

Computer Vision

Naeemullah Khan

naeemullah.khan@kaust.edu.sa



جامعة الملك عبد الله
للعلوم والتقنية
King Abdullah University of
Science and Technology

KAUST Academy
King Abdullah University of Science and Technology

May 19, 2025

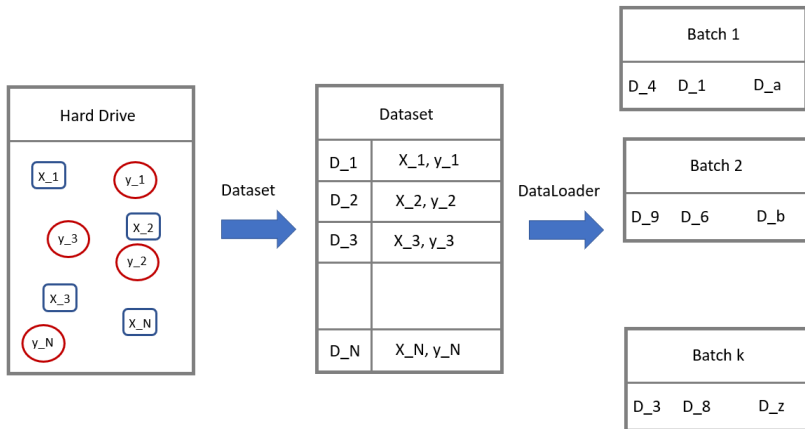
- ▶ Data Handling
- ▶ Data Augmentation
- ▶ Transfer Learning
- ▶ Ensembling
- ▶ Dropout
- ▶ Batch Normalization
- ▶ Full Training Workflow

- ▶ Implement efficient data handling techniques using PyTorch's Dataset and DataLoader classes
- ▶ Apply appropriate data augmentation strategies based on error analysis
- ▶ Utilize transfer learning effectively for different scenarios and dataset sizes
- ▶ Understand ensemble methods to improve model performance
- ▶ Apply regularization techniques like Dropout and Batch Normalization
- ▶ Design a complete deep learning training workflow from initial setup to inference

- ▶ Practical Implementation of Deep Learning algorithms is just as much an art as it is a science.
- ▶ The main takeaways if not to start from scratch rather to build on top of the previous knowledge.
- ▶ Today, we will look at some important tools used in the practical implementation of Deep Learning algorithms.

- ▶ As we have previously established that Deep Learning has been made possible by large amount of data and computational resource
- ▶ An important aspect to keep in mind is the data handling:
 - How do we handle large amounts of data?
 - How to we read different components of data (from possible different parts of our hard drive) and provide it to our training algorithms?
 - How do we feed this data to SGD algorithms in a streamlined manner?
- ▶ PyTorch provides Dataset and DataLoaders to handle data in an efficient manner.
- ▶ We will extend the Dataset and DataLoaders class to construct our own Dataloaders

- ▶ Datasets in PyTorch: The `torch.utils.data.Dataset` class is an abstract base class used to define and manage datasets for training and evaluation.
- ▶ Custom Dataset Creation: You can create a custom dataset by subclassing `Dataset` and implementing
 - `__len__()` (returns dataset size) and
 - `__getitem__()` (fetches a single data sample).
- ▶ DataLoaders for Batching: `torch.utils.data.DataLoader` is used to load data in mini-batches, shuffle data, and utilize multiprocessing for efficiency.
- ▶ Built-in Datasets: PyTorch provides datasets in `torchvision.datasets` and `torchtext.datasets`, making it easy to load common datasets like MNIST, CIFAR-10, and IMDB.

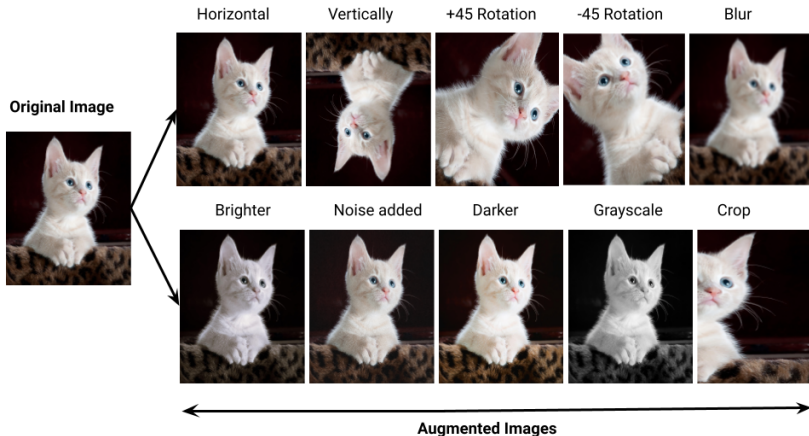


- ▶ Data is the fundamental building block of any machine learning algorithm
- ▶ In several applications we don't have access to unlimited data
- ▶ So we use Data Augmentation techniques to improve the performance of our models
- ▶ Note: It is better to spend time on data rather than fine-scale architecture search in deep learning

► Create virtual training samples

- Horizontal flip
- Random crop
- Color casting
- Geometric distortion
- Translation
- Rotation

Data Augmentation (cont.)



⁰<https://pranjal-ostwal.medium.com/>

[data-augmentation-for-computer-vision-b88b818b6010](https://pranjal-ostwal.medium.com/data-augmentation-for-computer-vision-b88b818b6010)

Data Augmentation (cont.)



⁰To simplify data augmentation, tools like

<https://github.com/albumentations-team/albumentations>

- ▶ But, there are many types of augmentations. How do we choose the right ones for our task?

- ▶ But, there are many types of augmentations. How do we choose the right ones for our task
- ▶ **Answer: Error Analysis** – Identify model weaknesses and apply augmentations that address those issues.

► Steps:

1. Train a baseline model.
2. Make predictions on validation data.
3. Inspect the worst predictions to identify model weaknesses.
4. Apply relevant augmentations to address these issues.

► Examples:

- **Failure with small objects** → Use *Scale Augmentation*.
- **Failure with different colors/environments** → Use *Color Augmentations*.
- **Failure with rotated images** → Use *Rotation Augmentations*.
- **Failure with blurry images** → Use *Noise Augmentations*.
- ... etc.

- ▶ **Note:** Data augmentation is applied only to the training data.
- ▶ Applying it to validation/test data **directly** would lead to **incorrect evaluation and misleading performance metrics**.

- ▶ However, there is a special inference technique called **Test-Time Augmentation (TTA)** where multiple augmented versions of the same image are passed through the model separately, and the predictions are averaged to improve accuracy.

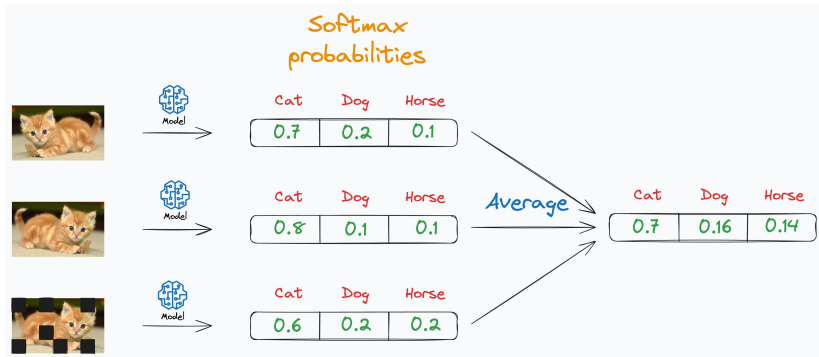


Figure 2: Test-time Augmentation

- ▶ TTA is an advanced technique.
- ▶ Applying it requires **custom scripts** to generate augmented versions of test images, pass them through the model, and aggregate predictions.

► What is Fine-Tuning?

- A strategy in **transfer learning** where a pre-trained model is adapted to a new task.
- Instead of training from scratch, we start from a model already trained on a related task.

► Why use Fine-Tuning?

- Saves time and computational resources.
- Improves performance, especially with limited data.

- ▶ New dataset is small + similar distribution to original dataset:
 - Freeze (or partially freeze) feature extraction layers and fine-tune the classifier.
- ▶ New dataset is small + different distribution to original dataset:
 - Use the pretrained network as a generic feature extractor and train a light classifier on top (e.g., SVM).
 - In modern practice, you might also freeze earlier layers and selectively fine-tune later layers.
- ▶ New dataset is large, regardless of the original data distribution:
 - Fine-tune the entire network (both features extractor and classifier).

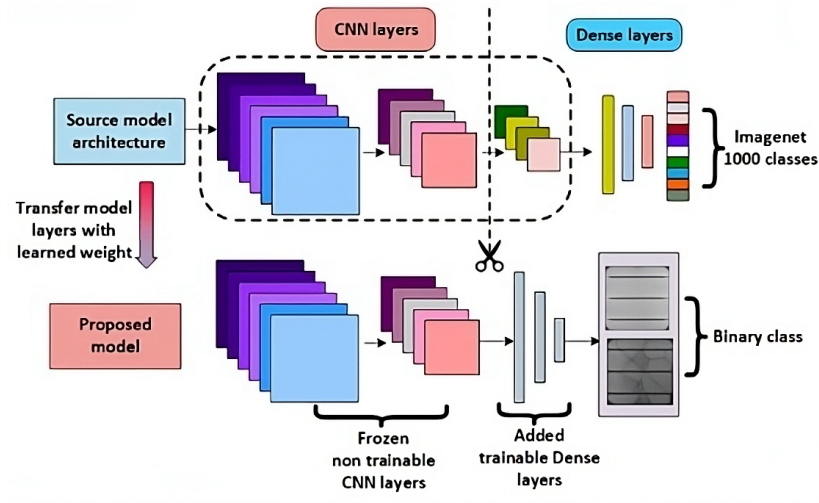


Figure 3: Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks

- ▶ No single model is perfect. Different models make different types of errors.
- ▶ Let's visualize the predictions of two different models:

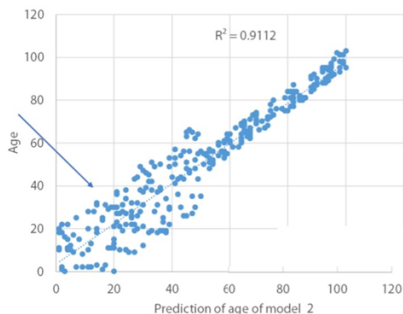
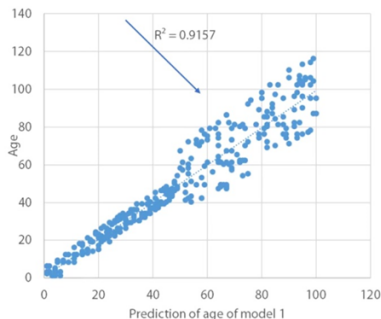


Figure 4: Predictions of Model 1 and Model 2

- Combining predictions of diverse models can reduce errors and improve accuracy. This technique is called **Ensembling**.

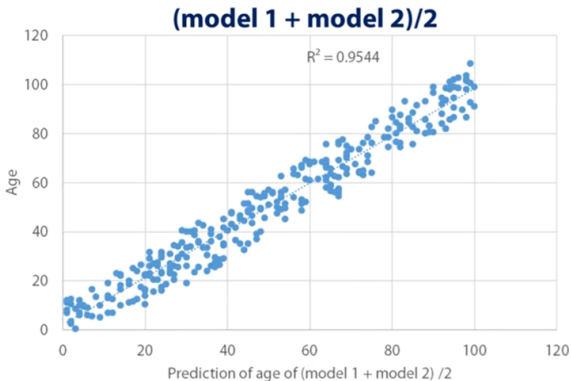


Figure 5: Averaged Predictions: Better Correlation

- Diverse models are key to effective ensembling. Here are two strategies:
 - **Bagging (Bootstrap Aggregating):** Train multiple models on different subsets of data (e.g., Random Forest model).

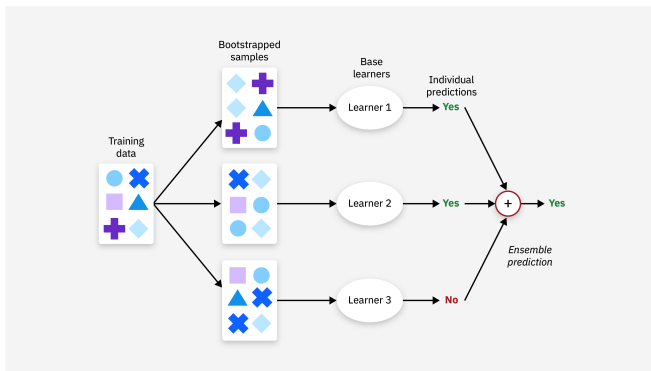


Figure 6: Bagging example

- **Boosting:** Train models sequentially, where each model focuses on the errors of the previous one (e.g., AdaBoost, Gradient Boosting).

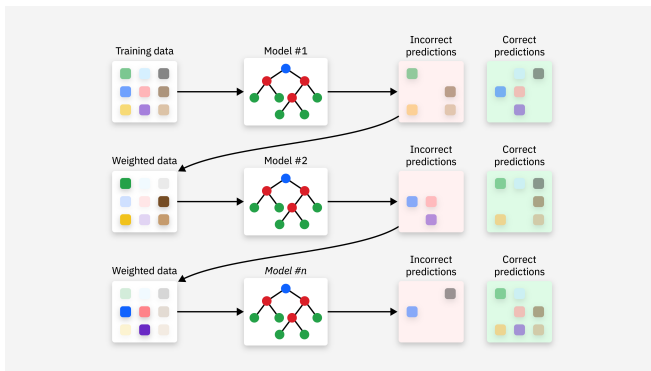


Figure 7: Boosting Example

- ▶ These strategies introduce diversity in models, but how can we combine their predictions?

- ▶ We can combine classifiers predictions using two ways:
 - **Hard Voting:**
 - ▶ Each model votes for a class.
 - ▶ The class with the majority of votes is selected.
 - **Soft Voting:**
 - ▶ Average the predicted probabilities from each model.
 - ▶ The class with the highest average probability is selected.

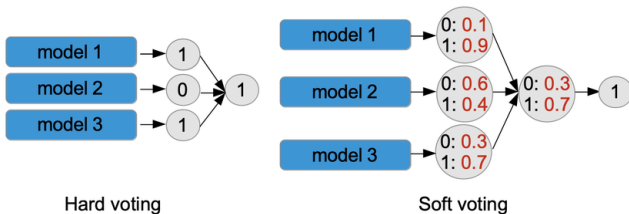


Figure 8: Illustration of Hard and Soft Voting for Classifiers

- ▶ We can combine classifiers predictions using two ways:
 - **Hard Voting:**
 - ▶ Each model votes for a class.
 - ▶ The class with the majority of votes is selected.
 - **Soft Voting:**
 - ▶ Average the predicted probabilities from each model.
 - ▶ The class with the highest average probability is selected.
- ▶ For regressors:
 - Take the average of predictions from all models.

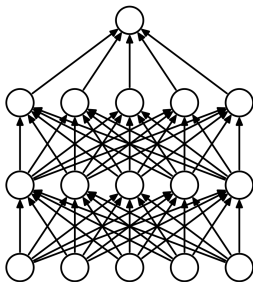
- ▶ Team-work is the best policy.
- ▶ We can train multiple networks for the same task then ensemble to get better results.

- ▶ Let's assume that we have a test dataset with N elements and an ensemble of M models.
- ▶ Also assume that the probability of error of the label for an image on a model in the ensemble is denoted by $p(e)$ and is Independent and Identically Distributed (i.i.d)
- ▶ For an example assume $M = 3$ and $e = 0.01$
- ▶ Then probability of error of label for the max voting ensemble will be

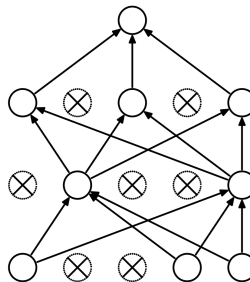
$$p(e) = 1 - (1 - e)^3 - \binom{3}{2}(1 - e)^2 e$$

- ▶ For the above example $p(e) = 0.0003$, which is significantly lower than a single model

- ▶ **Dropout** is a regularization technique used to prevent overfitting in neural networks.
- ▶ During training, it randomly drops (set to 0) a fraction of neurons in each layer based on a specified probability for every forward pass.

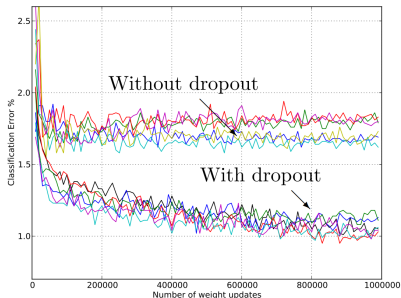


(a) Standard Neural Net



(b) After applying dropout.

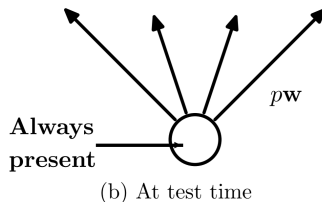
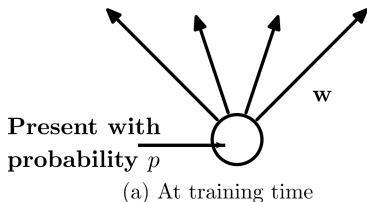
- ▶ Why is dropping neurons randomly useful?
 1. By dropping a subset of neurons during each forward pass, the network learns **not to rely too heavily on specific connections**, encouraging the network to use more connections.
 2. Dropout acts as an **efficient ensemble** of multiple smaller networks, each trained on a random subset of neurons.
 3. More Connections + Diverse Ensemble = **less overfitting**.



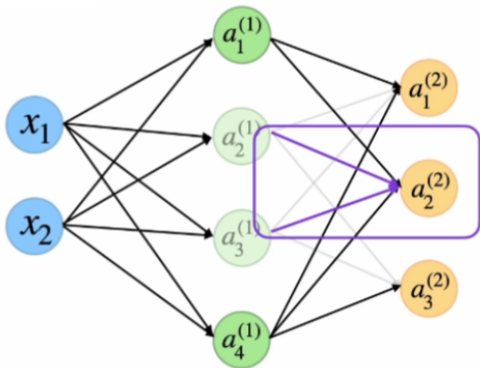
Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SVM on Fisher Vectors of Dense SIFT and Color Statistics	-	-	27.3
Avg of classifiers over FVs of SIFT, LBP, GIST and CSIFT	-	-	26.2
Conv Net + dropout (Krizhevsky et al., 2012)	40.7	18.2	-
Avg of 5 Conv Nets + dropout (Krizhevsky et al., 2012)	38.1	16.4	16.4

Table 6: Results on the ILSVRC-2012 validation/test set.

- ▶ We drop neurons during training, but what should we do during inference?
- ▶ **During inference:** We use all neurons.



- ▶ but...



Assume we have
present probability
 $p = 0.5$

During testing,
**when nodes are
present again,**
activations are
 $2 \times$ as large!

- ▶ **Problem:** Having inconsistent values during inference compared to training can cause the network to produce worse results.
- ▶ **Solution:** Scale the output during inference by p to keep the same expected activation as in training.
- ▶ **Example:**
 - **During training ($p=0.1$):**
 - ▶ Activation value = 2
 - ▶ Active only 10% of the time
 - ▶ \Rightarrow Effective contribution = $2 \times 0.1 = 0.2$
 - **During inference:**
 - ▶ Always active (\Rightarrow Activation value = 2)
 - ▶ Scale by (p) = 0.1

$2 \times 0.1 = 0.2$
 - **Result:** Consistent activation scale between training and inference.

- ▶ In PyTorch, dropout behavior is controlled by the model's mode:

```
# Training Mode:  Enables dropout
model.train()
output = model(input)
# Evaluation Mode: Disables dropout, scales activations
model.eval()
output = model(input)
```

- ▶ **Key Insight:** Networks train better when inputs are normalized
- ▶ So why not normalize intermediate layers too?
- ▶ **Problem:** forcing every layer to output zero-mean, unit-variance values might be too restrictive (not always optimal).
- ▶ **Solution:** Let the network learn if it wants normalization!

For each feature in a layer:

1. First normalize: $\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$
 - μ : mean across batch dimension
 - σ^2 : variance across batch dimension
 - Computed separately for each feature/channel
2. Then give the network control: $y = \gamma \hat{x} + \beta$
 - γ (scale): Can amplify or reduce the normalized values
 - β (shift): Can move the values away from zero
 - These are learned during training like normal weights!

- ▶ After normalization, outputs are always zero-mean, unit-variance
- ▶ But this might not be optimal for every layer!
- ▶ γ and β let each layer learn:
 - γ : "How much variance do I want?"
 - β : "What should my mean activation be?"
- ▶ Two extremes the network can learn:
 - "Keep normalization": $\gamma \approx 1, \beta \approx 0$
 - "Undo normalization": γ and β restore original scale and shift
- ▶ Network learns what's best for each layer!

Mathematically, for batch size N , at each feature/channel j :

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad (\text{batch mean})$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad (\text{batch variance})$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \quad (\text{normalize})$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad (\text{learnable transform})$$

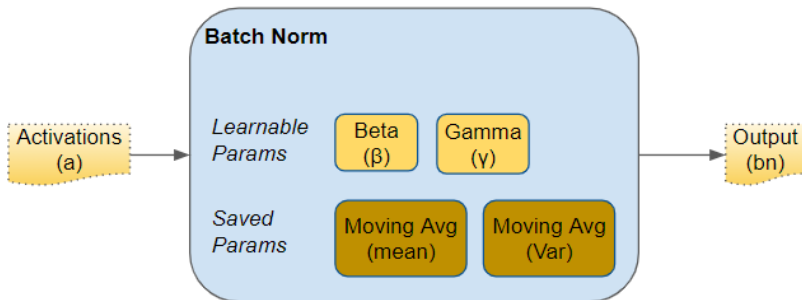
► During Training:

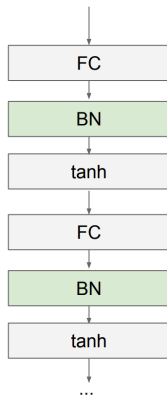
- Use statistics from current batch
- Keep running average of mean and variance (to use later in inference):

$$\mu_{running} = 0.9 \times \mu_{running} + 0.1 \times \mu_{batch}$$

► During Inference (We can't depend on mini-batches):

- Use stored running averages ($\mu_{running}, \sigma_{running}^2$)
- These are fixed values from training
- No batch statistics needed!





Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

- ▶ In Pytorch, batch normalization behavior is controlled by model's mode:

```
# Training Mode: Use batch statistics
model.train()
output = model(input)
# Evaluation Mode: Use running statistics
model.eval()
output = model(input)
```

► Advantages:

- Makes deep networks much easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!

⁰Slide based on CS231n by Fei-Fei Li, Yunzhu Li & Ruohan Gao

► Advantages:

- Makes deep networks much easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!

► Disadvantages:

- Behaves differently during training and testing: this is a very common source of bugs!

⁰Slide based on CS231n by Fei-Fei Li, Yunzhu Li & Ruohan Gao

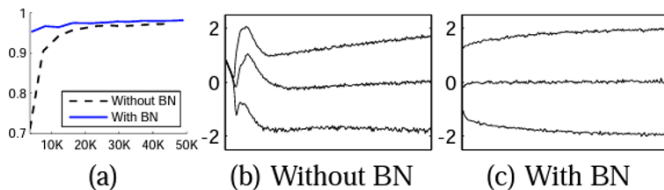


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as $\{15, 50, 85\}$ th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

► Step 1: Initial Setup

- Start with a pretrained model and just finetune when possible; it saves time and improves results.
- Define an initial architecture without regularization (e.g., dropout) or augmentations.
- Set up validation strategy and choose an appropriate evaluation loss and metric.
- Train the model to get a **baseline score**.

► Step 2: Improvement Process

- Overfitting is common at the start; use regularization techniques like:
 - Dropout, batch normalization,...
 - Perform error analysis to identify weaknesses and choose appropriate augmentations or preprocessing.
 - Tune hyperparameters: layers, epochs, learning rate, batch size, etc.
 - Optionally, use ensembling to boost scores (requires more resources).
- **Key Tip:** Track scores and improvements at every step to measure progress effectively.

► Step 3: Finalization

- Save the optimized model for deployment.
- Use the model for inference in real-world applications.