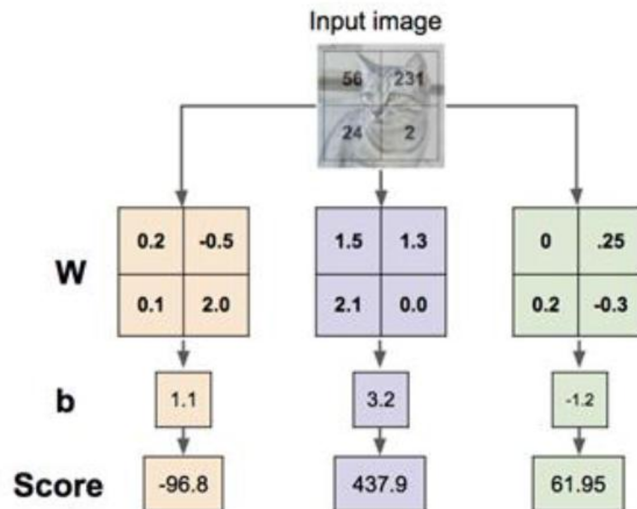
Linear Classification: Algebraic Viewpoint



Interpretation: Each row of W acts as a template for a class. The dot product measures similarity between the image and each template.

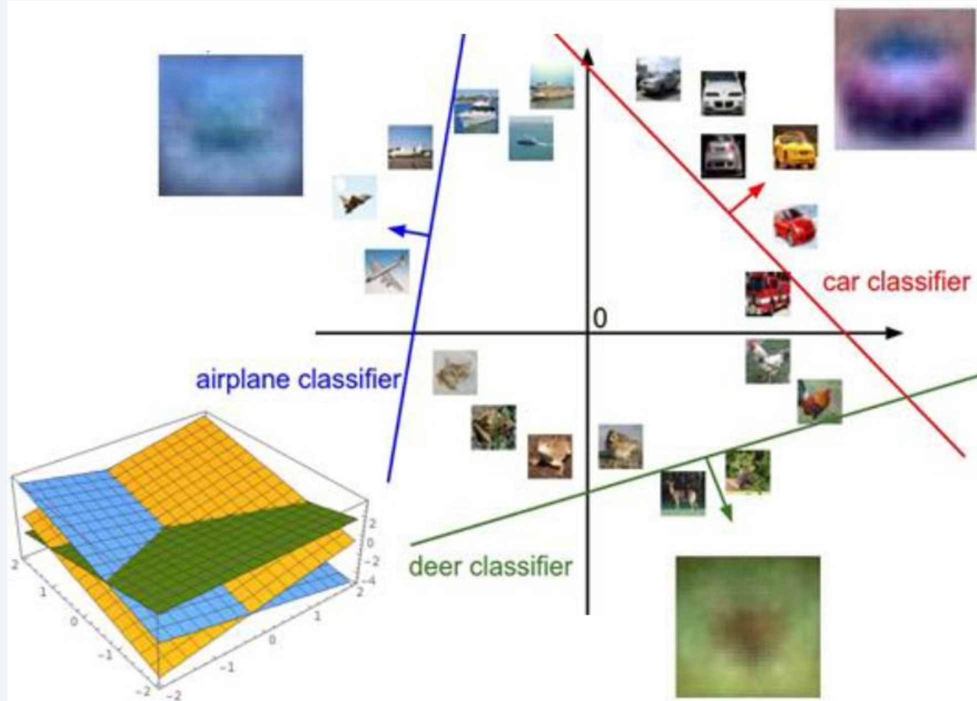
Linear Classification: Visual Viewpoint





Interpretation: Each row of the weight matrix W corresponds to a template for one class. The score for a class is the dot product (similarity) between the image and that template. The learned templates are blurry because the linear classifier can only learn one template per class.

Linear Classification: Geometric Viewpoint



Plot created using Wolfram Cloud

$$f(x,W) = Wx + b$$

Array of 32x32x3 numbers
(3072 numbers total)

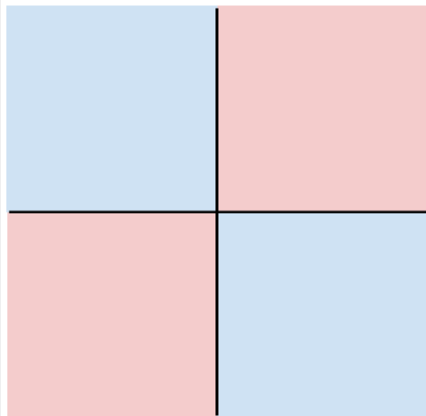
Cat image by Nikita is licensed under CC-BY 2.0

Interpretation: Each image is a point in high-dimensional pixel space. A linear classifier draws hyperplanes that carve this space into regions, one per class. W defines the orientation and b shifts the position of each decision boundary.

Limits of Linear Classifier

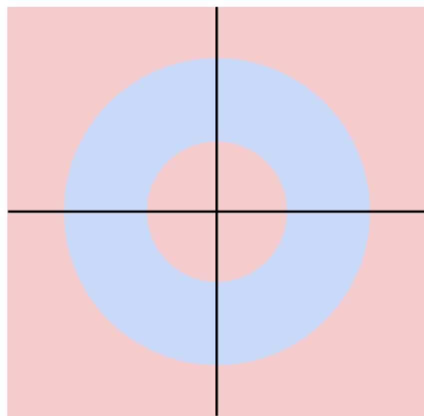
Class 1:
First and third quadrants

Class 2:
Second and fourth quadrants



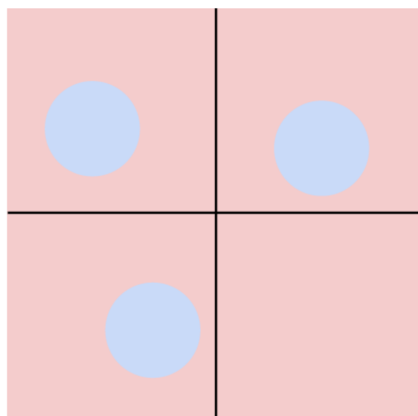
Class 1:
1 <= L2 norm <= 2

Class 2:
Everything else



Class 1:
Three modes

Class 2:
Everything else



The Solution: Go Deeper

1. **Add non-linearity**

   Apply activation function (ReLU, Sigmoid, etc.) after each linear transformation

2. **Stack layers**

   Input → Linear → ReLU → Linear → ReLU → Output = Multi-Layer Perceptron

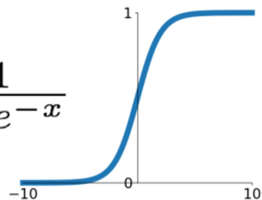
3. **Learn features**

   Each layer learns increasingly abstract representations → can model complex boundaries

Activation Functions

Sigmoid

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

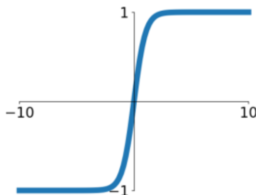


Output: (0, 1)

Squashes values to (0,1). Prone to vanishing gradients and outputs are not zero-centered.

tanh

$\tanh(x)$

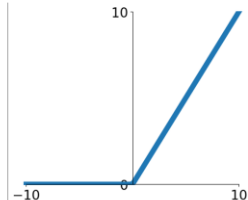


Output: (−1, 1)

Zero-centered output. Still suffers from vanishing gradients at saturation regions.

ReLU

$\max(0, x)$

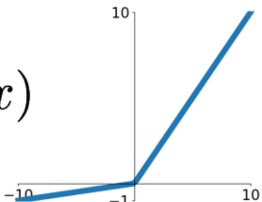


Output: [0, ∞)

Simple and fast. Most widely used. Can cause 'dead neurons' when units get stuck at zero.

Leaky ReLU

$\max(0.1x, x)$

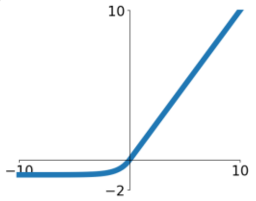


Output: (−∞, ∞)

Fixes dead neuron problem by allowing small negative slope. α is a small constant.

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

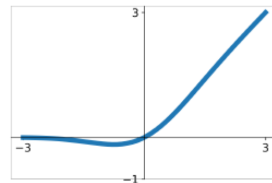


Output: (−α, ∞)

Smooth negative part with zero-centered outputs. More expensive to compute than ReLU.

GELU

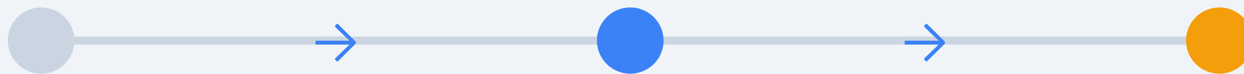
$\approx x\sigma(1.702x)$



Output: ≈ (−0.17, ∞)

Used in Transformers. Smooth approximation that weights inputs by their probability.

From Linear Classifier to MLP



Linear Classifier

$$f = Wx + b$$

One layer
Linear boundaries only

2-Layer Network

$$f = W_2 \cdot \sigma(W_1 x + b_1) + b_2$$

Hidden layer + activation
Can learn non-linear features

Deep Network

$$f = W_n \cdot \sigma(\ldots \sigma(W_1 x) \ldots)$$

Multiple layers
Hierarchical feature learning

Key Insight Depth + non-linearity = ability to learn any function. Next: CNNs add spatial structure.

Part 3

Loss & Optimization

How we measure "how wrong" the model is. & How we update weights to minimize loss.

Choosing a good W

$f(x,W) = Wx + b$

airplane	-3.45	-0.51	3.42
automobile	-8.87	**6.04**	4.64
bird	0.09	5.31	2.65
cat	**2.9**	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	**-4.34**
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

TODO:

1. Use a **loss function** to quantify how good a value of W is

2. Find a W that minimizes the loss function (**optimization**)

Loss Function

A **loss function** tells how good our current classifier is

Low loss = good classifier
High loss = bad classifier

(Also called: **objective function**; **cost function**)

Negative loss function sometimes called **reward function**, **profit function**, **utility function**, **fitness function**, etc

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^{N}$$

Where x_i is image and y_i is (integer) label

Loss for a single example is

$$L_i(f(x_i, W), y_i)$$

Loss for the dataset is average of per-example losses:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Loss Functions

Classification

Cross-Entropy Loss

$$L = -\sum y_i \log(p_i)$$

Here *i* indexes over classes: $y_i \in \{0,1\}$ is the one-hot label, p_i is the predicted probability. Since only correct class has $y_i = 1$, this reduces to:

L = −log p(correct class)

💡 PyTorch nn.CrossEntropyLoss = Softmax + CE

Why not MSE? CE gradient is stronger when prediction is very wrong → faster learning.

Regression

MSE (L2 Loss)

$$L = \frac{1}{n}\sum (y_i - \hat{y}_i)^2$$

Penalizes large errors quadratically. Sensitive to outliers, but smooth and differentiable everywhere → easy to optimize.

MAE (L1 Loss)

$$L = \frac{1}{n}\sum |y_i - \hat{y}_i|$$

More robust to outliers.
Used in bounding box regression.

Softmax Classifier

Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

$$L_i = -\log P(Y = y_i | X = x_i)$$

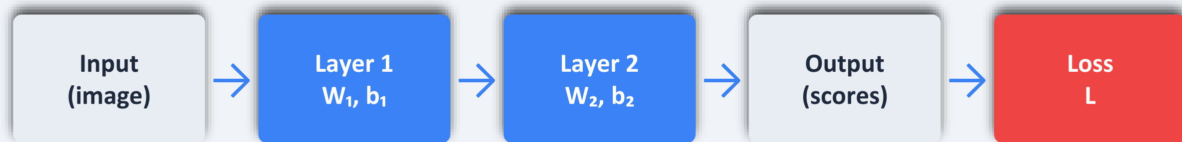
	Unnormalized log-probabilities / logits		unnormalized probabilities		probabilities		Correct probs
cat	3.2	exp →	24.5	normalize →	0.13	→ compare ←	1.00
car	5.1		164.0		0.87		0.00
frog	-1.7		0.18		0.00		0.00

Cross Entropy

$$H(P, Q) = H(p) + D_{KL}(P \| Q)$$

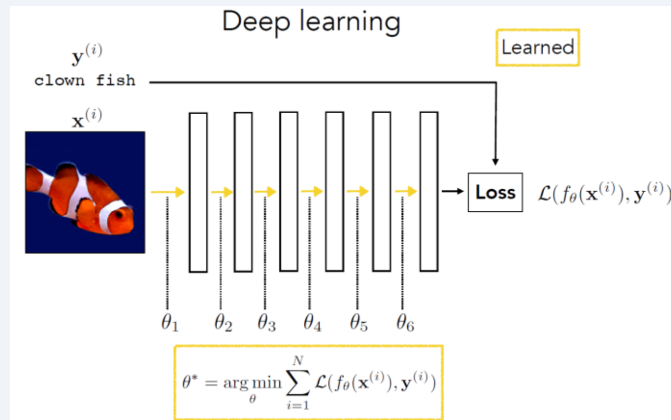
How Does a Model Learn?

Three steps, repeated thousands of times.



Forward →

← Backward (gradients flow back through each layer)



1 Forward Pass

Input passes through layers.
Prediction → compute loss.

```
outputs = model(x)
loss = loss_fn(outputs, y)
```

2 Backward Pass

Compute ∂L/∂W for each layer:
"how much did each W cause error?"

```
loss.backward()
```

3 Update Weights

Nudge W in the direction
that **reduces** loss.

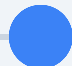
```
optimizer.step()
```

The math is handled by PyTorch. You need to understand the flow: forward → backward → update.

From SGD to AdamW

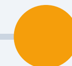
1951

1964



2011/2012

2014/2017



SGD

Basic gradient descent.
Noisy but simple
to implement.

$w = w - lr \times \nabla L$

+Momentum

Accumulates velocity.
Smooths oscillations.
"Rolling ball" analogy.

$w = w - lr \cdot vv$
$= \beta v + \nabla L$

AdaGrad → RMSProp

Per-parameter lr.
Adapts based on
gradient history.

Sparse grads
→ larger steps

Adam / AdamW

Momentum + RMSProp
combined. Best default
for deep learning.

★ Default choice

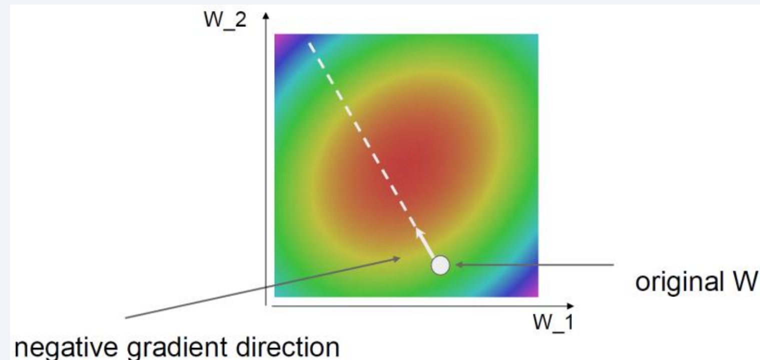
Practical tip: Start with AdamW. Fine-tune with SGD+Momentum+Cosine for best performance.

Gradient Descent & SGD

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

▲ *Computes gradient on full data → slow but accurate updates*




```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

▲ *Approximates with mini-batch (256) → fast but noisy updates (SGD)*

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

Loss Function (Full Batch)
Average loss over all N samples +
Regularization (λR(W) to prevent overfitting)

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Gradient (∇L)
Sum of per-sample gradients
→ Determines W update direction

SGD + Momentum / RMSProp

SGD (Vanilla)

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

▲ *Basic gradient update: w = w - lr × ∇L*

+ Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

▲ *Accumulates velocity(v) → faster convergence via inertia*

RMSProp

▼ *Adaptive lr per param: large grad → slower, small grad → faster*

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Key Comparison

SGD
Uses only gradient direction
✗ Zig-zag oscillation

Momentum
Accumulates velocity for inertia
✓ Reduces oscillation

RMSProp
Per-param lr scaling (large grad → small step)
✓ Strong on sparse data

Adam, AdamW

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

▲ Adam pseudo-code

▼ AdamW

Standard Adam computes L2 here

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdamW (Weight Decay) adds term here

Adam (L2 Regularization)

`dx = ∇L + λw` → then adaptive update

decay entangled with gradient moments

AdamW (Decoupled Weight Decay)

`w -= lr · (update + λw)`

decay applied directly to weights, not gradient

AdamW: decouples weight decay from gradient → better regularization, now the default choice

Adam Deep Dive: Momentum & RMSProp

Adam = Momentum (1st moment) + RMSProp (2nd moment) + Bias Correction

Momentum (1st Moment)

Formula

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

Intent

Exponential moving average of gradients. Smooths noisy gradients by accumulating past direction like a rolling ball.

Effect

Accelerates through consistent gradient directions, dampens oscillation in noisy ones. Typical $\beta_1 = 0.9$

RMSProp (2nd Moment)

Formula

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

Intent

Tracks how large gradients have been (variance of gradients). Divides update by $\sqrt{v}$ to normalize step size per parameter.

Effect

Per-parameter adaptive learning rate.
Large gradients → smaller step (stabilize).
Small gradients → larger step (speed up).
Typical $\beta_2 = 0.999$

Adam Deep Dive: Bias Correction & Weight Decay

Fixing initialization bias + decoupling regularization from the optimizer

Bias Correction

Formula

$$\hat{v}_t = v_t \,/\, (1 - \beta_2{}^t)\hat{m}_t = m_t \,/\, (1 - \beta_1{}^t)$$

Why needed?

Both m and v start at 0. In early steps they haven't "warmed up" yet → estimates are biased toward zero.

Effect

Dividing by $(1 - \beta^t)$ compensates: large correction early (t small, $\beta^t \approx 1$), nearly no effect later ($\beta^t \to 0$).

Weight Decay (AdamW)

Adam (L2) — Coupled

$dx = \nabla L + \lambda w$ → then adaptive update

Decay mixes into gradient → distorted by adaptive lr scaling

AdamW — Decoupled

$$w \mathrel{-}= lr \cdot (update + \lambda w)$$

Decay applied directly to weights, not through gradient

Why it matters

Cleaner regularization — decay strength is not distorted by Adam's moment estimates.

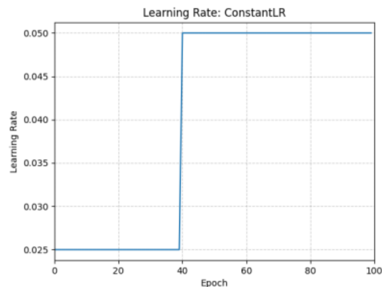
Now the default in most frameworks.

Learning Rate Scheduling

The learning rate is often the single most important hyperparameter.

Baseline

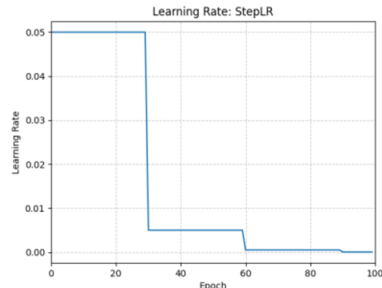
Constant



Fixed lr throughout. Simplest but rarely optimal.

Classic

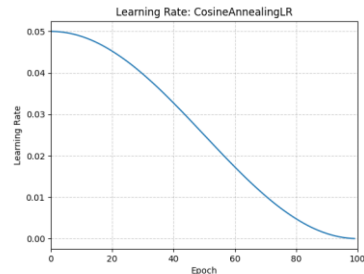
Step Decay



Reduce lr by factor every N epochs (e.g., ×0.1 at epoch 30, 60, 90).

Recommended

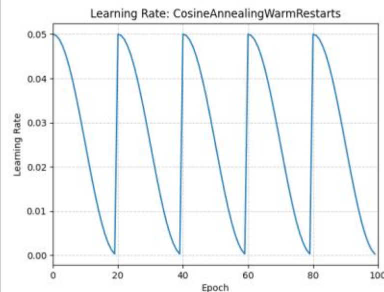
Cosine Annealing



Smooth decay following cosine curve. Most popular in modern training.

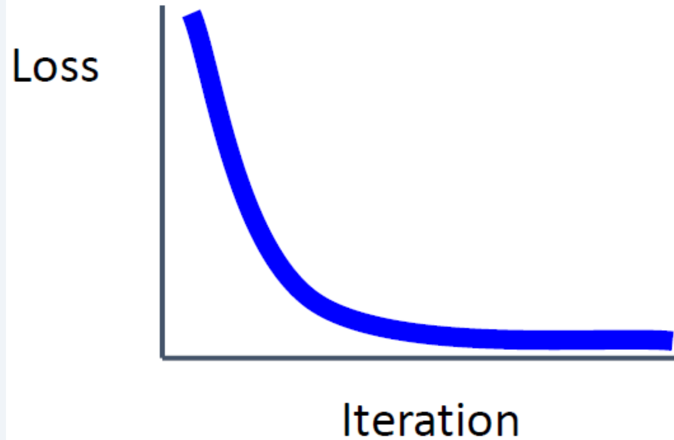
Modern default

Warmup + Cosine

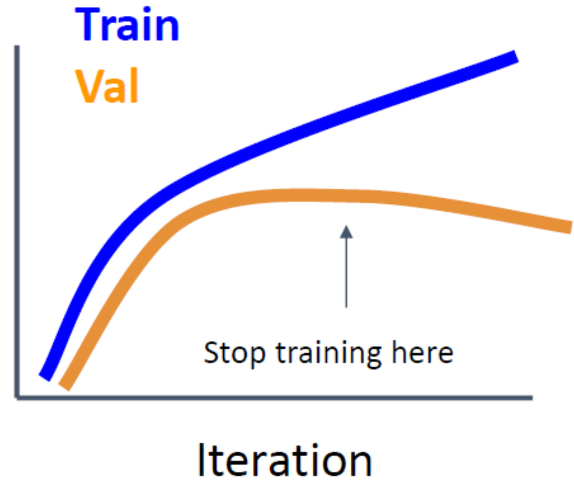


Start low, ramp up, then cosine decay. Essential for Transformers/ViT.

How long to train?

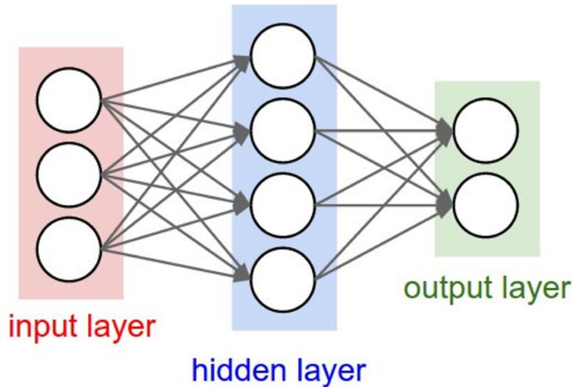


Accuracy



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val. **Always a good idea to do this!**

Weight Initialization



Why It Matters

Bad init → vanishing/exploding gradients
Good init → stable activations across layers

💡 Rule of Thumb

ReLU → **He (Kaiming)**
Sigmoid/Tanh → **Xavier (Glorot)**

Zero Init ⚠ Avoid

$$W = 0$$

All weights zero. Neurons learn identically (symmetry problem).

Random Normal Basic

$$W \sim N(0, 0.01)$$

Small random values. Works for shallow nets, fails for deep.

Xavier / Glorot Sigmoid/Tanh

$$W \sim N(0, \frac{2}{n_{in} + n_{out}})$$

Balanced variance for sigmoid/tanh. Keeps signal stable.

He / Kaiming ★ ReLU default

$$W \sim N(0, \frac{2}{n_{in}})$$

Designed for ReLU. Accounts for half-zeroed activations.

LeCun Init SELU

$$W \sim N(0, \frac{1}{n_{in}})$$

For SELU activation. Preserves self-normalizing property.

Orthogonal Init RNN/Deep

$$W = orthogonal\ matrix$$

Preserves gradient norm. Great for RNNs and very deep nets.

Weight Init in Transformers (Supp.)

GPT / BERT / ViT use different strategies than classic CNNs

COMPONENT-LEVEL INITIALIZATION

Linear Layers
`N(0, 0.02)`

Small fixed std, LayerNorm handles rest

Residual Proj
`N(0, 0.02/√2N)`

Scale down by depth to prevent amplification

Embeddings
`N(0, 0.02)`

Token + position. Prevent large init magnitudes

LayerNorm
`γ=1, β=0`

Identity init: normalize only, no shift/scale

FFN (GELU)
`He / Kaiming`

Fan-in scaling for ReLU/GELU activations

Key Insight Transformers rely on LayerNorm + AdamW, so N(0, 0.02) works surprisingly well.
The critical trick is **scaling residual projections by 1/√(2N)** to prevent signal explosion in deep models.

MODEL COMPARISON

GPT-2 / GPT-3
Decoder-only LLM

All weights **N(0, 0.02)**
Residual proj **N(0, 0.02/√2N)**
Embedding **N(0, 0.02)**
Activation **GELU**

BERT
Encoder-only (Bidirectional)

All weights **N(0, 0.02)**
Residual proj **Standard (no scale)**
Embedding **N(0, 0.02)**
Activation **GELU**

Original Transformer
Vaswani et al. 2017

All weights **Xavier Uniform**
Residual proj **Standard**
Embedding **d^(-0.5) scaling**
Activation **ReLU**

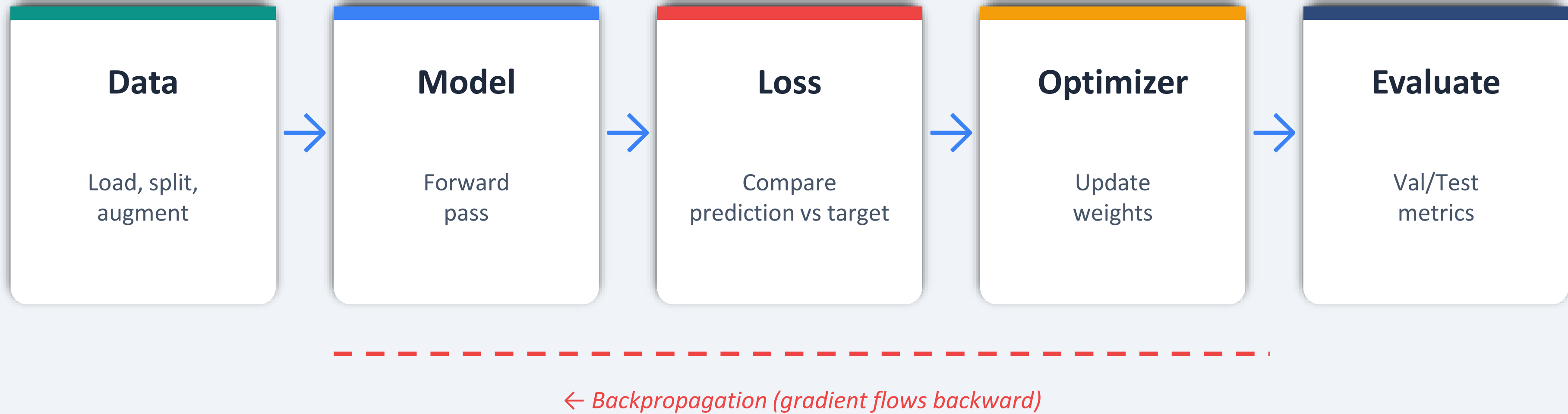
PyTorch default: nn.Linear uses Kaiming Uniform | HuggingFace Transformers: N(0, 0.02)

Part 4

Regularization & Training Tricks

How to train your model better.

The DL Training Pipeline



Key Terms Epoch: one full pass through training data | Batch: subset of data per step | Iteration: one weight update | Learning Rate: step size for optimization

Diagnosing Your Model

Read the loss curves. They tell you what to do next.

Underfitting

Train loss **HIGH**
Val loss **HIGH**
Gap **Small**

Model is too simple or learning rate is too low to capture patterns.

ACTIONS

- → Bigger model (more layers/units)
- → Train longer (more epochs)
- → Increase learning rate

Good Fit

Train loss **LOW**
Val loss **LOW**
Gap **Small**

Model generalizes well. Train and val loss converge together.

ACTIONS

- → You're done! Deploy it.
- → Fine-tune learning rate
- → Consider early stopping point

Overfitting

Train loss **LOW**
Val loss **HIGH ↑**
Gap **Large**

Model memorizes training data but fails on unseen examples.

ACTIONS

- → More data / augmentation
- → Add regularization (L2, Dropout)
- → Use a smaller model

Pro Tip Always check loss curves **before** tuning hyperparameters. If train loss isn't decreasing, fix that first (underfitting).
Only worry about val loss gap (overfitting) once train loss is low enough.

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

More complex:

Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

Why Regularize? Express preferences over weights to reduce model complexity

Prevent overfitting by penalizing large weights | Improve generalization to unseen data | Add curvature for better optimization

Key Regularization Techniques

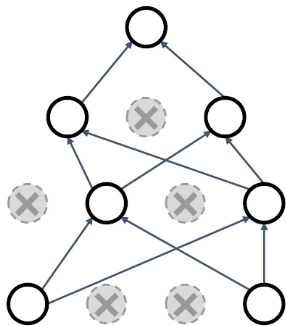
Weight Decay (L2 Reg)

Add penalty for large weights:

$$L_{total} = L + \lambda||w||^2$$

Prevents any single weight from becoming too dominant.

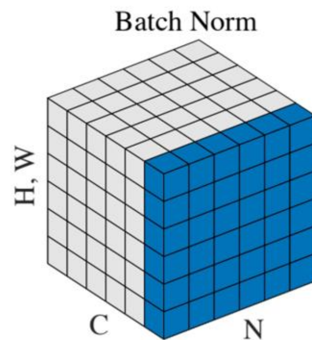
Dropout



Randomly disable neurons during
training (e.g., p=0.5).

Forces redundant representations.
Acts like model ensemble.
Disabled at test time.

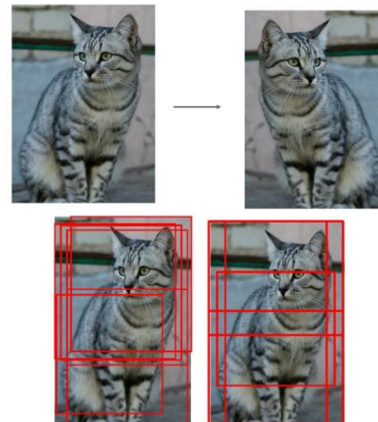
Batch Normalization



Normalize activations within each mini-batch.

Stabilizes training.
Allows higher learning rates.
Slight regularization effect.

Data Augmentation



Random crop, flip, rotation, color jitter, etc.

"Free" extra training data.
Teaches invariances.
Always use as first defense.

Practical Debugging Checklist

When your model isn't working, check these in order.

1	**Look at the data**	Are labels correct? Is preprocessing right? Visualize samples.
2	**Overfit a tiny batch first**	Can your model memorize 1 batch? If not, there's a bug.
3	**Check the loss at initialization**	For CE with C classes, initial loss should be ≈ -log(1/C).
4	**Start with a known architecture**	Don't innovate on model AND training at the same time.
5	**Tune learning rate first**	Too high → loss explodes. Too low → loss barely moves.
6	**Add complexity gradually**	More data → augmentation → regularization → bigger model.

Ref: Andrej Karpathy, "A Recipe for Training Neural Networks" (2019)

Part 5

PyTorch Walkthrough

The code pattern you'll use for the rest of this course.

PyTorch Building Blocks

Dataset / DataLoader

```
dataset = CIFAR10(...)
loader = DataLoader(
  dataset, batch_size=64,
  shuffle=True)
```

Pipeline: Data

nn.Module

```
class Model(nn.Module):
  def __init__(self):
    self.conv = nn.Conv2d(...)
  def forward(self, x):
    return self.conv(x)
```

Pipeline: Model

Loss + Optimizer

```
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(
  model.parameters(),
  lr=1e-3)
```

Pipeline: Loss + Optimizer

The Training Loop

These 4 lines are the core. Everything else is built around them.

```
for epoch in range(num_epochs):
  for images, labels in train_loader:

    # Forward pass
    outputs = model(images)
    loss = loss_fn(outputs, labels)

    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

  # Validation
  model.eval()
  with torch.no_grad():
    val_loss = evaluate(model, val_loader)
  model.train()
```

1. Forward

Input → Model → Prediction

2. Loss

Compare prediction vs target

3. Backward

Compute gradients

4. Update

Adjust weights

This exact pattern works for CNN, ViT, Diffusion — everything in this course.

Summary & Resources

What We Covered

- Images as tensors, semantic gap, benchmark datasets
- Linear classifier → MLP (why depth matters)
- Loss functions (CE, MSE) + Backprop intuition
- Optimizers (SGD → Adam) + LR scheduling
- Regularization (Dropout, BN, Augmentation)
- Overfitting diagnosis + debugging checklist
- PyTorch training loop

Further Reading & Self-study

Karpathy, "A Recipe for Training NNs"
karpathy.github.io/2019/04/25/recipe/

Michigan EECS 498 (YouTube, free):
L3 Linear Classifiers | L4 Optimization
L6 Backpropagation | L9-10 Training NNs

CS231n Notes: cs231n.github.io

PyTorch Tutorial (self-study):
CS231n Colab Notebook

Next Week Week 3: Convolutional Neural Networks — From convolution operation to ResNet

Acknowledgments

Some slides and figures in this course are adapted from or inspired by the following open resources.

Stanford CS231n: Deep Learning for Computer Vision (Spring 2025)

Fei-Fei Li, Ehsan Adeli, et al. | cs231n.stanford.edu

UMich EECS 498/598: Deep Learning for Computer Vision (Winter 2022)

Justin Johnson | web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

MIT 6.8300: Advances in Computer Vision (Spring 2025)

Vincent Sitzmann | scenerepresentations.org/courses/2025/spring/advances-in-cv/

MIT 6.7960: Deep Learning (Fall 2024)

Phillip Isola, Sara Beery, Jeremy Bernstein | ocw.mit.edu/courses/6-7960-deep-learning-fall-2024/

CMU 16-824: Visual Learning and Recognition (Fall 2025)

Jun-Yan Zhu | visual-learning.cs.cmu.edu