03 CNNs, ResNet, regularisation, normalisation

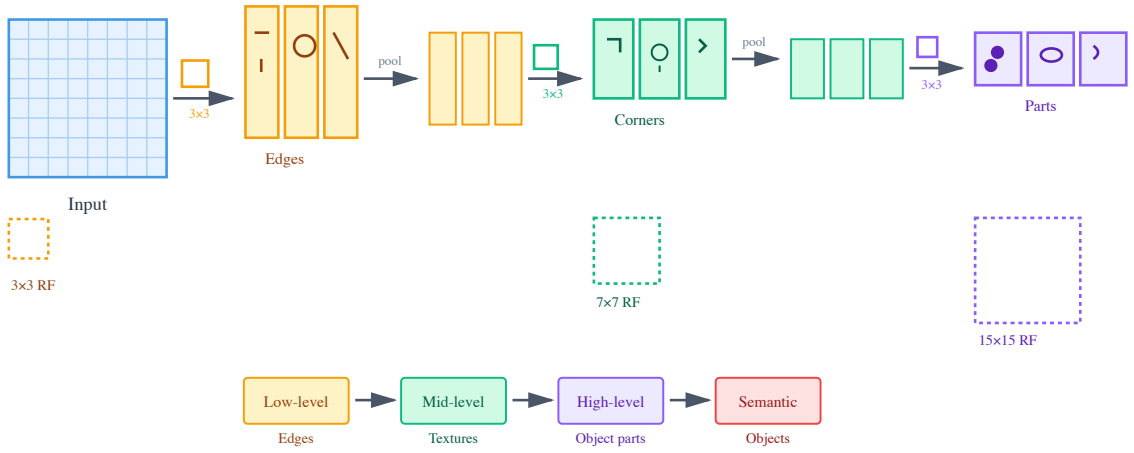
Outline

1. **CNN Quick Review**
2. **ResNet and Residual Learning**
3. **Evaluation Metrics & Benchmarks**
4. **Regularisation Fundamentals**
5. **Normalisation Techniques**

Ethics in machine learning / computer science / informatics?

dr Jakub Piwowar, sociologist and cultural researcher, SWPS, researches on ethics / learning about ethics among the computer science students. Could you, please, fill out an anonymous questionnaire https://forms.gle/m99Utuyi3aiwQEdz6 on that subject?

CNN fundamentals



Core Components

```
Input → [Conv → ReLU → Pool] × N → Flatten → FC → Output
```

cnn-priors.html

Key Properties:

- **Translation Invariance**: Detecting features regardless of position
- **Local Connectivity**: Each neuron connects to local region
- **Weight Sharing**: Same filter across entire image
- **Hierarchical Features**: Edges → Textures → Parts → Objects

Inductive Bias of CNNs:

- **Locality**: Nearby pixels more related
- **Translation Equivariance**: f(shift(x)) = shift(f(x))
- **Compositionality**: Complex patterns from simple ones

> 💡 **Remember**: CNNs encode strong priors about visual data structure

---

the problem CNNs were solving

Before Deep CNNs (Pre-2012):

```
Traditional Pipeline:
Image → Hand-crafted Features → Classifier
       (SIFT, HOG, etc.)      (SVM, RF)
```

After AlexNet (2012+):

```
End-to-End Learning:
Image → Learned Features → Learned Classifier
       (Conv layers)      (FC layers)
```

---

the priors encoded by CNNs

What is a Prior in ML?

A prior is an assumption built into the model architecture about the structure of the solution space.

CNN's Four Key Priors:

1. Spatial Locality Prior

Assumption: Pixels that are close together are more likely to be related

```
# A 3x3 conv filter assumes relevant info is within 3x3 neighborhood
conv = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3)
# NOT looking at pixel[0,0] and pixel[100,100] together initially
```

Real-world validity:

- ✅ Edges are local
- ✅ Textures are local patterns
- ❌ Long-range dependencies need deep networks

2. Translation Equivariance Prior

Assumption: The same pattern should be detected regardless of position

```
# Same filter slides across entire image
# Cat in top-left == Cat in bottom-right
filter_response[i,j] = same_filter @ image[i:i+k, j:j+k]
```

Real-world validity:

- ✅ Objects can appear anywhere
- ✅ Edges exist everywhere
- ❌ Position sometimes matters (face parts have expected locations)

3. Hierarchical Compositionality Prior

Assumption: Complex patterns are composed of simpler sub-patterns

```
Pixels → Edges → Corners → Parts → Objects
Layer 1   Layer 2   Layer 3   Layer 4   Layer 5
```

Example - Face Detection:

```
Round edges + Two circles + Line
    ↓           ↓            ↓
  Face outline  Eyes        Mouth
              ↓
            Face
```

4. Weight Sharing Prior

Assumption: The same feature detector works across the entire image

```
# Traditional approach: Different weights for each position
params_fully_connected = H × W × C × num_outputs  # Millions!

# CNN approach: Same weights everywhere
params_conv = k × k × C × num_filters  # Just thousands!
```

Real-world validity:

- ✅ An edge is an edge everywhere
- ✅ Massive parameter reduction
- ❌ Assumes spatial stationarity

CNN priors - a concrete example

Task: Detecting Horizontal Edges

Without CNN Priors (Fully Connected):

```
# Need to learn edge detection at EVERY position
fc_layer = nn.Linear(28*28, hidden)  # MNIST
# Must learn:
# - Edge at position (0,0)
# - Edge at position (0,1)
# - Edge at position (0,2)
# ... 784 different positions!
```

With CNN Priors (Convolutional):

```
# Learn ONE edge detector, use everywhere
sobel_horizontal = [[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]]
# This 3×3 filter works at ALL positions
```

The Power of Priors:

Aspect	Fully Connected	CNN
Parameters for edge detection	784 × hidden	3 × 3 = 9
Training samples needed	Must see edges everywhere	Edge at one position generalizes
Invariance	Must learn separately	Built-in
Failure mode	Memorizes positions	Learns concepts

When CNN Priors Fail:

```
# Example: Non-local dependencies
"Count all red pixels in image"  # Needs global view
"Is this image symmetric?"        # Needs to compare distant regions
"Parse the overall scene graph"   # Needs relational reasoning
```

Solution: Vision Transformers implement different priors

CNNs in one slide - the complete picture

The CNN Recipe 🎂

```
Raw Pixels → [Conv→ReLU→Pool]×N → Global Pool → FC → Classes
    ↓              ↓                    ↓          ↓       ↓
 Input      Feature Extraction      Aggregation  Decision  Output
```

What Makes CNNs Work:

Architectural Priors	Why It Helps	But Also...
🏘️ **Local connections**	Natural for vision	Misses global context
🔄 **Weight sharing**	100x fewer params	Assumes uniformity
📚 **Hierarchical depth**	Compositionality	Need many layers
📍 **Spatial structure**	Preserves topology	Fixed input size*

The Evolution:

```
1989: LeCun's LeNet     → Proved convolution works
1998: LeNet-5           → Digit recognition
2012: AlexNet           → ImageNet breakthrough (16.4% error)
```

```
2014: VGGNet            → Deeper is better (16 layers)
2015: ResNet            → MUCH deeper (152 layers, 3.57% error)
2017: DenseNet          → Connect everything
2019: EfficientNet      → Neural architecture search
2020: Vision Transformer → Maybe we don't need convolutions?
```

Key Problems CNNs Solve:

✅ **Parameter explosion** → Weight sharing
✅ **Translation variance** → Sliding windows
✅ **Learning hierarchy** → Deep stacking
✅ **Spatial reasoning** → Topology preservation

Key Problems CNNs Have:

❌ **Global dependencies** → Requires depth
❌ **Fixed receptive field** → Can miss large objects
❌ **Texture bias** → Relies on local patterns, not shape
❌ **Poor with rotations** → Not rotation equivariant*

> 💡 **The Deep Truth**: CNNs work because their priors match the structure of natural images. When they don't (graphs, text, audio), we need different architectures!

---

circuits in CNNs - how features compose

What are Circuits?

Zoom in: an introduction to circuits

Definition: Circuits are computational subgraphs within neural networks that implement specific functions

```
Input → [Feature Detectors] → [Combination Logic] → Output
           ↓                          ↓                  ↓
       Edge detector          AND/OR operations    Cat detector
```

Classic CNN Circuits Discovered:

1. Curve Detectors (Conv2)

```
Vertical edge   +   Horizontal edge
      ↓                    ↓
      └──────────┬─────────┘
```

```
                ↓
          Corner/Curve
```

2. Frequency Detectors (Conv3-4)

```
Multiple oriented edges → High frequency detector
Multiple smooth regions → Low frequency detector
```

3. Object Part Detectors (Conv4-5)

```
Curves + Textures → Eye detector
Lines + Symmetry → Face boundary
Circles + Position → Wheel detector
```

The Compositionality of Circuits:

```python
# Early layers: Simple, general circuits
conv1_circuits = ["edge", "color", "gabor"]

# Middle layers: Combination circuits
conv3_circuits = ["corner", "texture", "frequency"]

# Late layers: Semantic circuits
conv5_circuits = ["face", "car", "text"]

# Key insight: Later circuits REUSE earlier ones!
face_circuit = combine(eye_circuit, nose_circuit, mouth_circuit)
```

Circuit Discovery Methods:

1. **Feature Visualization**: Maximize neuron activation
2. **Attribution**: Which early features activate late features?
3. **Ablation**: Remove circuit, measure impact

Why Circuits Matter:

- 🔍 **Interpretability**: Understand what the network learned
- 🔧 **Debugging**: Find why model fails on certain inputs
- ✂️ **Pruning**: Remove unimportant circuits
- 🔄 **Transfer**: Reuse circuits across tasks

The Big Picture:

> CNNs automatically learn a **hierarchy of reusable visual circuits** that compose to form complex detectors. This compositionality is why transfer learning works!

We'll see how Transformers form different types of circuits (attention heads as information routing)!

ResNet and residual learning - the revolution in deep architecture

---

CNNs had issues:

- **degradation problem**: Deeper networks performed worse (not overfitting!)
- **vanishing gradients**: Deep networks hard to train
- **limited receptive field**: Hard to capture global context

We shall return to these problems in more detail later

This led to ResNet...

The Degradation Problem

The Paradox (He et al., 2015)

Observation: 56-layer network performed worse than 20-layer network on both training AND test sets!

This was not overfitting:

- Training error was HIGHER for deeper network
- Theoretically, deeper should be at least as good
- Deep network could learn identity mappings for extra layers

The Problem:

```
Optimization difficulty, not capacity limitation
```

Key Insight:

> "Is learning identity mappings hard for deep networks?"

Residual Learning Framework

Core Idea: Learn Residuals, Not Functions

Traditional Learning:

```
H(x) = Desired mapping
Network learns: H(x) directly
```

Residual Learning:

```
H(x) = F(x) + x     (Desired mapping)
Network learns: F(x) = H(x) - x  (Residual)
```

Why This Works:

1. **Easier Optimization**: Learning F(x) = 0 easier than H(x) = x
2. **Gradient Highway**: Gradients flow directly through shortcuts
3. **Implicit Regularization**: Encourages minimal changes

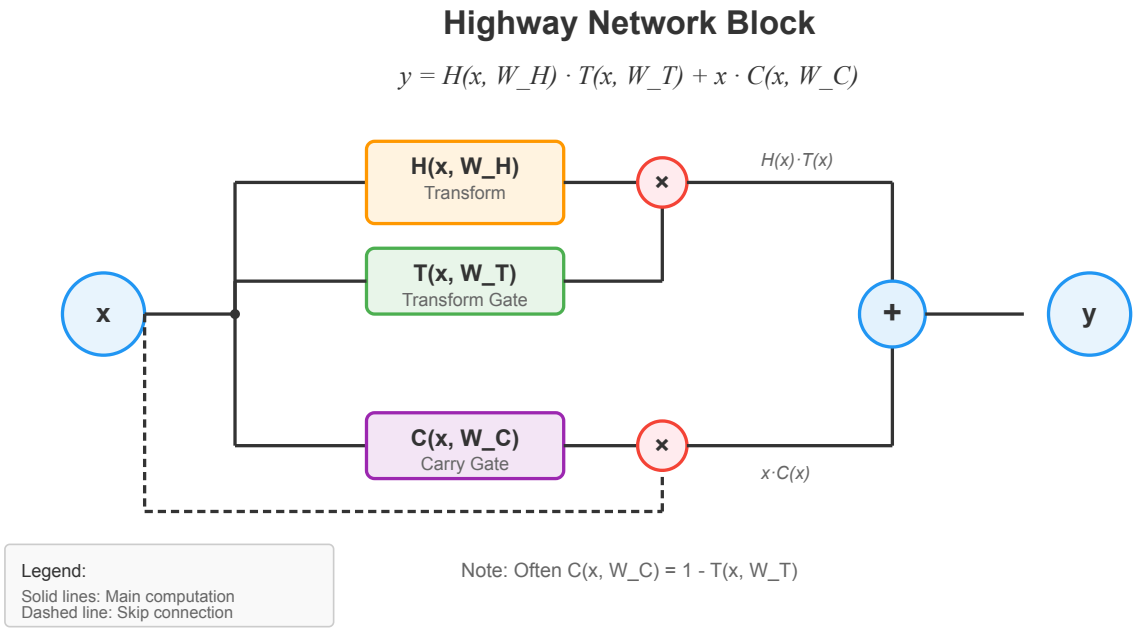
Mathematical Formulation:

$$y = \mathcal{F}(x, W_i) + x$$

where $\mathcal{F}$ is the residual mapping to be learned

Highway architecture

Similar to a Highway (Srivastava, Greff, Schmidhuber, *Highway networks*) network block



$$y = H(x, W_H) \cdot T(x, W_T) + x \cdot C(x, W_C)$$

where

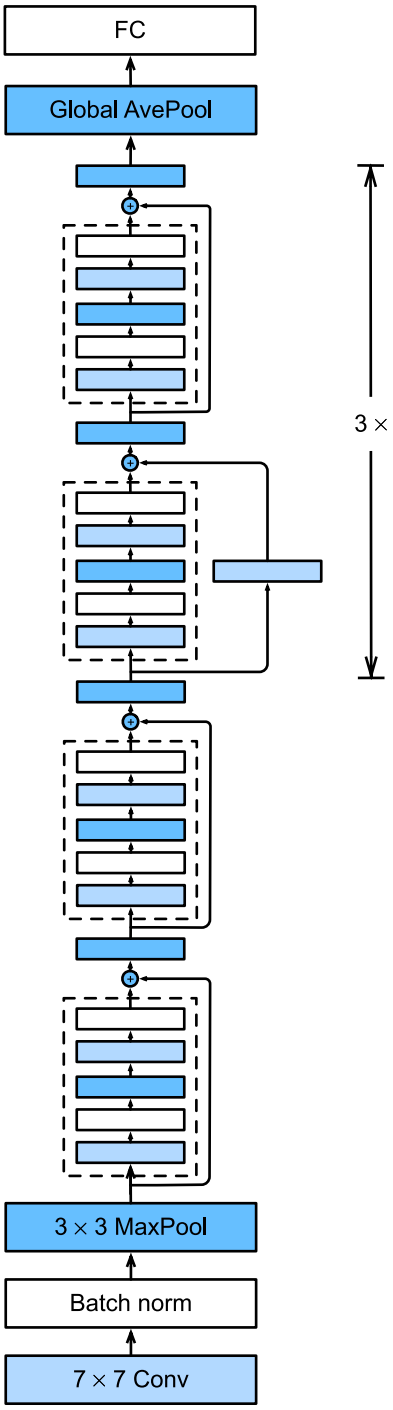
- $H(x, W_H)$: the hidden transformation
- $T(x, W_T)$: the transformation gate
- $C(x, W_C)$: the carry gate
  Setting $C = 1 - T$ we obtain

  $$y = H(x, W_H) \cdot T(x, W_T) + x \cdot (1 - T(x, W_C))$$

  which will lead to the *residual learning* paradigm.

- for very deep networks, the SGD learning can stall
- using $T(x) = \sigma(W_T^T x + b_T)$ helps
- the from using Highway networks grows with depth

ResNet Architecture



Building Block:

```
Standard Block:              Residual Block:

x → Conv → ReLU →          x → Conv → BN → ReLU →
    Conv → y                     Conv → BN → (+) → ReLU → y
                                              ↑
                                              x (identity)
```

resnet-architecture.html

Full Architecture (ResNet-18):

Stage	Output Size	Layers
Conv1	112×112×64	7×7, 64, stride 2
Pool	56×56×64	3×3 max pool, stride 2
Conv2	56×56×64	[1×1, 64; 3×3, 64; 1×1, 256] × 3
Conv3	28×28×128	[1×1, 128; 3×3, 128; 1×1, 512] × 4
Conv4	14×14×256	[1×1, 256; 3×3, 256; 1×1, 1024] × 6
Conv5	7×7×512	[1×1, 512; 3×3, 512; 1×1, 2048] × 3
Pool	1×1×512	Global average pool
FC	1024	Fully connected + softmax

ResNet-50

- more layers
- additionally **bottleneck** layers
  - 1×1 convolutions to reduce number of channels
  - 3×3 convolution to extract features from lower dimensional representation
  - 1×1 convolution to increase the number of channels to the original size
- reduces the computational cost by operating over a lower dimensionality space

---

Why Residual Learning Works - Theory

1. Gradient Flow Analysis

Forward pass:

$$\begin{aligned} x_{l+1} &= \mathcal{F}(x_l, W_l) + x_l \\ x_{l+2} &= \mathcal{F}(x_{l+1}) + x_{l+1} \\ &= \mathcal{F}(x_{l+1}) + \mathcal{F}(x_l) + x_l \end{aligned}$$

Backward pass (chain rule):

$$\frac{\partial \mathcal{L}}{\partial x_l} = \frac{\partial \mathcal{L}}{\partial x_{l+1}} \cdot \frac{\partial x_{l+1}}{\partial x_l} = \frac{\partial \mathcal{L}}{\partial x_{l+1}} \cdot \left(1 + \frac{\partial \mathcal{F}}{\partial x_l}\right)$$

Key: The "1" ensures gradient doesn't vanish!

2. Optimisation Landscape

- **Without residuals**: Loss landscape highly non-convex, many poor local minima

- **With residuals**: Smoother landscape, easier optimisation
- **Effective depth**: Network can adaptively choose its depth

3. Ensemble Interpretation

ResNet = Ensemble of shallow networks

- Each path through network is a different "model"
- Exponentially many paths through residual connections
- Explains robustness and performance

What are the difficulties once again

```
20-layer network: Train error = 8%, Test error = 10%
56-layer network: Train error = 15%, Test error = 18%  ← Worse on BOTH!
```

If It's Not Overfitting, What Is It?

cnn-limitations.html

1. Optimisation Difficulty

The Core Problem: Deeper networks are harder to optimise, even though they have more representational power.

```
# Theoretically, 56-layer SHOULD be able to do at least as well as 20-
layer
# It could just learn identity mappings for layers 21-56:
perfect_56_layer = twenty_layer_solution + identity_layers[21:56]

# But in practice, SGD can't find this solution!
```

Why SGD Fails:

- Exponentially many parameters to coordinate
- Information/gradients must flow through many layers
- Each layer compounds the optimisation difficulty

2. Vanishing/Exploding Gradients

Even with careful initialisation and BatchNorm:

```
# Gradient flow through 56 layers
gradient = dL/dy₅₆ × dy₅₆/dy₅₅ × dy₅₅/dy₅₄ × ... × dy₂/dy₁

# If each |dy_i/dy_{i-1}| < 1: gradient → 0 (vanishing)
# If each |dy_i/dy_{i-1}| > 1: gradient → ∞ (exploding)
```

The Multiplication Effect:

- $0.9^{56} \approx 0.003$ (vanishes)
- $1.1^{56} \approx 304$ (explodes)

3. The Identity Mapping Problem

The Fundamental Issue: It's surprisingly hard for deep networks to learn identity mappings!

```
# What we want for extra layers:
H(x) = x  # Identity function

# What networks must learn:
H(x) = W₂ · ReLU(W₁ · x + b₁) + b₂  # Complex!

# To achieve H(x) = x requires:
# - W₁ and W₂ perfectly calibrated
# - Biases exactly right
# - All coordinated across layers
```

4. Shattered Gradients

In very deep networks, gradients become increasingly chaotic:

```
# Gradient correlation between nearby inputs decreases exponentially with
depth
correlation(∇L(x), ∇L(x + ε)) ≈ exp(-depth)

# Result: Similar inputs get wildly different gradient signals
# Makes learning consistent features nearly impossible
```

5. The Optimisation Landscape Perspective

As networks get deeper:

Depth	Optimization Landscape
Shallow (≤10)	Smooth, few critical points
Medium (10-20)	More critical points, but navigable

Depth	Optimization Landscape
Deep (50+)	Highly non-convex, many bad saddle points
Very Deep (100+)	Chaotic, fractal-like, gradient noise dominates

Why ResNet's Solution Works

ResNet's skip connections solve ALL these problems:

```
# Instead of learning H(x) directly:
y = F(x) + x  # F(x) is residual

# Now the gradient flow:
dy/dx = dF/dx + 1  # The "+1" ensures gradients flow!

# Learning identity is trivial:
# Just set F(x) = 0 (all weights → 0)
```

The Key Insight

> **It's not about network capacity (what it CAN represent)**
> **It's about optimization difficulty (what SGD can FIND)**

The degradation problem shows that:

1. ❌ **More layers ≠ Better** (without proper architecture)
2. ✅ **Architecture matters more than depth alone**
3. ✅ **Good gradient flow is essential**

Practical Implications

This is why modern architectures ALL use skip connections:

- ResNet (additive skip)
- DenseNet (concatenative skip)
- Transformer (residual around attention & FFN)
- U-Net (skip across scales)

> 💡 **The degradation problem revealed that the challenge isn't giving networks more capacity, but making that capacity accessible to optimisation.**

ResNet variants and impact

variants:

ResNet v2 (Pre-activation):

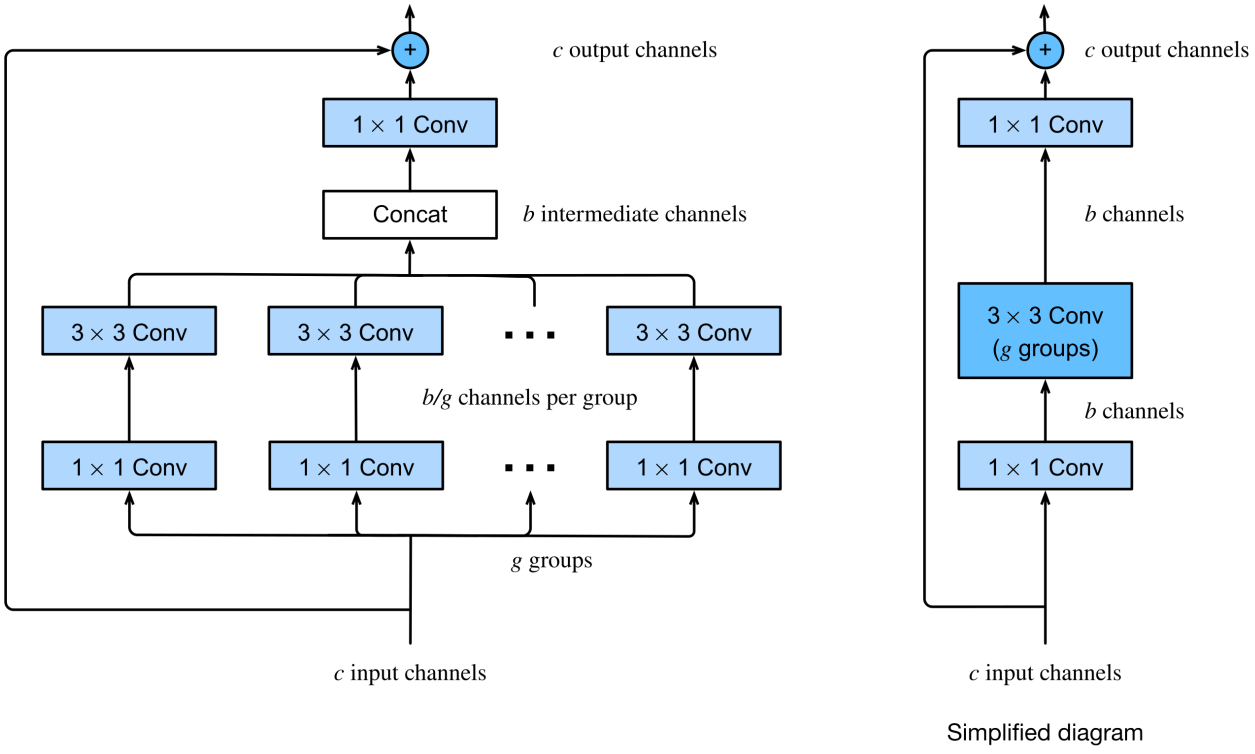
```
x → BN → ReLU → Conv → BN → ReLU → Conv → (+) → y
                                           ↑
                                           x
```

Wide ResNet:

- Increase width instead of depth
- smart width vs depth manipulation gives better accuracy together with computational gains

ResNeXt:

- Grouped convolutions for efficiency



Simplified diagram

DenseNet: Connect to ALL previous layers

Impact on Deep Learning:

✅ Enabled training of 1000+ layer networks
✅ Won ImageNet 2015 (3.57% error)
✅ Influenced all subsequent architectures
✅ Skip connections now universal (Transformers, U-Net, etc.)

> 💡 **Key Takeaway**: Residual connections solve optimisation, not just vanishing gradients

---

Evaluation metrics and benchmarks

how do we know if it works?

---

evaluation-metrics.html

evaluation landscape

Core Questions:

1. What are we measuring? (accuracy, robustness, fairness?)
2. On what data? (train, validation, test, production?)
3. Compared to what? (baselines, humans, SOTA?)
4. How reliably? (confidence intervals, multiple runs?)

Types of Evaluation:

Type	Purpose	When to Use
Intrinsic	Model quality	Development
Extrinsic	Task performance	Deployment
Online	Real-world impact	Production
Human	Perceptual quality	Final validation

fundamental trade-offs:

- Accuracy vs Interpretability
- Performance vs Efficiency
- Average case vs Worst case
- In-distribution vs Out-of-distribution

---

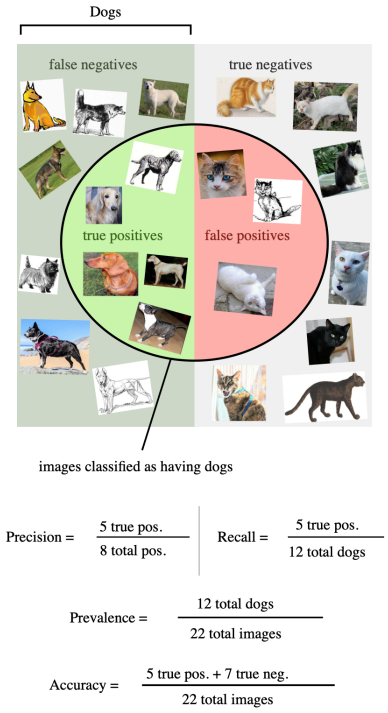
classification metrics

Basic Metrics:

Accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision & Recall:



[wikipedia]

precision: ratio of relevant retrieved instances to all retrieved instances

$$\text{Precision} = \frac{TP}{TP + FP}$$

recall: ratio of relevant retrieved instances to all relevant instances

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1 Score (Harmonic mean):

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

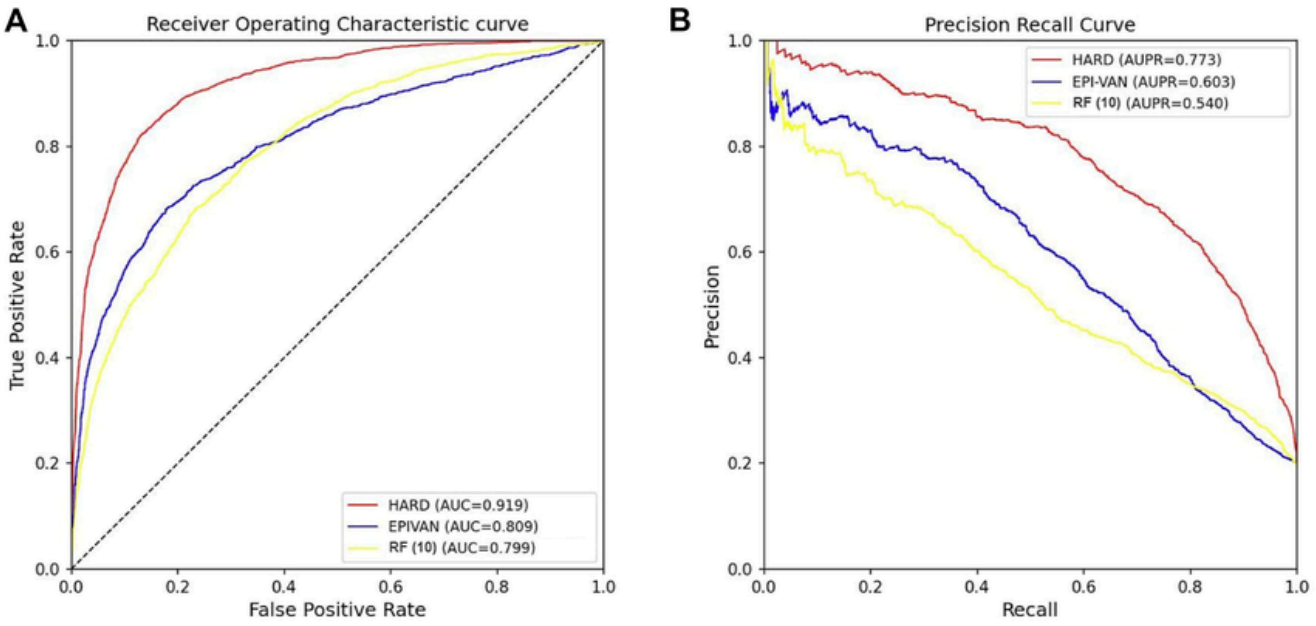
Advanced Metrics:

AUROC (Area Under ROC): Threshold-independent performance

- true positive rate TPR TP/(TP+FN) (vertical) vs false positive rate FPR FP/(FP+TN) at different thresholds
- ideal is top left corner
- the diagonal gives values for a random classifier
- AUROC is the area below the curve a model for all thresholds: the higher, the better

AUPRC (Area Under Precision-Recall): Better for imbalanced data

- precision TP/(TP+FP) (vertical axis)) vs recall TP/(TP+FN) (horizontal axis) at different threshold points
- the ideal is the right-top corner
- use in case of imbalanced classes, in particular when the positive class examples are rare
- AUPRC summarises model's prediction for all thresholds with perfect equal to 1



[Wikipedia]

Matthews Correlation Coefficient:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Multi-class Extensions:

- **Macro-average**: Average per-class metrics is an unweighted score, e.g. taking first the score for each class and then averaging them, e.g.

$$score_{macro} = \frac{\sum_1^n score_i}{n}$$

- **Micro-average**: Pool all predictions takes into account the score for each class, considering each sample's TP, FP, FN

$$avg_{micro} = \frac{\sum_1^n TP_i}{\sum_1^n (TP_i + FP_i)}$$

- **Weighted-average**: compute metric for each class separately, and then compute the weighted average by class frequency to be used in case of class imbalance, e.g.

$$score_{weigh} = prob_{cls_1} * score_{cls_1} + \cdots + prob_{cls_n} * score_{cls_n}$$

---

task-specific metrics

Computer Vision:

Task	Metrics	Description
Detection	mAP@IoU	Mean Average Precision at IoU threshold
Segmentation	mIoU	Mean Intersection over Union
Generation	FID, IS	Fréchet Inception Distance, Inception Score

Natural Language:

Perplexity (Language Modeling):

$$PPL = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log p(x_i|x_{<i})\right)$$

BLEU (Translation):

$$BLEU = BP \cdot \exp\left(\sum_{1=1}^{N} w_i \log p_i\right)$$

where p_n = n-gram precision, BP = brevity penalty

ROUGE (Summarisation): Recall-based n-gram overlap

Time Series:

- **MAE/RMSE**: Absolute/squared errors
- **MAPE**: Percentage errors
- **DTW**: Dynamic time warping distance

---

data splitting strategies

Standard Splits:

```
# Traditional 60-20-20 split
├── Train (60%)      # Model training
├── Validation (20%) # Hyperparameter tuning
└── Test (20%)       # Final evaluation
```

Cross-Validation (for small datasets):

K-Fold CV:

```
Fold 1: [Test] [Train] [Train] [Train] [Train]
Fold 2: [Train] [Test] [Train] [Train] [Train]
...
Fold 5: [Train] [Train] [Train] [Train] [Test]
```

Stratified K-Fold: Preserves class distributions

Time Series CV: Respects temporal order

```
Fold 1: [Train: 1-100] [Test: 101-120]
Fold 2: [Train: 1-120] [Test: 121-140]
Fold 3: [Train: 1-140] [Test: 141-160]
```

Advanced Strategies:

- **Leave-One-Out**: N folds for N samples
- **Group K-Fold**: No group appears in train and test
- **Nested CV**: Hyperparameter selection within CV

---

benchmarks and their limitations

Popular Benchmarks:

Domain	Benchmark	What It Measures	Limitations
Vision	ImageNet	Object recognition	Western-centric labels
NLP	GLUE/SuperGLUE	Language understanding	English-only
RL	Atari	Game playing	Not real-world
Multimodal	COCO	Image-text alignment	Limited diversity

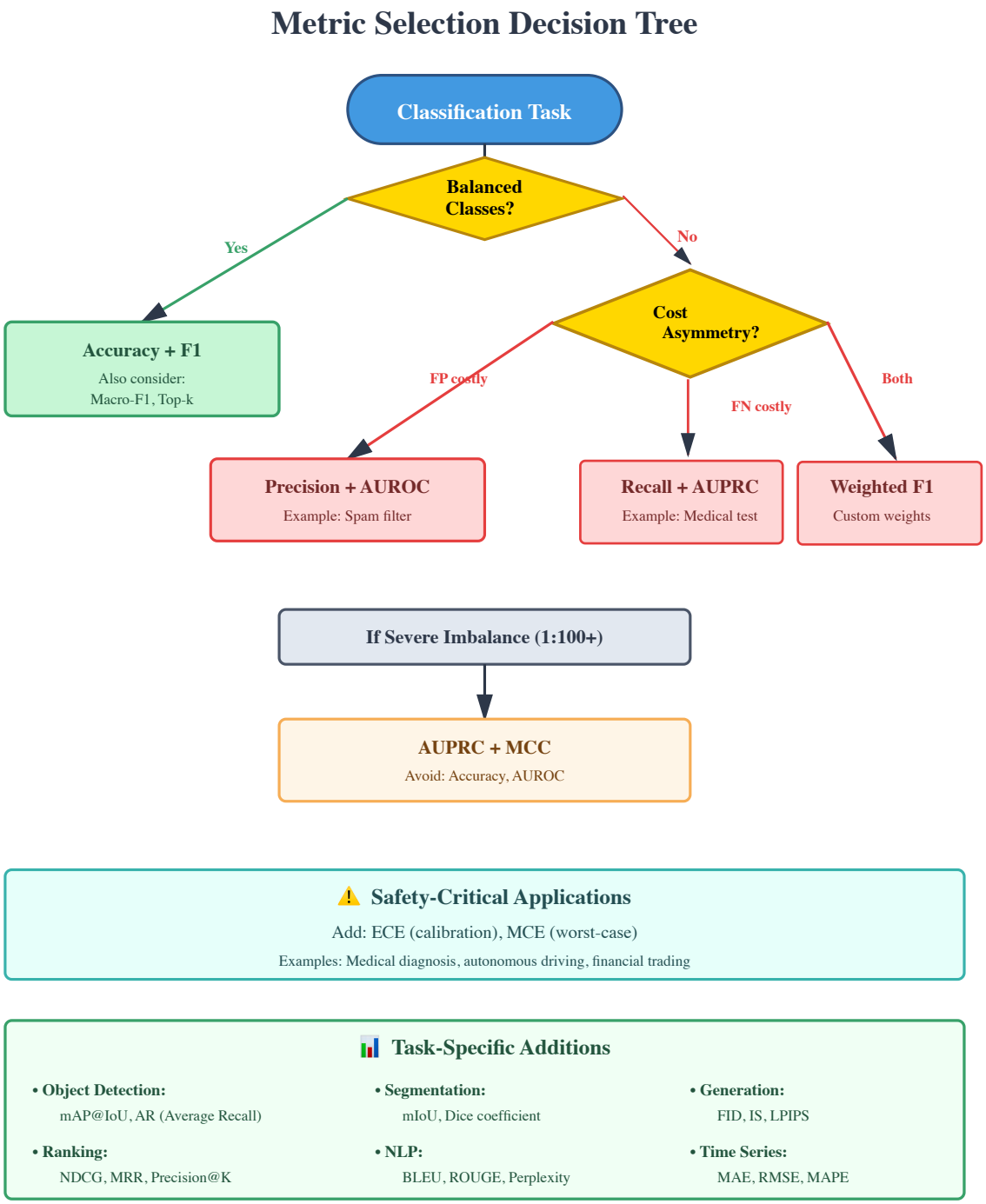
Problems with Benchmarks:

1. **Overfitting to Test Sets**
   - Models optimized for benchmark, not task
   - "Clever Hans" effects
2. **Distribution Shift**
   - Train/test from same distribution
   - Real world is different
3. **Goodhart's Law**
   - "When a measure becomes a target, it ceases to be a good measure"
   - if you focus on obtaining a specific value, this measure ceases to be a good measure
   - e.g. building a MNIST model that achieves a best generalisation on the test set
   - Kaggle
     - given a problem
     - people try to solve it and are given results on some test set
     - the final results are computed on some held out, but **different** set

Better Practices:

- **Living benchmarks**: Continuously updated
- **Adversarial evaluation**: Test robustness
- **Out-of-distribution testing**: Real generalisation

- **Human baselines**: Meaningful comparison



Statistical tests

how to compare multiple classifiers and decide which is best?

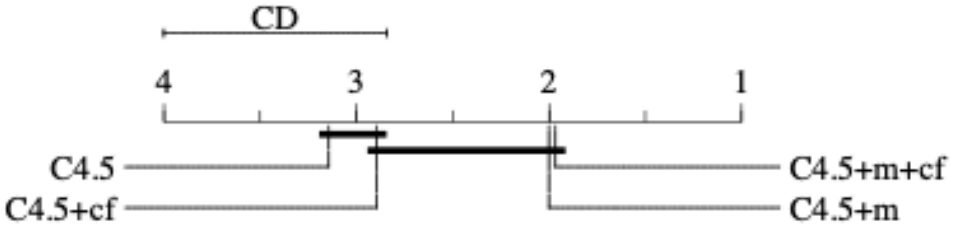
- simple comparison of error means and variances are not sufficient
- we need to compare on multiple datasets
- the comparisons require the sets to be sufficiently large and different
- the comparisons require appropriate statistical tests
- possible procedure for two classifiers over N datasets
    - run each classifier on all the datasets

- compute difference d_i between performance scores between i-th dataset
- rank the differences (averages are taken if d_i are very close e.g. take $r_i = 1.5$ if d_1 is very similar for the best score for both classifiers)
- take means
  - R^+ where the second algorithm outperformed the first one
  - R^- where the first outperformed the second
- compute $T = min(R^+, R^-)$
- compute statistics

$$z = \frac{T - \frac{1}{4}N(N+1)}{\sqrt{\frac{1}{24}N(N+1)(2N+1)}}$$

  which for large N should be distributed normally (usually 25)
- with $\alpha = 0.05$ the null hypothesis (that classifiers are equal) can be rejected if z is smaller than -1.96

- there are other tests for smaller number of datasets available (see Statistical comparisons of classifiers over multiple data sets, Demsar, JMLR 2006)
- software is available, also to draw nice figures like one below (the journals like them!), that draw the confidence difference intervals CD (here the statistical important differences in ranks) and the comparisons of classifiers



Model Calibration: Beyond Accuracy

The Confidence Problem

A model can be accurate but poorly calibrated:

```
Prediction: "Cat" with 95% confidence
Reality: Correct only 60% of the time when this confident
```

The problem

- the model is overconfident when making predictions,
- at the same time its accuracy may be low even with that high confidence!
- we need to calibrate the model - the confidence should be related to the correctness

What is Calibration?

> A model is **well-calibrated** if, when it predicts something with X% confidence, then it is correct X% of the time.

Example:

- 100 predictions at 80% confidence → Should be right ~80 times
- If right only 60 times → **overconfident** (poor calibration)
- If right 95 times → **underconfident** (also poor calibration)

Expected Calibration Error (ECE)

$$ECE = \sum_{m=1}^{M} \frac{|B_m|}{N} |acc(B_m) - conf(B_m)|$$

where:

- M = number of bins (typically 10-20)
- B_m = set of samples with confidence in bin m
- $acc(B_m)$ = accuracy of predictions in bin m
- $conf(B_m)$ = average confidence in bin m
- N = total number of samples
  measures the **relation of confidence to correctness** in bins
- the lower the ECE, the better the calibration
- **Pros:**
  - Simple, interpretable
  - Standard in literature
- **Cons:**
  - Bin size matters
  - Can hide worst-case errors

Visual Intuition

```
Perfect Calibration:
Confidence |
     1.0   |    /     (diagonal line)
```

```
          |  /
     0.5  |/
          |
     0.0  └──────────
          0.0  0.5  1.0
              Accuracy

Poor Calibration:
Confidence |
     1.0  |      /--
          |   /    (curve far from diagonal)
     0.5  |/
          |
     0.0  └──────────
          0.0  0.5  1.0
              Accuracy
```

Lower ECE = Better calibration (typically want ECE < 0.05)

Why Calibration Matters

Real-World Scenarios

1. **Medical Diagnosis**

```
# BAD: High accuracy but poor calibration
Model: "95% sure it's cancer" → Actually cancer only 60% of time
# perhaps the overcautiousness maybe welcome in some of such
situations
# GOOD: Calibrated
Model: "60% sure it's cancer" → Actually cancer ~60% of time
```

2. **Autonomous Driving**

```
# Overconfident model
Model: "99% sure road is clear" → Crashes 5% of time (disaster!)

# Calibrated model
Model: "95% sure road is clear" → Can use this to decide whether to
slow down
```

3. **Financial Trading**
   - Confidence scores determine position sizes
   - Poor calibration → Wrong risk assessment → Losses

Deep Networks are Often Overconfident

Modern neural networks tend to be:

- ✅ Accurate (high top-1 accuracy)
- ❌ Overconfident (poor calibration)

Why?

- Deep networks can memorize
- No explicit calibration objective
- Softmax amplifies differences

Maximum Calibration Error (MCE)

$$MCE = \max_{m \in \{1,\dots,M\}} |acc(B_m) - conf(B_m)|$$

Pros:

- checks worst-case miscalibration
- perfect for safety-critical applications

Cons:

- sensitive to bin with few samples
- too conservative

Adaptive ECE (AdaECE)

- Adaptive binning based on data distribution
- Better for non-uniform confidence distributions

Calibration Techniques

1. Temperature Scaling (Most Popular)

```
# After training, find optimal temperature T
logits = model(x)
calibrated_probs = softmax(logits / T)
```

- T > 1: Makes model less confident
- T < 1: Makes model more confident
- T = 1: No change

How it works:

- Single scalar parameter T
- Optimised on validation set
- Preserves accuracy, improves calibration

2. Platt Scaling

- Fit logistic regression on top of model outputs
- More flexible than temperature scaling

3. Isotonic Regression

- Non-parametric calibration
- Fits monotonic function to calibration curve

4. Ensemble Methods

- use average predictions from multiple models
- Natural calibration improvement
- usually more computationally expensive

When to Measure Calibration

✅ **Safety-critical applications**

- Medical diagnosis
- Autonomous vehicles
- Financial decisions

✅ **Confidence scores are used in computing (prediction) pipelines (downstream)**

- Active learning (sample selection)
- Human-in-the-loop systems
- Multi-stage pipelines

✅ **Deploying to production**

- Users see confidence scores
- Decisions based on uncertainty

Calibration vs Accuracy Trade-off

- High Accuracy + Poor Calibration:
  - Model is often right but doesn't know when
  - Dangerous for decision-making
- Lower Accuracy + Good Calibration:
  - Model knows its limitations
  - Can abstain on uncertain cases
  - Often better for deployment

Metric Selection Framework

Step 1: Understand Your Problem

```
Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBoundingClientRect')
```

Step 2: Consider Data Characteristics

Data Characteristic	Recommended Metrics	Avoid
Balanced classes	Accuracy, Macro-F1	Micro-F1
Imbalanced (1:10)	AUPRC, Weighted-F1	Accuracy
Severe imbalance (1:100)	AUPRC, MCC	AUROC
Multi-class balanced	Macro-F1, Top-k	Micro-F1
Multi-class imbalanced	Weighted-F1, MCC	Accuracy

Step 3: Add Task-Specific Metrics

```
# Safety-critical
metrics = ['Accuracy', 'Recall', 'ECE', 'MCE']  # Must catch all positives

# Recommendation system
metrics = ['Precision@K', 'NDCG', 'Diversity']  # Top-k quality matters

# Anomaly detection
metrics = ['AUPRC', 'F1@threshold', 'Coverage']  # Positive class is rare


# Production deployment
```

```
metrics = ['Accuracy', 'ECE', 'Latency', 'Throughput']  # Performance + calibration
```

Regularisation methods

Preventing Overfitting at Scale

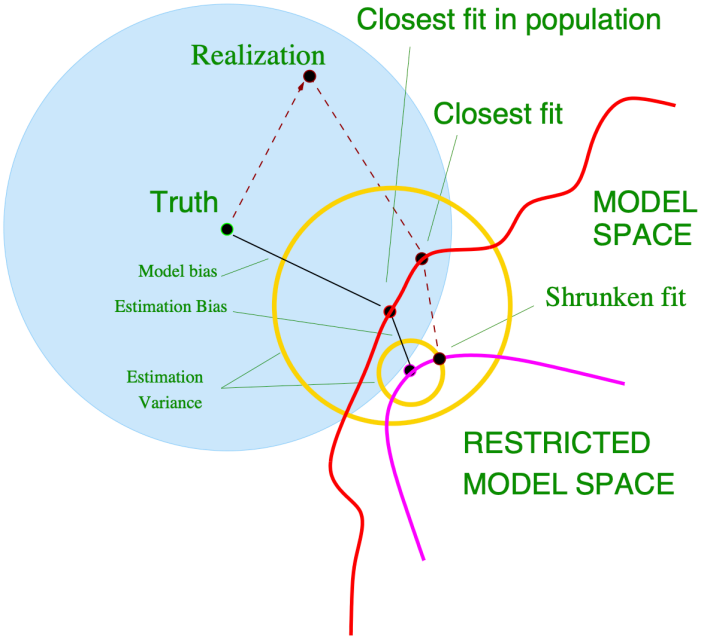
---

regularisation-methods.html

regularisation spectrum

What is Regularisation?

> Any modification to the learning algorithm intended to reduce generalisation error but not training error



The Overfitting Equation:

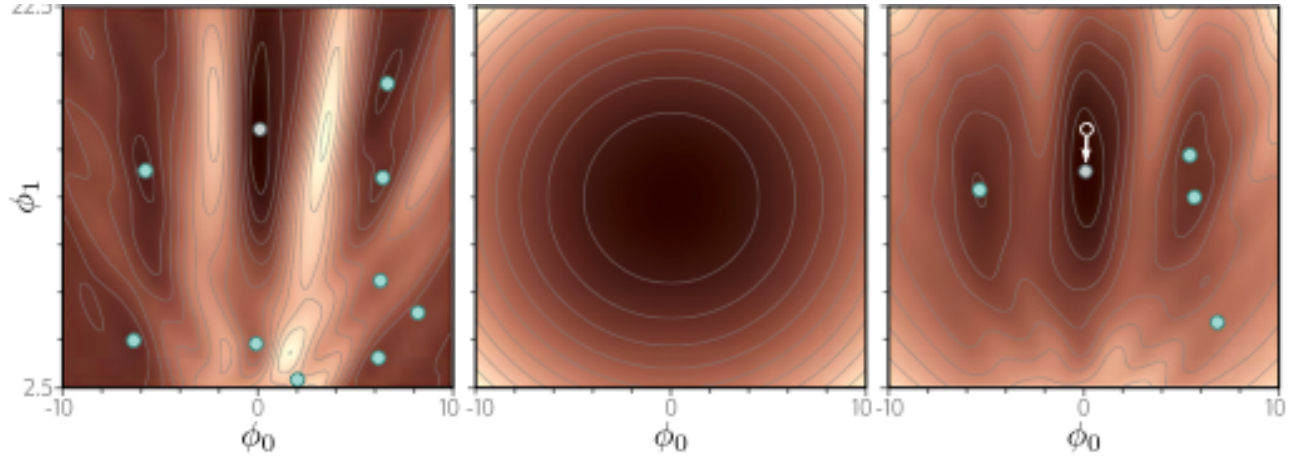
$$\mathcal{L}_{total} = \underbrace{\mathcal{L}_{data}}_{\text{Fit the data}} + \lambda \underbrace{\Omega(\theta)}_{\text{Keep it simple}}$$

Types of Regularization:

Type	Method	What It Controls
Explicit	L1/L2 penalty	Weight magnitude
Implicit	Early stopping	Training time
Structural	Dropout	Network capacity
Data	Augmentation	Effective dataset size
Architectural	Conv/Pooling	Parameter sharing
Algorithmic	Batch norm	Internal distributions

---

an explicit regularisation



- the Gabor function

$$f(x, w) = \sin(w_0 + 0.06 \cdot w_1 x) \cdot \exp(-\frac{(w_0 + 0.06 \cdot w_1 x)^2}{32.0})$$

- several local minima
- a global minimum (left drawing) in center/upper dot
- the center drawing describes some regularisation term that prefers some solutions in the center
- the right map shows that solutions move to the center

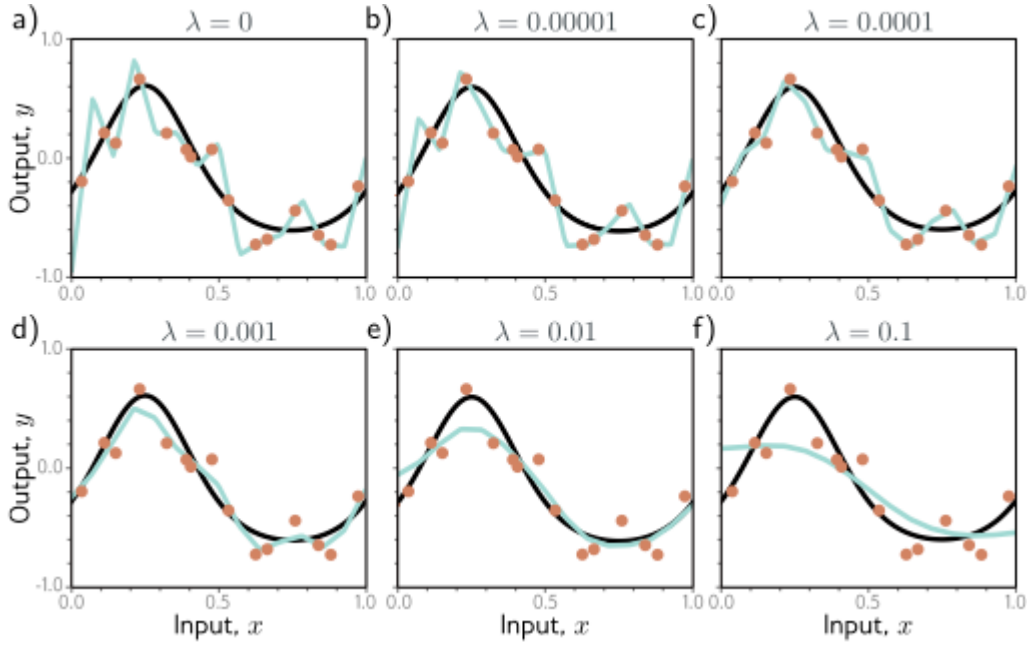
Weight Decay - L2 vs L1

L2 Regularization (Ridge/Weight Decay):

$$\mathcal{L} = \mathcal{L}_{data} + \frac{\lambda}{2} \sum_i w_i^2$$

Gradient update:

$$w_i \leftarrow w_i - \eta \left(\frac{\partial \mathcal{L}_{data}}{\partial w_i} + \lambda w_i \right)$$



Effect: Weights decay by factor $(1 - \eta\lambda)$ each step

L1 Regularisation (Lasso):

$$\mathcal{L} = \mathcal{L}_{data} + \lambda \sum_i |w_i|$$

Gradient update:

$$w_i \leftarrow w_i - \eta \left(\frac{\partial \mathcal{L}_{data}}{\partial w_i} + \lambda \cdot \text{sign}(w_i) \right)$$

Effect: Constant push toward zero → sparsity

Comparison:

Property	L2 (Ridge)	L1 (Lasso)
Solution	Dense	Sparse
Gradient at 0	Proportional to weight	Constant magnitude
Optimization	Smooth	Non-smooth at 0
Feature selection	No	Yes
Outlier sensitivity	Lower	Higher

Weight Decay - Geometric Interpretation

Constraint Region Visualisation:

```
L2 constraint: w₁² + w₂² ≤ c  (circle/sphere)
L1 constraint: |w₁| + |w₂| ≤ c (diamond/octahedron)
```

Why L1 Gives Sparsity:

The optimum occurs where loss contours touch constraint region:

- **L2**: Can touch anywhere on smooth sphere
- **L1**: Likely to touch at corners (axes) → some weights = 0

Bayesian Interpretation:

L2 = Gaussian Prior:

$$p(w) \propto \exp\left(-\frac{w^2}{2\sigma^2}\right)$$

L1 = Laplace Prior:

$$p(w) \propto \exp\left(-\frac{|w|}{\sigma}\right)$$

Elastic Net (L1 + L2):

$$\Omega(w) = \alpha||w||_1 + \frac{1-\alpha}{2}||w||_2^2$$

Best of both worlds: Sparsity + grouped selection

Early Stopping as Regularisation

The Algorithm:

```python
best_val_loss = inf
patience_counter = 0

for epoch in range(max_epochs):
    train_step()
    val_loss = evaluate()
```

```
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        save_model()
        patience_counter = 0
    else:
        patience_counter += 1

    if patience_counter >= patience:
        break  # Early stop
```

Why It's Regularization:

Theoretical Connection to L2:

- For linear models with gradient descent
- Early stopping ≈ L2 with $\lambda = \frac{1}{\eta t}$
- Proof: Weights grow as $w(t) \approx w^*(1 - e^{-\eta t})$

Advantages:

✅ No hyperparameter in loss
✅ Computational savings
✅ Automatic selection of capacity

Best Practices:

- Use validation set (not test!)
- Save best model during training
- Consider learning rate scheduling

Dropout - Regularization by Randomness

The Dropout Algorithm:

Training:

```
mask = Bernoulli(p)  # Random binary mask
h = (x * mask) / p    # Scale to maintain expectation
```

Test time:

```
h = x  # Use all neurons, already scaled
```

Why Dropout Works:

1. **Ensemble Interpretation**:
   - Training samples from 2^n different sub-networks
   - Test time approximates ensemble average
2. **Co-adaptation Prevention**:
   - Neurons can't rely on specific other neurons
   - Forces redundant representations
3. **Noise Injection**:
   - Acts as adaptive noise on activations
   - Stronger regularisation for confident predictions

Theoretical View:

Dropout approximately minimises:

$$\mathcal{L}_{dropout} = \mathbb{E}_{mask}[\mathcal{L}(f_{mask}(x), y)] + \mathrm{KL}(q||p)$$

where KL term encourages robustness

Making deep networks trainable

---

normalisation-techniques.html

Batch Normalisation - the game changer

the problem: internal covariate shift

During training, each layer's input distribution changes:

- Layer L adapts to distribution from layer L-1
- But L-1's parameters change → distribution shifts
- Layers constantly "chase" moving targets

The BatchNorm Solution:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

where:

- μ_B, σ_B^2 = batch mean and variance
- γ, β = learned scale and shift
- ϵ = small constant for stability

Training vs Inference:

- **Training**: Use batch statistics
- **Inference**: Use running average from training

```
# Training
mean = x.mean(dim=0)
var = x.var(dim=0)
x_norm = (x - mean) / sqrt(var + eps)

# Inference
x_norm = (x - running_mean) / sqrt(running_var + eps)
```

Why BatchNorm Works - Multiple Theories

Theory 1: Internal Covariate Shift Reduction

- Original hypothesis (Ioffe & Szegedy, 2015)
- Recent work questions this explanation
- ICS still occurs with BN!

Theory 2: Optimization Landscape Smoothing

- BN makes loss landscape smoother
- Enables larger learning rates
- Proof: Lipschitz constant of loss is reduced

Theory 3: Length-Direction Decoupling

- BN separates optimization of:
  - Weight magnitude (length)
  - Weight direction
- Makes optimization more stable

Theory 4: Implicit Regularization

- Each mini-batch sees different normalization
- Acts like noise injection
- Explains why larger batches hurt generalization

The Current Consensus:

> BatchNorm helps primarily through **optimisation benefits**, not covariate shift reduction

---

Normalisation Zoo

Layer Normalisation:

Normalise across features (for each sample):

$$\hat{x}_i = \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}}$$

Use case: Transformers, RNNs (batch-independent)

Group Normalisation:

Normalise within groups of channels:

$$\hat{x}_i = \frac{x_i - \mu_G}{\sqrt{\sigma_G^2 + \epsilon}}$$

Use case: Small batch sizes, detection models

Instance Normalization:

Normalize each channel of each sample:

$$\hat{x}_i = \frac{x_i - \mu_I}{\sqrt{\sigma_I^2 + \epsilon}}$$

Use case: Style transfer, image generation

Comparison:

Method	Normalize Over	Batch Required	Where Used
BatchNorm	Batch, Height, Width	Yes	CNNs

Method	Normalize Over	Batch Required	Where Used
LayerNorm	Channels, Height, Width	No	Transformers
GroupNorm	Group, Height, Width	No	Detection
InstanceNorm	Height, Width	No	StyleGAN

Advanced Regularization Techniques

Data Augmentation:

```
# Standard augmentations
- Random crop/flip/rotation
- Color jittering
- Mixup: x = λx₁ + (1-λ)x₂
- CutMix: Patch-based mixing
- AutoAugment: Learned policies
```

Stochastic Depth:

Randomly drop entire ResNet blocks:

$$H_l = \begin{cases} x_l + F_l(x_l) & \text{with probability } p_l \\ x_l & \text{with probability } 1 - p_l \end{cases}$$

Label Smoothing:

Instead of hard targets [0,1,0,0]:

$$y_{smooth} = (1 - \epsilon)\, y_{true} + \epsilon / K$$

Prevents overconfidence, improves calibration

Gradient Penalties:

- **Gradient clipping**: Prevent exploding gradients
- **Spectral normalization**: Constrain Lipschitz constant
- **Gradient penalty** (WGAN-GP): Enforce 1-Lipschitz

Putting It All Together

Modern Training Recipe:

```
model = ResNet50()

# Regularization stack
model = apply_weight_decay(model, 1e-4)      # L2
model = add_dropout(model, p=0.1)            # Stochastic
model = add_batchnorm(model)                 # Normalization

# Data pipeline
train_data = apply_augmentation(data)        # Data reg
train_data = apply_mixup(train_data, α=0.2)  # Mixup

# Training
optimizer = AdamW(model.parameters())        # Decoupled WD
scheduler = CosineAnnealingLR(optimizer)     # LR schedule

for epoch in epochs:
    train(model, train_data)
    val_loss = evaluate(model, val_data)

    if early_stopping(val_loss):             # Early stop
        break
```

Key Principles:

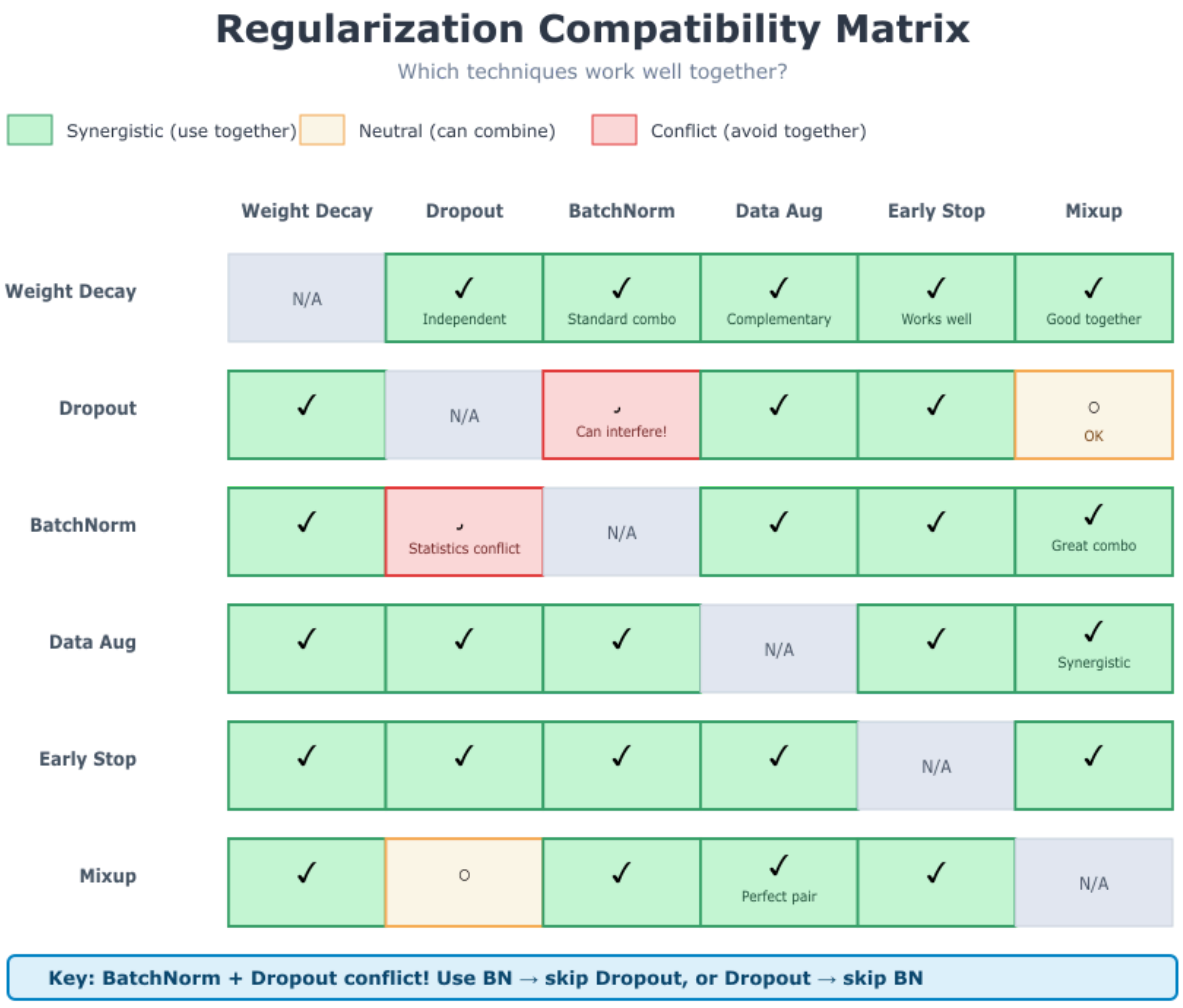
1. **Start simple**: Add regularization incrementally
2. **Validate everything**: Each technique should improve validation
3. **Watch for interactions**: BN + Dropout can conflict
4. **Problem-specific**: Vision ≠ NLP ≠ RL

Regularisation Interactions

The Compatibility Matrix

Technique	BatchNorm	Dropout	Weight Decay	Data Aug	Early Stop
BatchNorm	-	⚠️	✅	✅	✅
Dropout	⚠️	-	✅	✅	✅
Weight Decay	✅	✅	-	✅	✅
Data Augmentation	✅	✅	✅	-	✅
Early Stopping	✅	✅	✅	✅	-

✅ = Synergistic
⚠️ = Can interfere



Critical Interaction: BatchNorm + Dropout

The Problem:

```
# This order can hurt performance!
x = conv(x)
x = batchnorm(x)
x = relu(x)
x = dropout(x)  # ← Interferes with BN statistics!
```

Why it's problematic:

1. BatchNorm normalises using batch statistics
2. Dropout randomly zeros some activations randomly
3. This changes the statistics BatchNorm sees
4. Train/test mismatch worsens

Solutions:

```
# Option 1: Drop Dropout (if using BatchNorm)
x = conv(x)
x = batchnorm(x)
x = relu(x)
# No dropout needed - BN regularizes enough

# Option 2: Dropout before BatchNorm
x = conv(x)
x = dropout(x)
x = batchnorm(x)  # BN sees consistent dropout pattern throughout the
batch
x = relu(x)

# Option 3: Use only Dropout (without BatchNorm)
x = conv(x)
x = relu(x)
x = dropout(x)
```

Architecture-Specific Recommendations

ResNet / CNNs

```
regularization = {
    'weight_decay': 1e-4,       # Standard L2
    'batchnorm': True,          # At each conv block
    'dropout': 0.0,             # Usually no need with BN
    'data_aug': 'standard',     # Random crop, flip
    'early_stop': True,         # On validation plateau
}
```

Vision Transformer

```
regularization = {
    'weight_decay': 0.05,       # Higher than CNNs!
    'dropout': 0.1,             # Path dropout
    'drop_path': 0.1,           # Stochastic depth
    'data_aug': 'strong',       # RandAugment, Mixup
    'layer_decay': 0.65,        # Layer-wise LR decay
    'early_stop': False,        # Train to completion
}
```

Small Dataset (< 10K images)

```
regularization = {
    'weight_decay': 1e-3,       # Stronger
    'dropout': 0.5,             # Higher
    'data_aug': 'aggressive',   # All augmentations
    'early_stop': True,         # Critical!
    'pretrained': True,         # Use transfer learning
}
```

Practical Hyperparameter Ranges

Quick Reference Table

Technique	Small Data	Medium Data	Large Data	Notes
Weight Decay	1e-3 to 1e-2	1e-4 to 1e-3	1e-5 to 1e-4	Decrease with more data
Dropout (FC)	0.5 - 0.7	0.3 - 0.5	0.1 - 0.3	Higher for small data
Dropout (Conv)	0.1 - 0.3	0.05 - 0.1	0.0 - 0.05	CNNs need less
Label Smooth	0.0 - 0.05	0.05 - 0.1	0.1 - 0.2	More for large models
Mixup α	0.0 - 0.2	0.2 - 0.4	0.4 - 1.0	Vision transformers use higher

Starting Point Recipe

```
# For most CV tasks with moderate data (10K-100K images)
config = {
    # Optimizer
    'optimizer': 'AdamW',
    'lr': 1e-3,
    'weight_decay': 1e-4,

    # Regularization
    'dropout': 0.1,           # If using FC layers
    'label_smoothing': 0.1,
    'data_aug': {
        'random_crop': 0.9,
        'random_flip': 0.5,
        'color_jitter': 0.3,
    },

    # Training
```

```
    'epochs': 100,
    'early_stop_patience': 10,
    'lr_schedule': 'cosine',
    'warmup_epochs': 5,
}
```

Then tune based on validation performance!

Red Flags: When Regularisation Fails

Symptom 1: Training Loss Not Decreasing

```
Epoch 1: train_loss=2.3, val_loss=2.3
Epoch 5: train_loss=2.1, val_loss=2.2
Epoch 10: train_loss=2.0, val_loss=2.1
...
Epoch 50: train_loss=1.9, val_loss=2.0  ← Still high!
```

Diagnosis: Over-regularization
Fix: Reduce weight decay, dropout, or data augmentation strength

---

Symptom 2: Large Train-Val Gap

```
Epoch 50: train_loss=0.1, val_loss=1.5  ← Huge gap!
          train_acc=98%,  val_acc=75%
```

Diagnosis: Under-regularization (overfitting)
Fix: Increase regularization or get more data

Symptom 3: Unstable Training

```
Epoch 1: val_loss=2.0
Epoch 2: val_loss=1.5
Epoch 3: val_loss=2.5  ← Big jump!
Epoch 4: val_loss=1.2
Epoch 5: val_loss=3.0  ← Crash!
```

Diagnosis: Conflicting regularization (e.g., BN + Dropout issue)
Fix: Remove one technique or adjust order

Summary: Regularisation practical rules (possible)

```
Start → BatchNorm (almost always)
   ↓
   Using CNNs?
   ├─ Yes → Skip Dropout, use data augmentation
   └─ No → Consider Dropout (0.1–0.3)
   ↓
   Small dataset (< 10K)?
   ├─ Yes → Strong augmentation + early stopping + pretrained
   └─ No → Moderate augmentation
   ↓
   Add weight decay (start with 1e-4)
   ↓
   Monitor train/val gap
   ├─ Gap large? → Increase regularization
   └─ Both losses high? → Decrease regularization
```

Key Principle

> **Start simple, add complexity gradually**
>
> 1. Baseline: Weight decay only
> 2. Add BatchNorm (if applicable)
> 3. Add data augmentation
> 4. Add other techniques if needed
> 5. Always validate on held-out data!

Summary

1. **Start with proven recipe:**
   - ResNet/EfficientNet for vision
   - Add standard augmentations
   - Use AdamW with cosine schedule
2. **Regularization priorities:**
   - Weight decay (always)
   - Data augmentation (always for vision)
   - Dropout (carefully, conflicts with BN)
   - Early stopping (free and effective)

3. **Debugging regularisation:**

```
# Should see:
train_loss > train_loss_without_reg  # Regularization hurts training
val_loss < val_loss_without_reg      # But helps validation
```

4. **Hyperparameter ranges:**
   - Weight decay: [1e-5, 1e-3]
   - Dropout: [0.1, 0.5]
   - Label smoothing: [0.05, 0.1]

> "All models are wrong, but some generalise better than others"