edureka!
a Veranda Enterprise | ILLINOIS TECH

POST GRADUATE PROGRAM IN GENERATIVE AI AND ML

Deep Learning and Neural Network Architectures

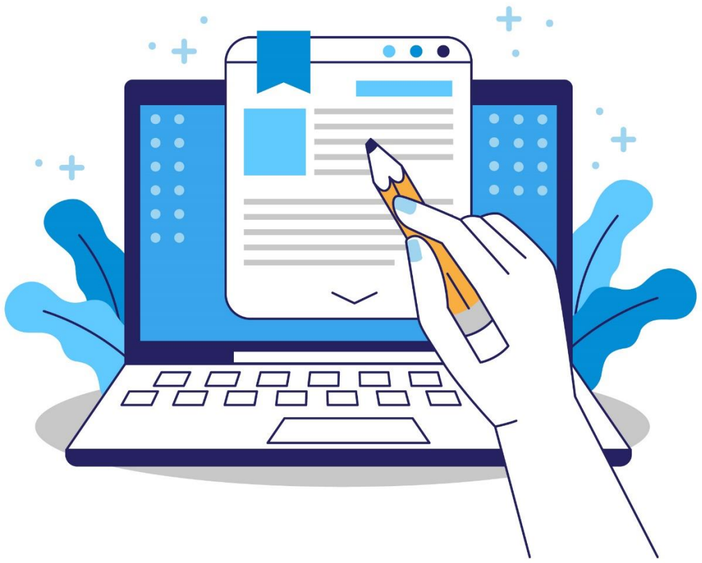
edureka!
a Veranda Enterprise

ILLINOIS TECH

Tuning and Optimizing Deep Neural Networks

Topics

- Introduction to Model Optimizers
- Model Optimization Techniques
- Learning rate schedule Strategies
- Regularization Techniques
- Batch normalization and layer normalization
- Data augmentation techniques
- Gradient clipping
- Vanishing and exploding gradient problem
- Model checkpointing and saving
- Tuning with validation sets
- Grid search and random search
- AutoML for hyperparameter tuning
- Visualizing training with TensorBoard



Learning Objectives

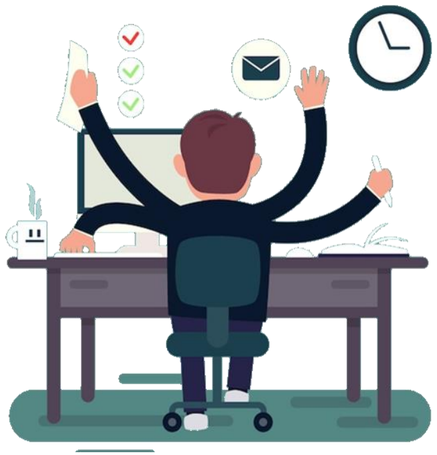
By the end of this lesson, you will be able to:

- Understand how optimizers and learning rates affect training.
- Improve generalization with regularization and augmentation.
- Handle training issues like gradient instability.
- Monitor progress with validation sets and TensorBoard.
- Perform hyperparameter tuning and manage models using Keras.

Introduction To Model Optimizers

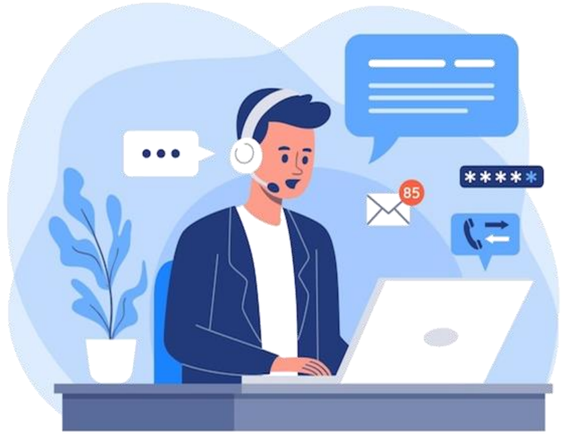
Introduction to Model Optimizers

Model optimizers are essential in deep learning because they guide the learning process by **updating model weights** to minimize the loss function. Without an optimizer, the network wouldn't know **how to improve its predictions.**



Minimize Loss Efficiently

Accelerate Convergence



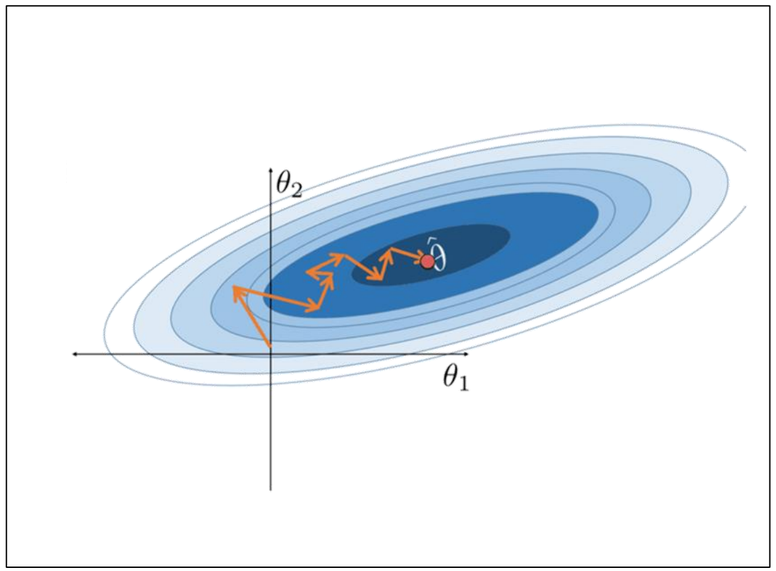
Support Stability in Deep Architectures

Handle Complex, High-Dimensional Spaces

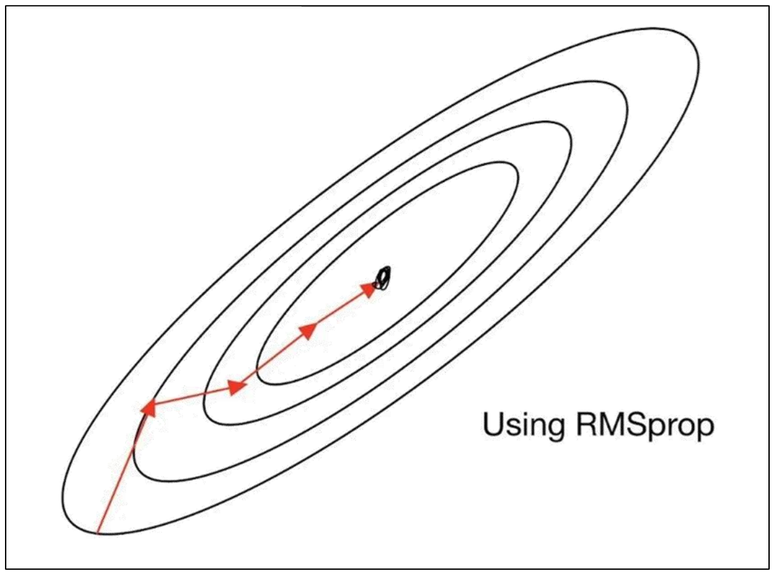
Model Optimizers

Optimization algorithms are crucial in **training deep neural networks**. They update the model's parameters (weights and biases) to **minimize the loss function**.

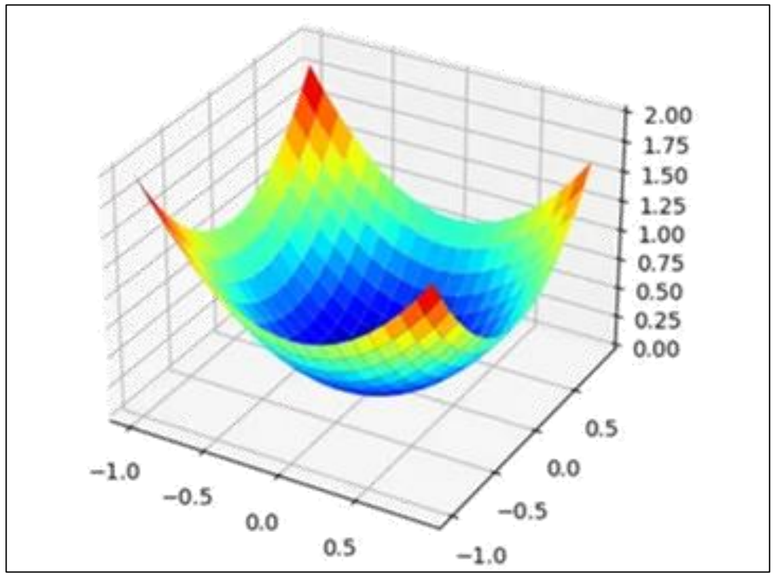
Popular Optimizers:



Stochastic Gradient Descent



RMSprop



ADAM

Model Optimization Techniques

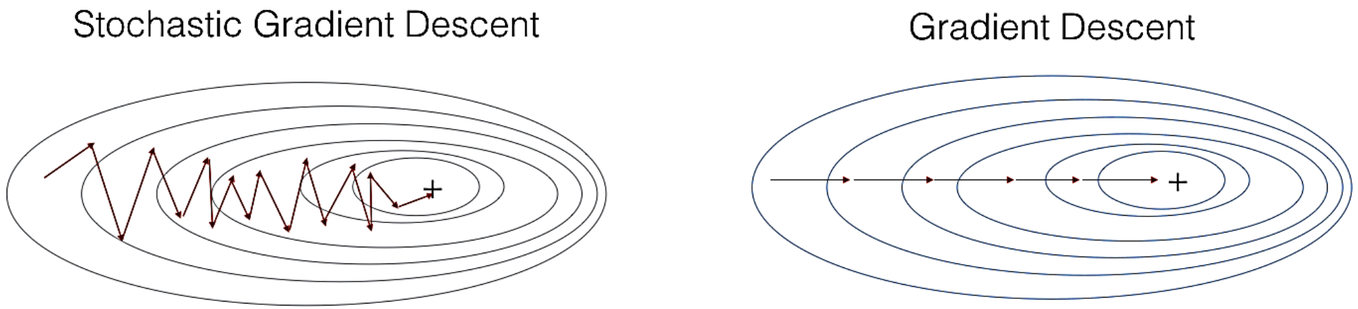
Stochastic Gradient Descent (SGD)

SGD updates parameters using the gradient of the loss with **respect to parameters**, calculated from a single training example or mini-batch.

Update Rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta L(\theta)$$

- ***θ:*** model parameters
- ***η:*** learning rate
- ***∇θ L (θ):*** gradient of loss function w.r.t. parameters



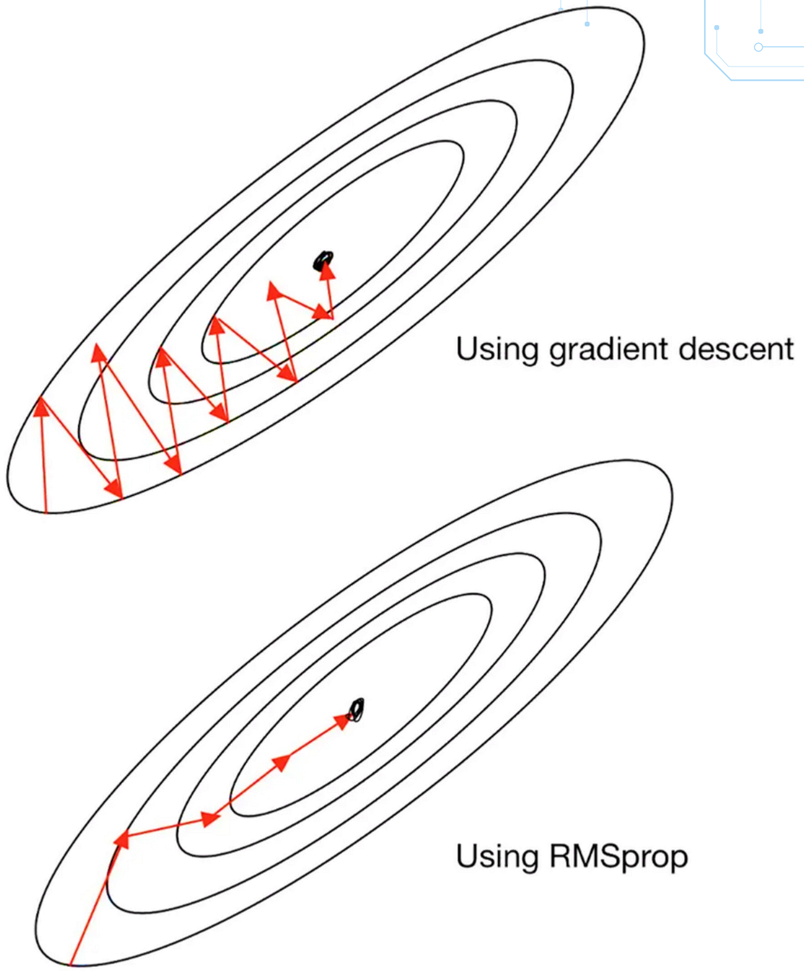
Due to SGD's random nature, the path towards the global cost minimum is **not "direct"** as in GD, but goes **"zig-zag"** if we are visualizing the cost surface in a 2D space.

RMSProp (Root Mean Square Propagation)

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- ***E[g^2]t:*** running average of squared gradients
- ***γ:*** decay rate (e.g., 0.9)
- ***ϵ:*** small constant to avoid division by zero



Adaptive Moment Estimation (ADAM)

- First moment estimate (mean):

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

- Second moment estimate (variance):

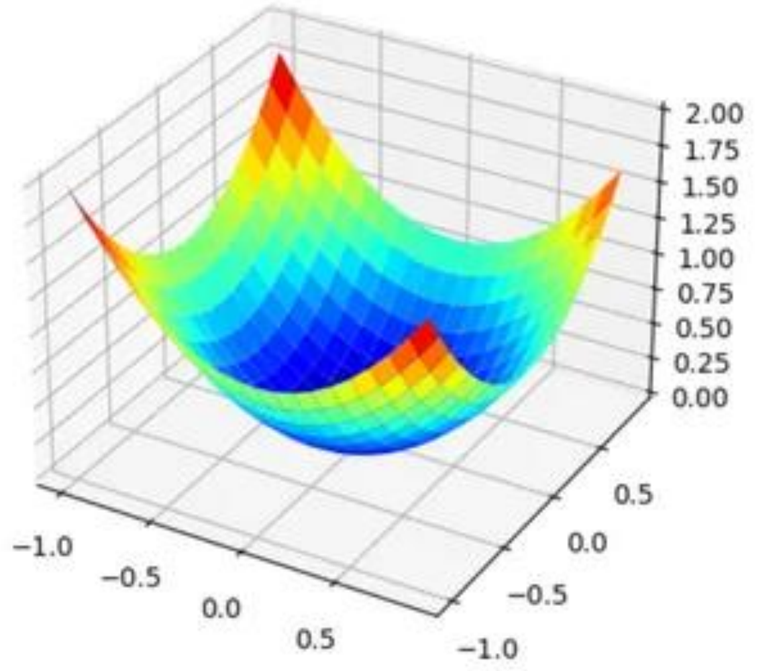
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- Bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Final parameter update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$



Learning Rate Schedule Strategies

Learning Rate Schedules

Learning rate schedules-strategies to dynamically adjust this rate over time-have emerged as indispensable tools for balancing convergence speed, optimization stability, and final model performance.

- Modern frameworks like **PyTorch and TensorFlow/Keras** provide **built-in schedulers**, reflecting their centrality in deep learning pipelines.

- The choice of schedule depends on factors such as dataset size, model architecture, and optimization goals.

Learning Rate Scheduling Strategies

Step Decay Scheduling

- The learning rate at epoch E is computed as:

$$\alpha(E) = \alpha_0 \times \text{factor}^{\left\lfloor \frac{E}{\text{dropEvery}} \right\rfloor}$$

- α_0 is the initial rate,
- **factor** controls decay steepness,
- **dropEvery** specifies the interval.

Exponential Decay

- Exponential decay applies continuous reduction using:

$$\alpha(E) = \alpha_0 \times \text{decayRate}^{E}$$

- Where **decayRate < 1** (typically 0.950.95)

ReduceLROnPlateau

- The update rule is:

$$\alpha_{\text{new}} = \alpha \times \text{factor}$$

- α_0 is the initial rate,
- **factor** controls decay steepness

Learning Rate Scheduling Strategies (contd.)

Cosine Annealing

- Cosine annealing cyclically varies η along a cosine curve, reaching minima at cycle ends. The PyTorch Cosine Annealing LR implements:

$$\alpha(E) = \alpha_{\min} + \frac{1}{2}(\alpha_{\max} - \alpha_{\min})\left(1 + \cos\left(\frac{E\pi}{T_{\max}}\right)\right)$$

- where T max is the cycle length, and *α* min is the floor rate

Cyclic Learning Rates

- The Cyclic LR in PyTorch oscillates η between αbase and αmax using triangular, triangular2, or exponential scaling. For triangular mode:

$$\alpha(E) = \alpha_{\text{base}} + (\alpha_{\max} - \alpha_{\text{base}}) \times \max(0, 1 - |\frac{E}{T} - 1|)$$

- where T is the cycle period

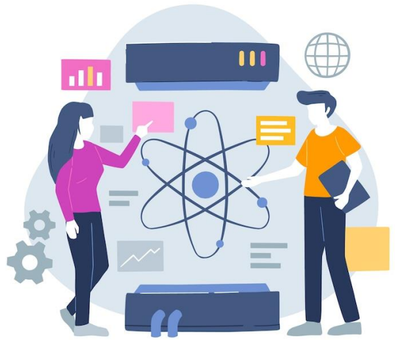
Regularization Techniques

Regularization Techniques

Regularization is a collection of techniques used to **prevent overfitting** in deep neural networks. Overfitting happens when a model performs **very well on training data** but **poorly on unseen data**, due to high model complexity.

Why is Regularization Required?

Prevent Overfitting



Enable Deeper Models

Improve Generalization

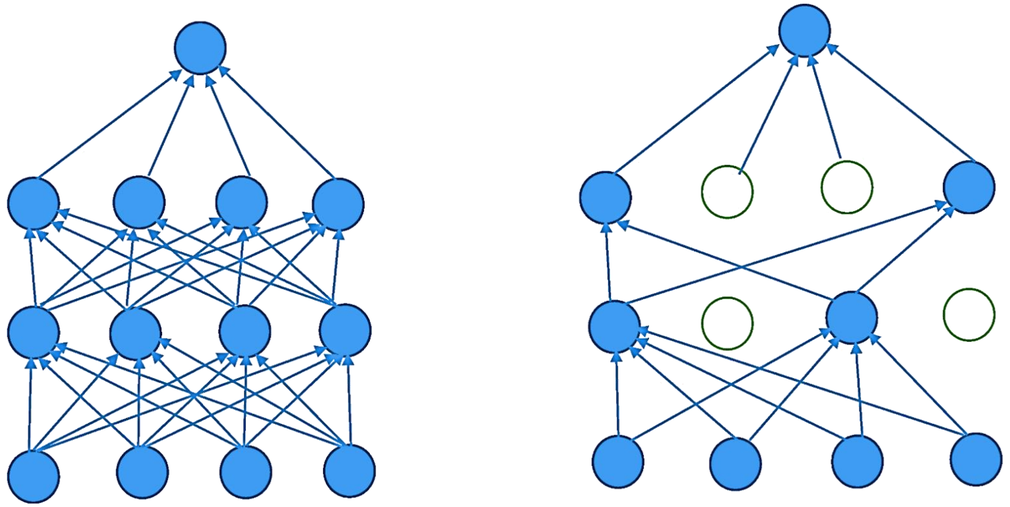
Stabilize Training

Dropout Regularization

A fundamental regularization technique that addresses overfitting by **randomly "dropping" neurons** during training.

Mechanism:

- During each training iteration, **neurons are randomly deactivated** with a probability p (typically 0.2-0.5).
- This creates a different **"thinned" network architecture** for each batch.
- At test time, all neurons remain active, but their **outputs are weighted** by probability p to compensate.



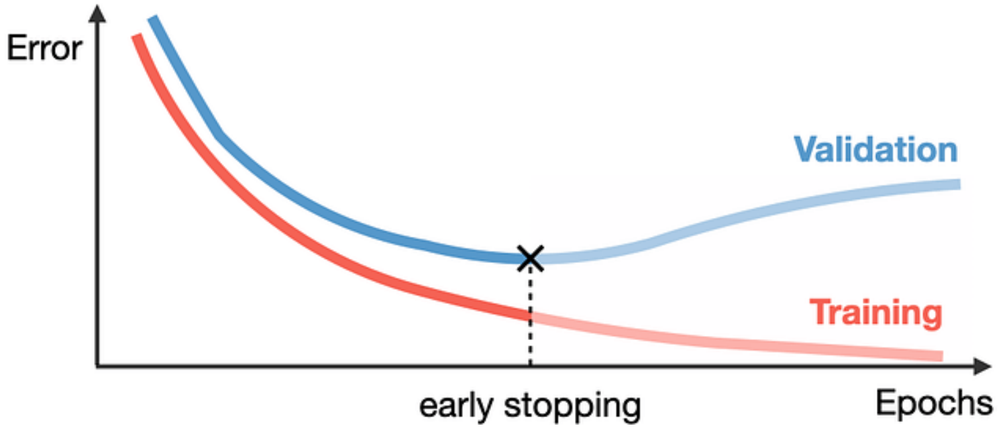
An illustration of a standard neural network (A) and after applying dropout (B).

Early Stopping

Early stopping is one of the simplest yet most effective regularization techniques.

Implementation:

- **Monitor validation metrics** (error/accuracy) during training.
- Stop training when validation performance plateaus or deteriorates.
- Optionally lower the learning rate before final stopping.



L1 and L2 Regularization

- **L1 Regularization** (**Lasso Regression**): Promotes sparsity by forcing many weights to zero, effectively performing feature selection.
- L1 norm: When p=1, we get L1 norm, the sum of the absolute values of the components in the vector:

$$L_1(\mathbf{x}) = ||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i|$$

- **L2 Regularization** (**Ridge Regression**): Prevents weights from becoming excessively large while preserving important weight components.
- L2 norm: When p=2, we get L2 norm, the Euclidean distance of the point from the origin in n-dimensional vector space:

$$L_2(\mathbf{x}) = ||\mathbf{x}||_2 = (\sum_{i=1}^{n} |x_i|^2)^{1/2}$$

Batch Normalization and Layer Normalization

Batch Normalization

Batch Normalization Statistics

- **Mean Calculation:** For each feature/channel, BatchNorm computes the mean across all samples in the current mini-batch.

$$\mu_k = (1/m) \sum_{i=1}^{m} x^k_i$$

- **Variance Calculation:** Similarly, BatchNorm computes the variance across all samples in the mini-batch for each feature:

$$\sigma^2_k = (1/m) \sum_{i=1}^{m} (x^k_i - \mu_k)^2$$

This measures how spread out the values are from the mean for that specific feature

Where,

- m = batch size
- x^k_i = value of the kth feature for the ith sample.

Layer Normalization

Layer Normalization Statistics:

- **Mean Calculation:** For each individual sample, LayerNorm computes the mean across all features/channels (and spatial dimensions if present):

$$\mu_i = (1/H) \sum_{j=1}^{H} a_{ij}$$

- **Variance Calculation:** For each sample, LayerNorm computes the variance across all features:

$$\sigma^2_i = (1/H) \sum_{j=1}^{H} (a_{ij} - \mu_i)^2$$

This measures the spread of feature values within each individual sample

Where,

- H = number of hidden units/features
- a_{ij} = jth feature of the ith sample.

Batch Normalization vs. Layer Normalization

Aspect	Batch Normalization	Layer Normalization
Normalization Dimension	Across samples in mini-batches	Across features within each sample
Batch Size Dependency	Requires sufficiently large batches	No dependency on batch size
Best Applications	CNNs, feedforward networks	RNNs, Transformers, variable sequence models
Computation	Statistics across mini-batches	Statistics across features
Inference Behavior	Requires stored statistics from training	Consistent behavior between training and inference

Data Augmentation Techniques

Data Augmentation

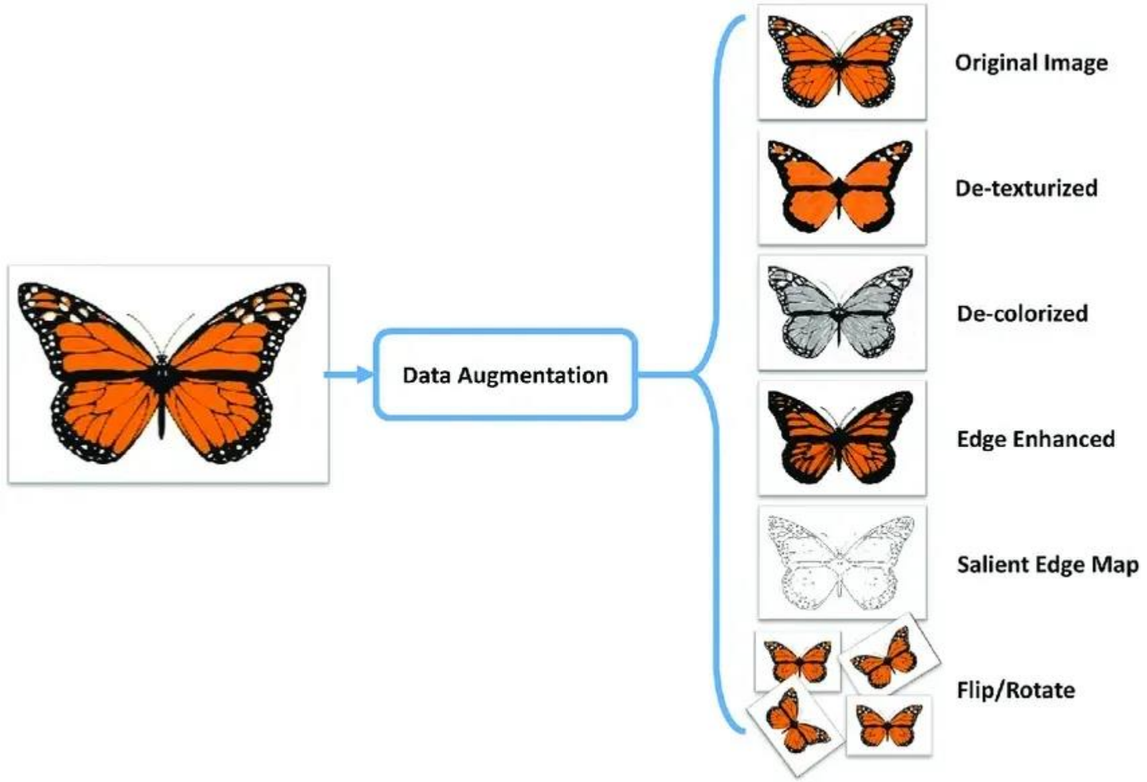
Data augmentation artificially expands the training dataset by creating modified versions of existing samples.

Implementation Approaches:

- For image data: rotations, flips, crops, color shifts, etc.
- Advanced techniques: MixUp, AugMix, and CutMix.
- Domain-specific transformations for other data types.

Key Augmentation Techniques:

- Noise and Advanced Techniques
- Color Transformations
- Geometric Transformations
- Text Data Augmentation



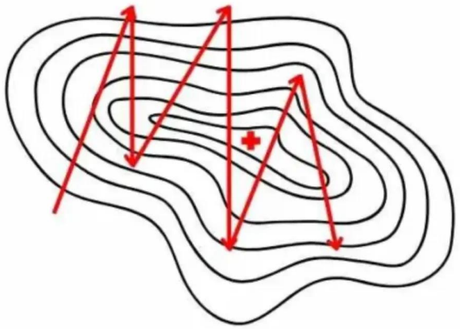
Data Augmentation Techniques

Technique	Code (using keras.preprocessing.image.ImageDataGenerator)	Function
Rotation	rotation_range	Rotates images randomly in a specified degree range.
Width Shift	width_shift_range	Shifts images horizontally.
Height Shift	height_shift_range	Shifts images vertically.
Zoom	zoom_range	Randomly zooms in or out of the image.
Horizontal Flip	horizontal_flip=True	Flips images horizontally.
Shear	shear_range	Applies random shearing transformations.
Brightness Adjust	brightness_range	Randomly changes image brightness.

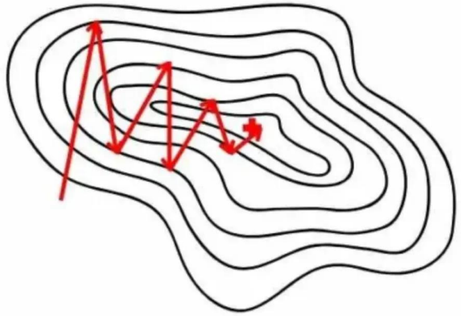
Gradient Clipping

Gradient Clipping

Gradient clipping is a crucial optimization technique used to address the **"exploding gradients" problem** during the training of deep neural networks, especially in very **deep architectures and recurrent neural networks (RNNs).**



Without Gradient Clipping



With Gradient Clipping

Exploding gradients cause unstable training in deep networks due to excessively large updates.Gradient clipping stabilizes learning by capping these gradients within safe limits.

How Does Gradient Clipping Work?

1. Clipping by Value

- Each component of the gradient vector is clipped to lie within a specified minimum and maximum value (e.g., between -1 and 1).
- If a gradient exceeds the threshold, it is set to the threshold value.

2. Clipping by Norm (Norm-Based Clipping)

- The overall norm (e.g., L2 norm) of the gradient vector is computed.
- If this norm **exceeds a specified threshold**, the entire gradient vector is scaled down proportionally, so its norm matches the threshold, preserving the direction of the gradient.
- Mathematical Expression: If $\|g\| \geq c$, then

$$g \leftarrow c \cdot \frac{g}{\|g\|}$$

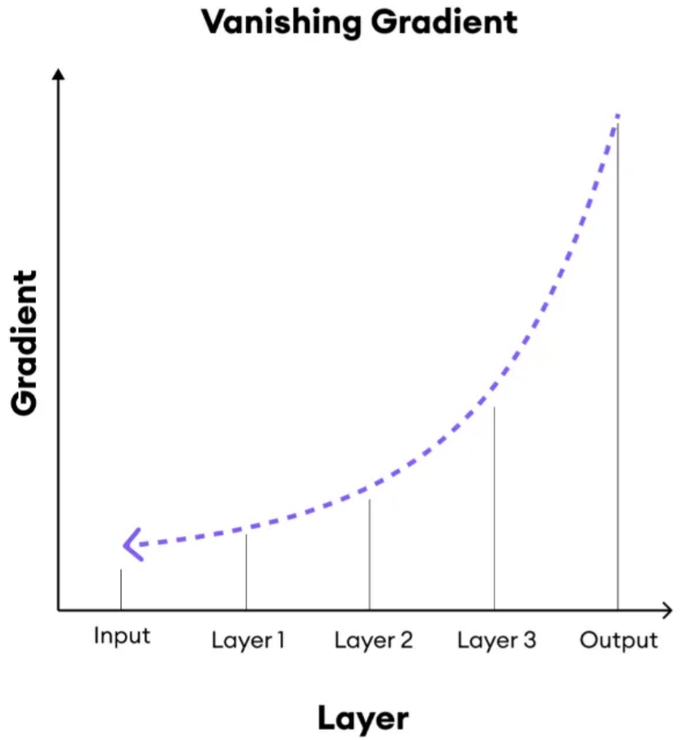
- where gg is the gradient vector and cc is the threshold67.

Vanishing and Exploding Gradient Problem

Vanishing Gradient Problem

The vanishing gradient problem occurs when **gradients become extremely small** as they **propagate backward** through the network during training.

As the gradient flows backward from the output layer to the input layer, it gets **repeatedly multiplied by small values,** causing it to diminish exponentially.



Solutions and Mitigation Strategies for Vanishing Gradients

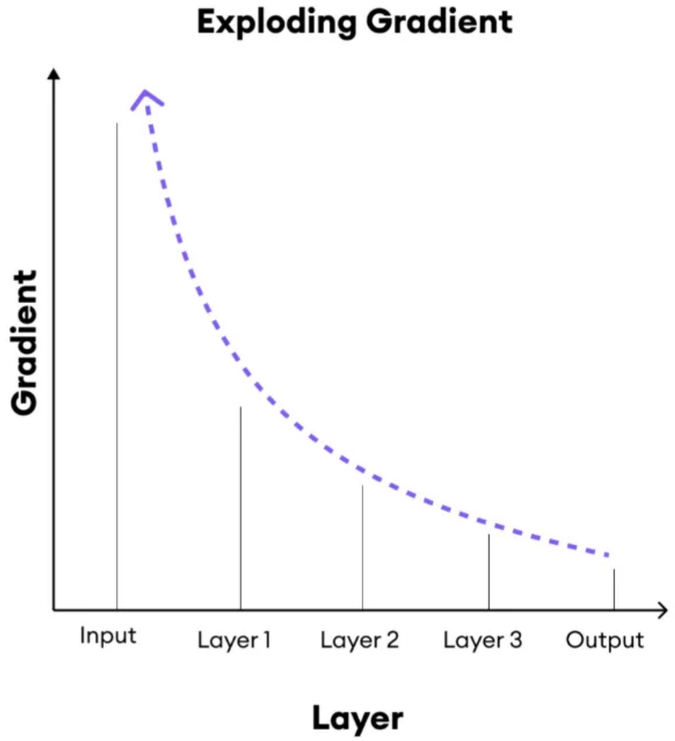
- Alternative Activation Functions
- Leaky ReLU
- Skip Connections
- Proper Weight Initialization

Exploding Gradient Problem

The exploding gradient problem occurs when gradients become excessively large during backpropagation, causing extreme weight updates

Solutions and Mitigation Strategies for Exploding Gradients

- Gradient Clipping
- Weight Initialization
- Batch Normalization
- L2 Regularization



Model Checkpointing and Saving

What is Model Checkpointing?

Model checkpointing refers to the **periodic saving** of a neural network's complete state-including **weights, optimizer parameters, and training metadata**-during the training process

The process involves **three key components:**

Model Architecture

Optimizer State

Training Context

Checkpointing Lifecycle in Distributed Training

- At scale, checkpointing becomes **computationally nuanced**.
- Distributed training frameworks often shared models across GPUs or nodes, requiring coordinated checkpointing.
- The universal checkpointing paradigm addresses this by **decoupling the checkpoint** format from hardware configurations.

Saving Phase: Maintain distributed parameter shards for I/O efficiency.

Loading Phase: Reconstruct parameters on arbitrary hardware using metadata-driven sharding

This approach enables **elastic resumption**-for instance, continuing training on remaining GPUs after a node failure-while avoiding the prohibitive cost of consolidating terabyte-scale models into single files.

Checkpointing and Saving- TensorFlow / Keras

```
# Save Only Weights After Every Epoch
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint(
    "model_weights.h5", save_weights_only=True, save_best_only=True, monitor='val_loss'
)
model.fit(x_train, y_train, validation_data=(x_val, y_val), callbacks=[checkpoint_cb])

# Save Entire ModelpythonCopyEdit
model.save('full_model.h5')  # Save architecture + weights
# Load it later:
new_model = tf.keras.models.load_model('full_model.h5')

# Resume Training from Checkpoint
model.load_weights("model_weights.h5")
model.fit(...)  # Continue training
```

Checkpointing and Saving- PyTorch

```
# Save Only Model Weights
torch.save(model.state_dict(), "model_weights.pth")
# Load weights:
model.load_state_dict(torch.load("model_weights.pth"))model.eval()  # Set to inference
mode

# Save Model + Optimizer + Epoch
torch.save({
      'epoch': epoch,
      'model_state_dict': model.state_dict(),
      'optimizer_state_dict': optimizer.state_dict(),
}, "checkpoint.pth")

# Resume from Check
checkpoint = torch.load("checkpoint.pth")
model.load_state_dict(checkpoint['model_state_dict’])
optimizer.load_state_dict(checkpoint['optimizer_state_dict’])
start_epoch = checkpoint['epoch']
```

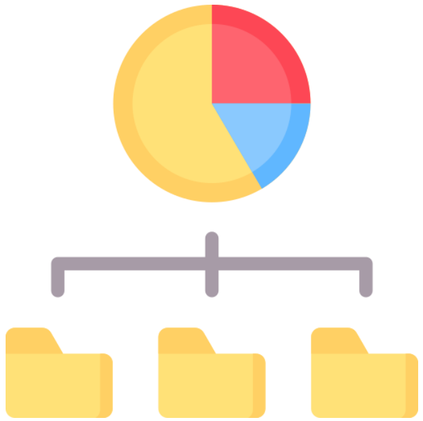
Tuning with Validation Sets

Tuning with Validation Sets

A validation set is your “unseen” yardstick for **choosing hyperparameters and preventing overfit**. You **train on one portion** of data, **tune on another**, and finally **report on a third held-out test set.**

Search hyperparameters

Evaluate to estimate final generalization



Split your data into training, validation, and test sets

Top hyperparameters, retaining on training + validation set

Implementation- Tuning with Validation Sets

Step 1 – Partition Your Data

- Train set (≈ 70–80 %): used to update model weights.
- Validation set (≈ 10–15 %): used only to evaluate intermediate models and compare hyperparameter choices—never for weight updates.
- Test set (≈ 10–15 %): strictly held out until the very end.

Step 2 – Hyperparameter Search Loop

For each candidate configuration (learning rate, batch size, optimizer, # layers, dropout, weight decay, etc.):

- Train the DNN on the train set.
- Compute your chosen metric (loss, accuracy, F1, perplexity) on the validation set at each epoch or after training.
- Log validation metrics alongside hyperparameter values for later comparison.

Implementation- Tuning with Validation Sets

Step 3 – Early Stopping & Checkpointing

- Monitor your validation loss or metric every epoch.
- Stop training when validation performance hasn’t improved for patience epochs (to avoid over-fitting).
- Save the model checkpoint at its best validation score—you’ll roll back to this later.

Step 4 – Learning-Rate Scheduling Based on Validation

- ReduceLROnPlateau: automatically lower the learning rate when validation loss plateaus.
- Warm restarts / cyclical schedules: sometimes tied to validation cycles to escape local minima.

Implementation- Tuning with Validation Sets

Step 5 – Select & Retrain

- Choose the hyperparameter set with the best validation performance (e.g., highest validation accuracy or lowest validation loss).
- Merge train + validation sets, then retrain a new model from scratch using those hyperparameters to maximize data usage.

Step 6 – Final Evaluation

- Evaluate your retrained model once on the untouched test set to estimate real-world generalization.
- Report test metrics; do not peek at test performance during tuning.

Grid search and random search

Grid Search Methodology

Grid Search is a systematic approach to **hyperparameter optimization** that works by evaluating all possible combinations from a predefined set of hyperparameter values.

Code example

```
param_grid = {
    'learning_rate': [0.001, 0.01, 0.1],
    'batch_size': [32, 64, 128]
}
grid = GridSearchCV(estimator=model,
param_grid=param_grid, cv=3)
grid_result = grid.fit(X, Y)
```

This implementation would **evaluate all nine combinations** of the specified learning rates and batch sizes, **using 3-fold cross-validation** to assess performance.

Random Search Methodology

Random Search approaches hyperparameter optimization by **randomly sampling configurations** from predefined distributions rather than exhaustively evaluating a fixed grid.

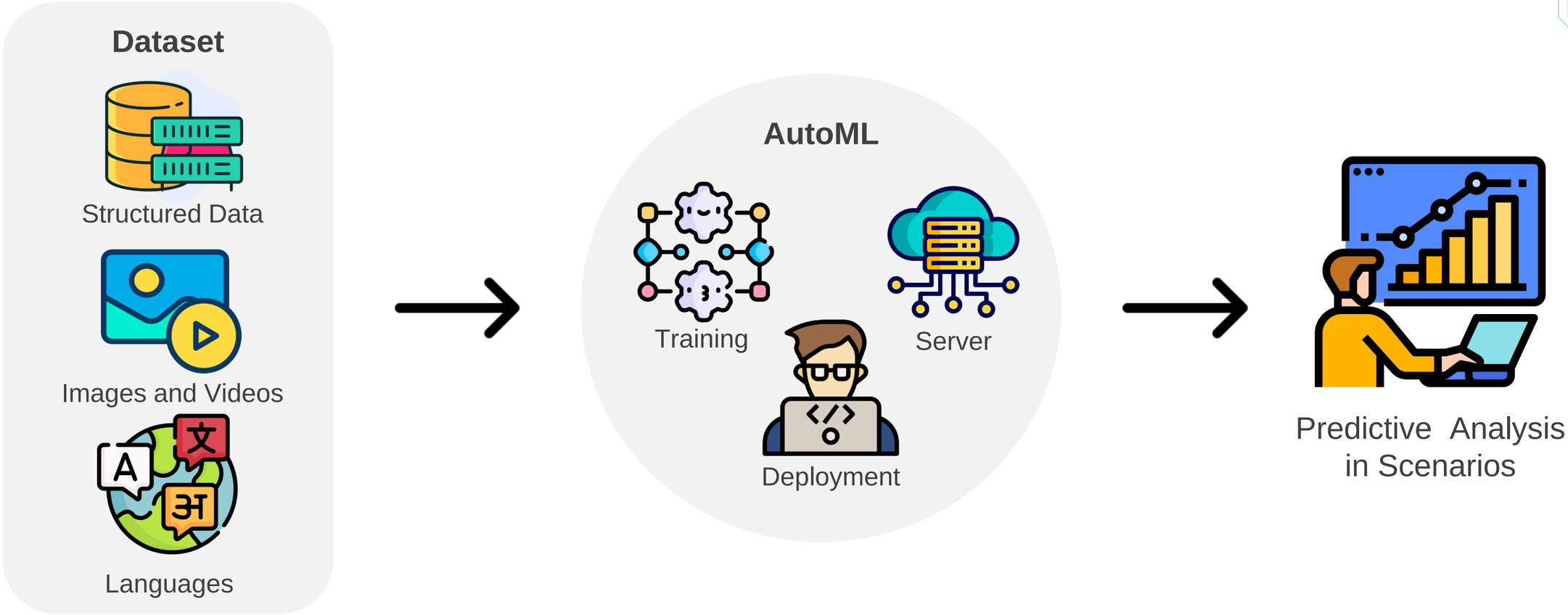
Code example

```
tuner = RandomSearch(
    hypermodel=model_builder,
    objective='val_accuracy',
    max_trials=50,
    seed=42
)
tuner.search(x_train, y_train, epochs=50,
validation_data=(x_val, y_val))
```

In this example, max_trials=50 indicates that **50 random hyperparameter configurations** will be evaluated

AutoML for Hyperparameter Tuning

AutoML in Hyperparameter Tuning

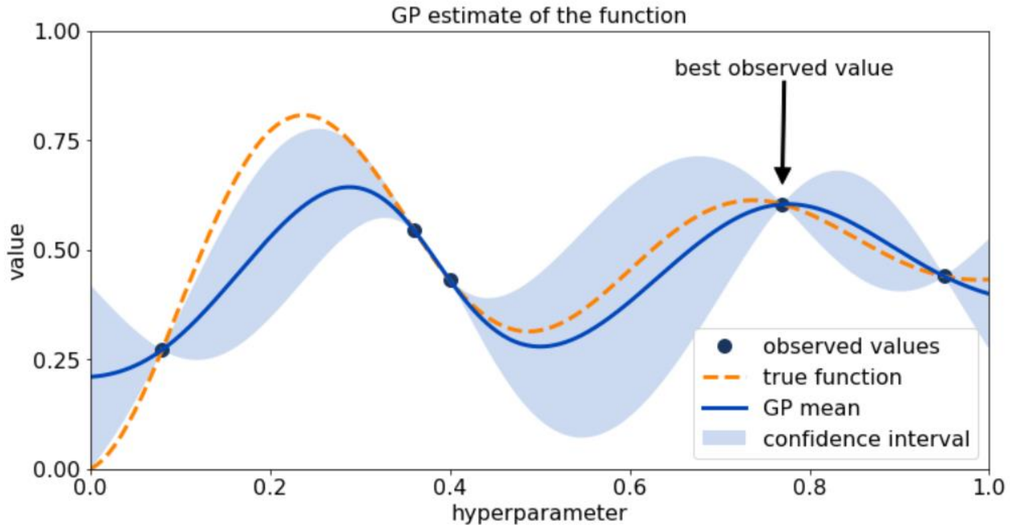


Approach - Bayesian Optimization

One of the most sophisticated approaches in AutoML's toolkit and this technique builds a **probabilistic model** of the objective function (e.g., validation accuracy) to predict which hyperparameters will perform best.

The approach works through an iterative process:

- Initialize by evaluating a few **random hyperparameter configurations**
- **Build a surrogate model** (typically a Gaussian Process) of the objective function
- Use an **acquisition function** to determine the next promising configuration to evaluate
- **Update the surrogate model** with new results
- Repeat until **convergence or budget exhaustion**



Approach - Hyperband Algorithm

Hyperband represents an **advancement over traditional random search methods**, employing adaptive resource allocation and early stopping to quickly converge on high-performing models20.

This algorithm operates using a sports championship-style bracket:

- It begins by **training many models** with different hyperparameters for a small number of epochs
- Only the **top-performing subset** (typically half) advances to the next "round"
- Survivors receive **additional training resources** (more epochs)
- The process continues **until identifying the best configuration**

Approach - Genetic Algorithms

Genetic algorithms apply **evolutionary principles to hyperparameter optimization**, treating configurations as "individuals" in a population that evolves over generations:

- Initialize a population of random hyperparameter configurations
- Evaluate each **configuration's fitness** (model performance)
- Select the **best performers** for "reproduction"
- Create new configurations through **crossover and mutation operations**
- **Replace the original population** with the new generation
- Repeat until convergence

Simpler than other AutoML approaches, **grid and random** search remain relevant components in hyperparameter tuning

The Role and Impact of Hyperparameters

Hyperparameters are adjustable configuration variables that **govern the training process** of deep neural networks but are not learned during training itself.

Architectural hyperparameters

Number of hidden layers, nodes per layer, activation functions

01

Optimization hyperparameters

Learning rate, batch size, optimizer choice

02

Regularization hyperparameters

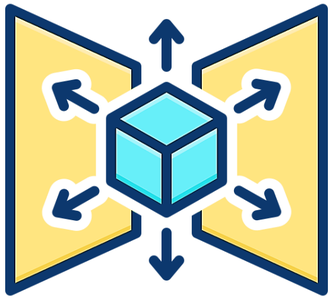
Dropout rates, weight decay coefficients

03

The Hyperparameter Tuning Challenges

Computational expense

Non-intuitive relationships



High dimensionality

Domain specificity

Visualizing Training with TensorBoard

What is TensorBoard?

TensorBoard is a suite of web applications for **inspecting and understanding your TensorFlow** runs and graphs. It allows visualization of metrics such as loss and accuracy, model graphs, histograms of weights, images, and more.

Key Features for Tuning and Optimization:

- Real-Time Metric Tracking
- Hyperparameter Tuning
- Model Graph Visualization
- Histograms and Distributions
- Embedding Visualizations
- Image, Audio, and Text Summaries

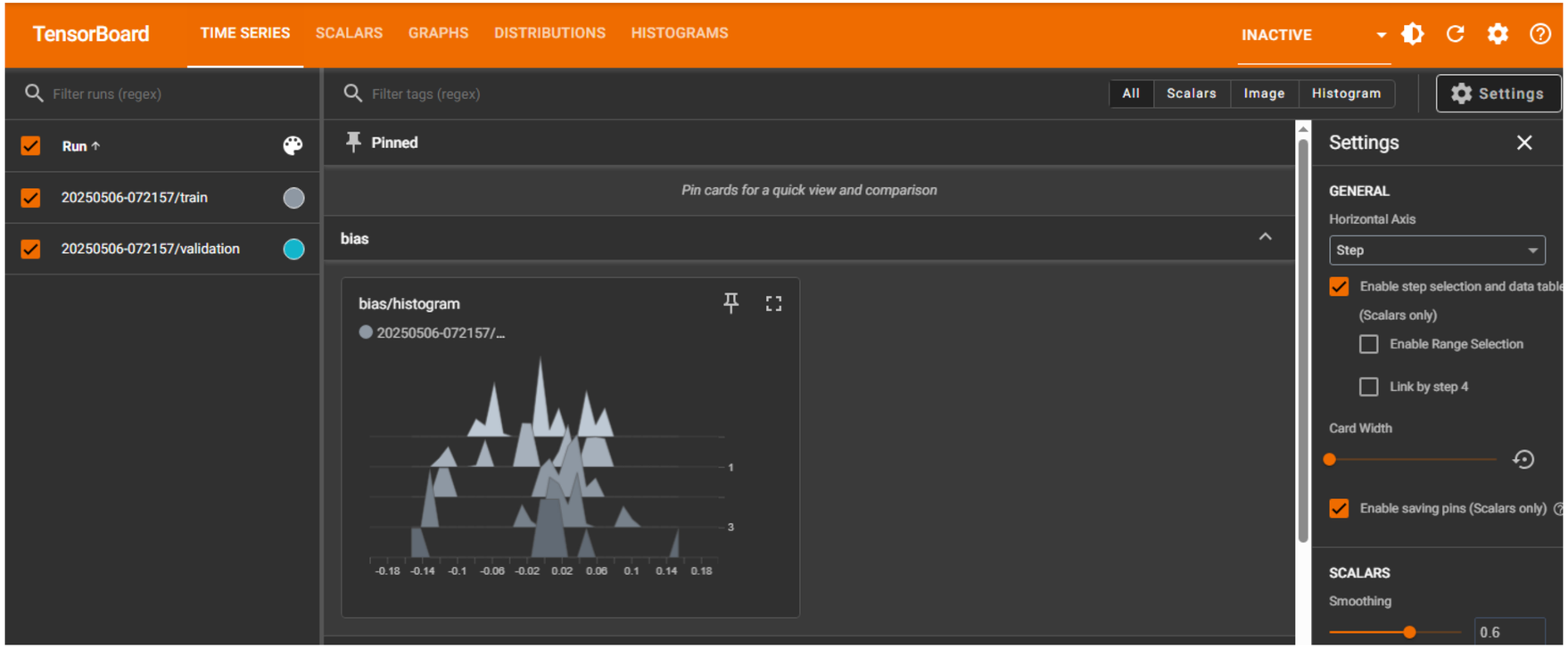
Major Visualizations

Visualization	Description
Scalars	Plots training/validation **loss**, **accuracy**, learning rate over time
Graphs	Visualizes **model architecture** (layers and data flow)
Histograms	Shows how **weights**, **biases**, or **activations** evolve per epoch
Images	Displays logged images such as **inputs** or **feature maps**
Distributions	Smooth, detailed view of parameter distributions
Embeddings	Projects high-dimensional data (e.g., word vectors) into 2D or 3D
Text	Logs prediction output, class names, notes, etc.
Hyperparameters	Visualizes multiple runs for parameter comparison
Profiler	Displays bottlenecks in training (data pipeline, GPU utilization, etc.)

Time Series

This tab provides a general view of how scalar metrics evolve over time (or training steps).

- **Main Function**: It allows comparison of multiple scalar metrics (like loss, accuracy) across different runs or experiments.
- **Usage**: Good for tracking training trends, model performance, and comparing multiple models in one plot.



Scalars

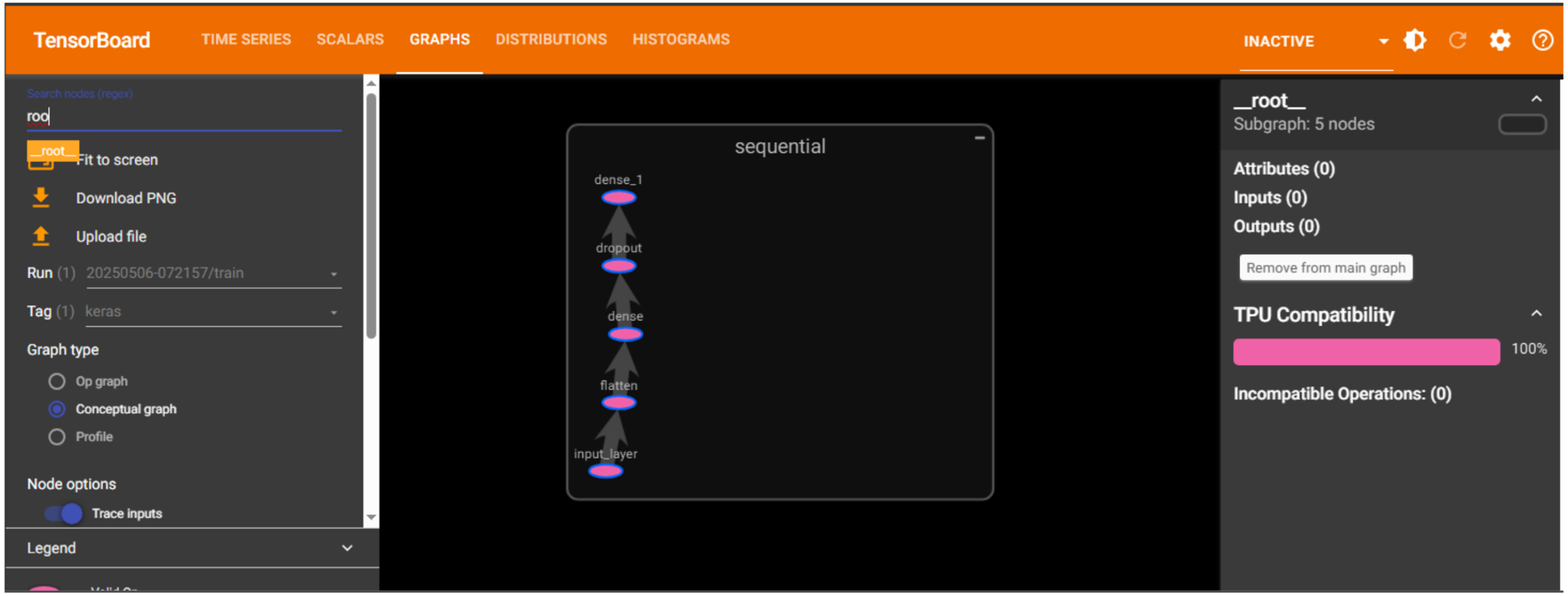
Displays single-value metrics (scalars) such as training loss, validation accuracy, learning rate, etc.

- **Main Function**: Helps visualize and track how scalar metrics change during training.
- **Usage**: Ideal for monitoring overfitting, convergence, and training dynamics.

Graphs

Shows the computational graph (model architecture) of your neural network.

- **Main Function**: Helps inspect the structure of the model, layers, and operations.
- **Usage**: Useful for debugging model architecture and verifying the correct flow of tensors.



Distributions

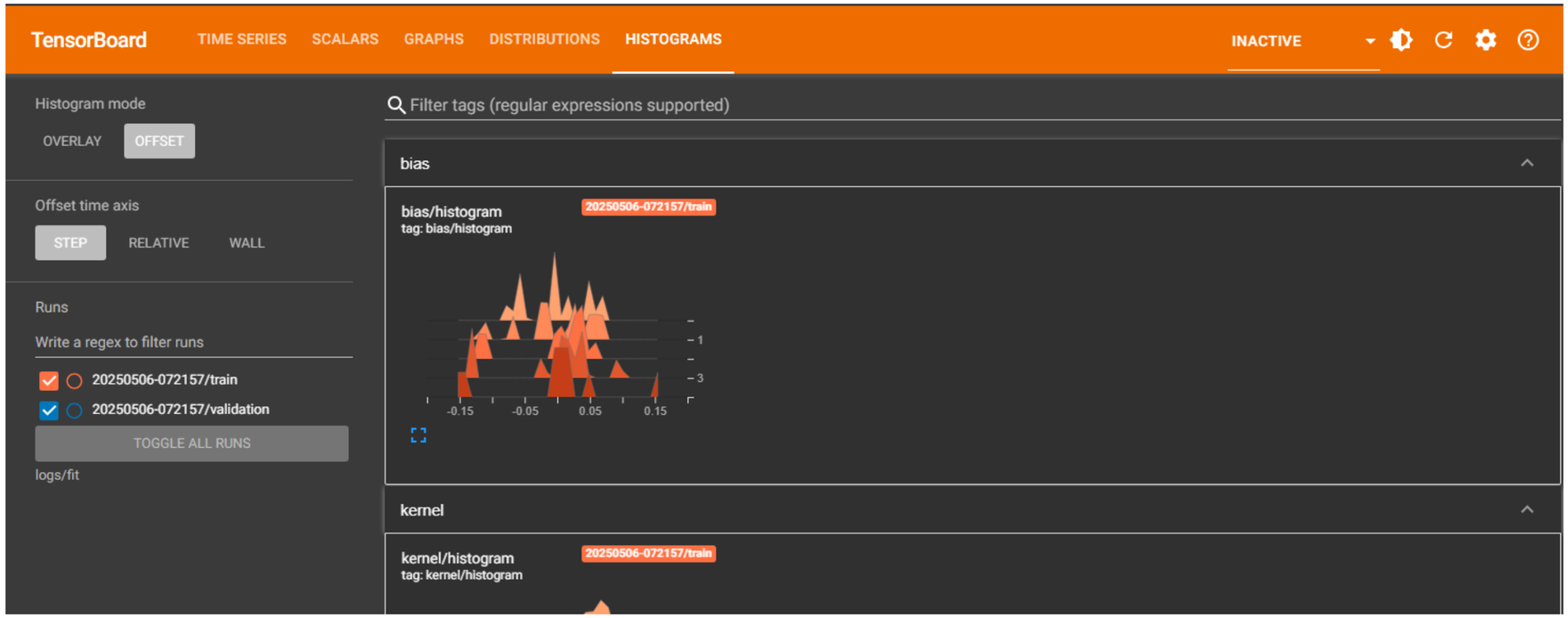
Visualizes the distribution of tensors (e.g., weights, biases) over time.

- **Main Function**: Tracks how model parameters change and stabilize during training.
- **Usage**: Helps detect vanishing/exploding gradients or abnormal parameter updates.

Histograms

Plots histograms of tensor values at each training step.

- **Main Function**: Provides detailed view of how the range and shape of tensors (like weights) evolve.
- **Usage**: Complements the distributions tab to deeply analyze parameter shifts.



CNN-based image classification using CIFAR-10 (Demonstration)

Note: Refer to the Module 2: Demo 1 on LMS for detailed steps.

Sentiment Analysis using CNN + Hyperparameter Tuning (Demonstration)

Note: Refer to the Module 2: Demo 2 on LMS for detailed steps.

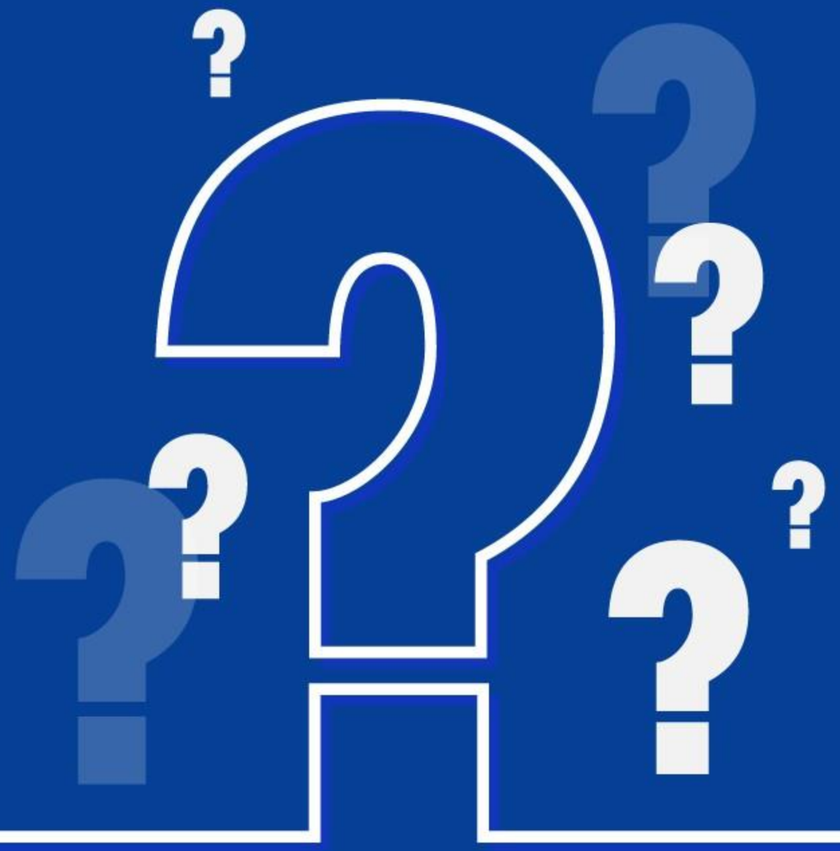
Summary

In this lesson, you have learned to:

- Understand optimizers and learning rate techniques.
- Apply regularization, normalization, and augmentation.
- Handle gradient issues and training instabilities.
- Perform hyperparameter tuning using manual/search/AutoML.Save, restore, and run a basic DNN in Keras.

edureka! a Veranda ENTERPRISE

Questions



edureka! a Veranda ENTERPRISE

Feedback

edureka! a Veranda ENTERPRISE

Thank You

For information, Please Visit our Website
www.edureka.co