The Journey of PyTorch

12 November 2024 19:14

PyTorch Overview

- Open-Source Deep Learning Library: Developed by Meta AI (formerly Facebook AI Research).
- Python & Torch: Combines Python's ease of use with the efficiency of the Torch scientific computing framework, originally built with Lua. Torch was known for high-performance tensor-based operations, especially on GPUs.

PyTorch Release Timeline

PyTorch 0.1 (2017)

- Key Features:
  - Introduced the dynamic computation graph, enabling more flexible model architectures.
  - Seamless integration with other Python libraries (e.g., numpy, scipy).
- Impact:
  - Gained popularity among researchers due to its intuitive, Pythonic interface and flexibility.
  - Quickly featured in numerous research papers.

PyTorch 1.0 (2018)

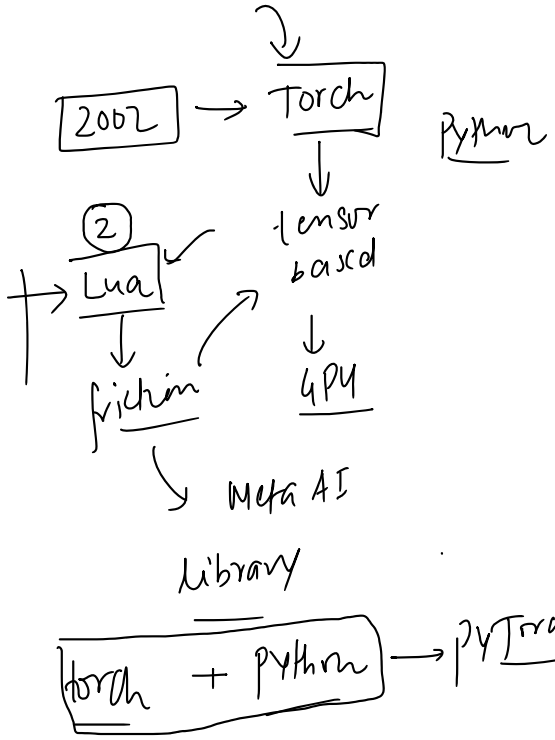
- Key Features:
  - Bridged the gap between research and production environments.
  - Introduced TorchScript for model serialization and optimization.
  - Improved performance with Caffe2 integration.
- Impact:
  - Enabled smoother transitions of models from research to deployment.

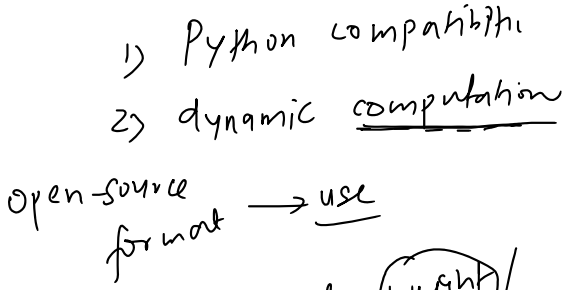
PyTorch 1.x Series

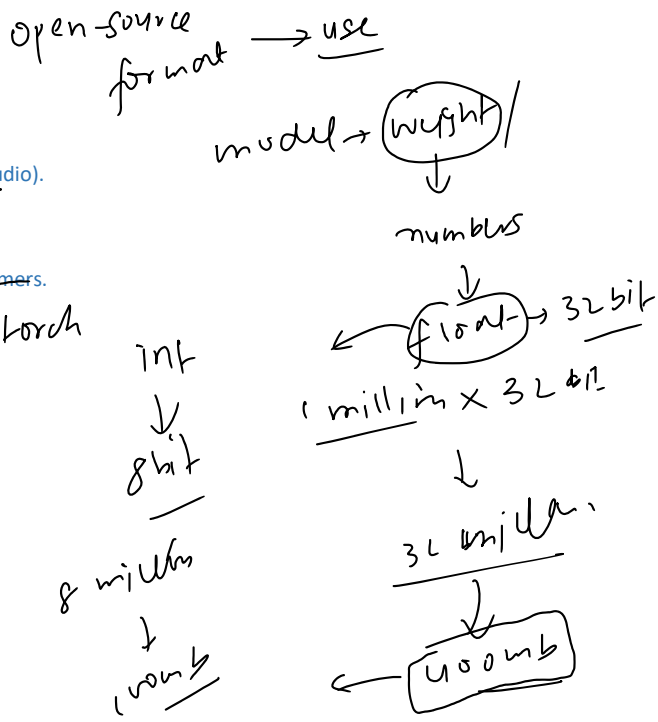
- Key Features:
  - Support for distributed training.
  - ONNX compatibility for interoperability with other frameworks.
  - Introduced quantization for model compression and efficiency.
  - Expanded ecosystem with torchvision (CV), torchtext (NLP), and torchaudio (audio).
- Impact:
  - Increased adoption by the research community and industry.
  - Inspired community libraries like PyTorch Lightning and Hugging Face Transformers.
  - Strengthened cloud support for easy deployment.

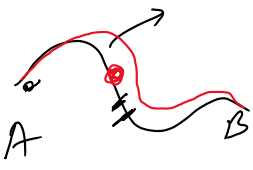
PyTorch 2.0

- Key Features:
  - Significant performance improvements.
  - Enhanced support for deployment and production-readiness.
  - Optimized for modern hardware (TPUs, custom AI chips).
- Impact:
  - Improved speed and scalability for real-world applications.
  - Better compatibility with a variety of deployment environments.









static
comp

Core Features

12 November 2024 19:15

1. Tensor Computations
2. GPU Acceleration
3. Dynamic Computation Graph
4. Automatic Differentiation ✓
5. Distributed Training
6. Interoperability with other libraries

PyTorch Vs TensorFlow

12 November 2024 19:15

2015 → industry

Aspect	PyTorch	TensorFlow	Verdict
Programming Language	Primarily Python; provides a Pythonic interface with deep integration.	Supports multiple languages: Python (primary), C++, Java, JavaScript (TensorFlow.js), and Swift (experimental).	**Depends**: PyTorch for Python-centric development; TensorFlow for multi-language support.
Ease of Use	Known for its intuitive and Pythonic syntax, making it user-friendly and easier for beginners.	TensorFlow 2.x improved usability with Keras integration, but can still be complex.	**PyTorch Wins**: Generally considered easier to learn and more intuitive.
Deployment and Production	Offers TorchScript for model serialization; PyTorch Mobile supports mobile deployment; growing support for production environments.	Strong production support with TensorFlow Serving, TensorFlow Lite, and TensorFlow.js; more mature tools.	**TensorFlow Wins**: More mature and comprehensive deployment options.
Performance	Competitive performance; dynamic graphs may introduce slight overhead; optimized with TorchScript and JIT compilation.	Optimized through static graphs and XLA compiler; efficient for large-scale models.	**Tie**: Both offer high performance; differences are often negligible in practice.
Community and Ecosystem	Rapidly growing community; strong in academia; rich ecosystem with libraries like TorchVision and integration with Hugging Face.	Large and established community; extensive ecosystem with tools like TensorBoard and TFX; widely used in industry.	**Depends**: PyTorch excels in research community; TensorFlow has a broader industry ecosystem.
High-Level APIs	Uses native modules like `torch.nn`; high-level interfaces provided by PyTorch Lightning and Fast.ai.	Integrates Keras (`tf.keras`) as the high-level API.	**TensorFlow Wins**: Keras provides a more established and user-friendly high-level API.
Mobile and Embedded Deployment	PyTorch Mobile enables deployment on iOS and Android; supports model optimization like quantization.	TensorFlow Lite provides robust support for mobile and embedded devices; TensorFlow.js for web deployment.	**TensorFlow Wins**: More mature and versatile options for mobile and embedded deployment.
Preferred Domains	Favored in research and academia; excels in rapid prototyping; strong in computer vision and NLP tasks.	Widely used in industry and production; versatile across various domains.	**Depends**: PyTorch for research; TensorFlow for industry applications.
Learning Curve	Easier to learn due to intuitive design and dynamic execution.	Steeper learning curve; improved in TensorFlow 2.x but can still be complex.	**PyTorch Wins**: More beginner-friendly.
Interoperability	Seamless integration with Python libraries; supports exporting models to ONNX	Interoperable through TensorFlow Hub and SavedModel; supports ONNX	**PyTorch Wins**: Better integration with Python

Interoperability	Seamless integration with Python libraries; supports exporting models to ONNX format.	Interoperable through TensorFlow Hub and SavedModel; supports ONNX with some limitations.	**PyTorch Wins**: Better integration with Python ecosystem.
Customizability	High level of customization; easier to implement custom layers and operations.	Custom operations possible but can be complex; TensorFlow 2.x improves flexibility.	**PyTorch Wins**: Greater customizability and flexibility.
Deployment Tools	TorchServe for model serving; integrates with AWS, Azure, and Google Cloud.	TensorFlow Serving, TensorFlow Extended (TFX) for ML pipelines; strong cloud support.	**TensorFlow Wins**: More mature deployment tools and pipeline support.
Parallelism and Distributed Training	Supports distributed training with `torch.distributed`; enhanced by libraries like Horovod.	Extensive support with `tf.distribute.Strategy`; optimized for large-scale computing.	**TensorFlow Wins**: More advanced and user-friendly distributed training options.
Model Zoo and Pre-trained Models	Access via TorchVision, Hugging Face; strong community sharing.	TensorFlow Hub offers a wide range; extensive community models.	**Tie**: Both offer extensive pre-trained models; choice depends on specific needs.

PyTorch Core Modules

12 November 2024 19:16

Core PyTorch Modules

Module	Description
`torch`	The core module providing multidimensional arrays (tensors) and mathematical operations on them.
`torch.autograd`	Automatic differentiation engine that records operations on tensors to compute gradients for optimization.
`torch.nn`	Provides a neural networks library, including layers, activations, loss functions, and utilities to build deep learning models.
`torch.optim`	Contains optimization algorithms (optimizers) like SGD, Adam, and RMSprop used for training neural networks.
`torch.utils.data`	Utilities for data handling, including the `Dataset` and `DataLoader` classes for managing and loading datasets efficiently.
`torch.jit`	Supports Just-In-Time (JIT) compilation and TorchScript for optimizing models and enabling deployment without Python dependencies.
`torch.distributed`	Tools for distributed training across multiple GPUs and machines, facilitating parallel computation.
`torch.cuda`	Interfaces with NVIDIA CUDA to enable GPU acceleration for tensor computations and model training.
`torch.backends`	Contains settings and allows control over backend libraries like cuDNN, MKL, and others for performance tuning.
`torch.multiprocessing`	Utilities for parallelism using multiprocessing, similar to Python's `multiprocessing` module but with support for CUDA tensors.
`torch.quantization`	Tools for model quantization to reduce model size and improve inference speed, especially on edge devices.
`torch.onnx`	Supports exporting PyTorch models to the ONNX (Open Neural Network Exchange) format for interoperability with other frameworks and deployment.

PyTorch Domain Libraries

Library	Description
`torchvision`	Provides datasets, model architectures, and image transformations for computer vision tasks.
`torchtext`	Tools and datasets for natural language processing (NLP), including data preprocessing and vocabulary management.
`torchaudio`	Utilities for audio processing tasks, including I/O, transforms, and pre-trained models for speech recognition.
`torcharrow`	A library for accelerated data loading and preprocessing, especially for tabular and time series data (experimental).
`torchserve`	A PyTorch model serving library that makes it easy to deploy trained models at scale in production environments.
`pytorch_lightning`	A lightweight wrapper for PyTorch that simplifies the training loop and reduces boilerplate code, enabling scalable and reproducible models.

Popular PyTorch Ecosystem Libraries

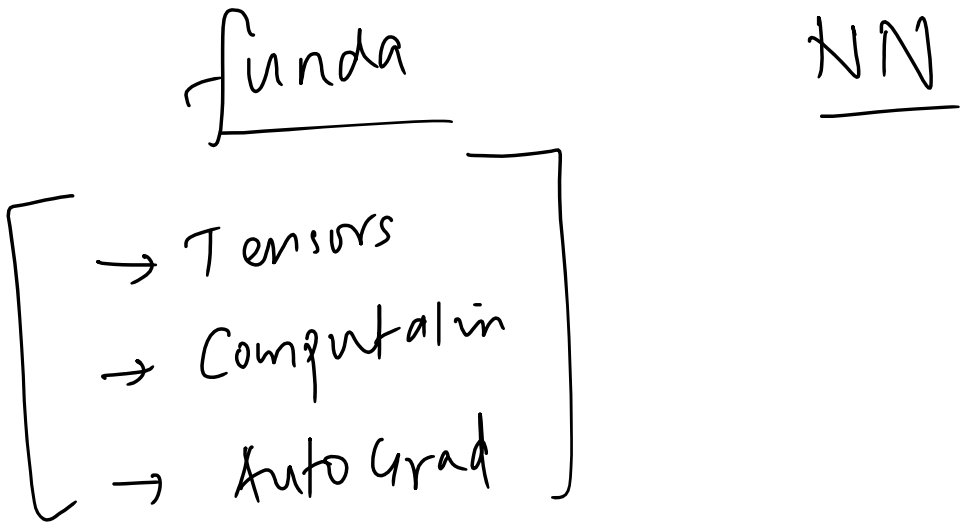
Library	Description
Hugging Face Transformers	Provides state-of-the-art pre-trained models for NLP tasks like text classification, translation, and question answering, built on PyTorch.
Fastai	High-level library that simplifies training fast and accurate neural nets using modern best practices, built on top of PyTorch.
PyTorch Geometric	Extension library for geometric deep learning, including graph neural networks and 3D data processing.
TorchMetrics	A modular metrics API for PyTorch, compatible with PyTorch Lightning and provides standardized implementations of many common metrics.
TorchElastic	Enables dynamic scaling of PyTorch distributed training jobs, allowing for elasticity in resource management.
Optuna	An automatic hyperparameter optimization software framework, integrating well with PyTorch for tuning models.

Catalyst	Provides high-level features for training neural networks, focusing on reproducibility and fast experimentation.
Ignite	High-level library to help with training neural networks in PyTorch, offering a lightweight engine for training and evaluating models.
AllenNLP	An NLP research library built on PyTorch, designed to support researchers in deep learning for NLP.
Skorch	A scikit-learn compatible wrapper for PyTorch that allows the use of PyTorch models with scikit-learn utilities and APIs.
PyTorch Forecasting	High-level library for time series forecasting, making it easy to build, train, and evaluate complex models.
TensorBoard for PyTorch	Allows visualization of training metrics, model graphs, and other useful data within TensorBoard for PyTorch models.

Who uses PyTorch

12 November 2024 19:16

Company	Products/Services Using PyTorch	Description of Usage
Meta Platforms (Facebook)	- Facebook App - Instagram - Meta AI Research Projects	Developed PyTorch and uses it extensively for computer vision, natural language processing, and AI research across its platforms.
Microsoft	- Azure Machine Learning - Bing Search - Office 365 Intelligent Features	Integrates PyTorch into Azure services for AI development; employs PyTorch in search relevance, productivity tools, and various AI applications.
Tesla	- Autopilot System - Full Self-Driving (FSD) Capability	Uses PyTorch for training deep neural networks in computer vision and perception tasks critical for autonomous driving systems.
OpenAI	- GPT Models - DALL·E - ChatGPT	Utilizes PyTorch for training large-scale language models and generative models in natural language processing and computer vision.
Uber	- Uber Ride-Hailing Platform - Uber Eats Recommendations - Pyro (Probabilistic Programming)	Employs PyTorch for demand forecasting, route optimization, and developed Pyro, a probabilistic programming language built on PyTorch.



Before starting

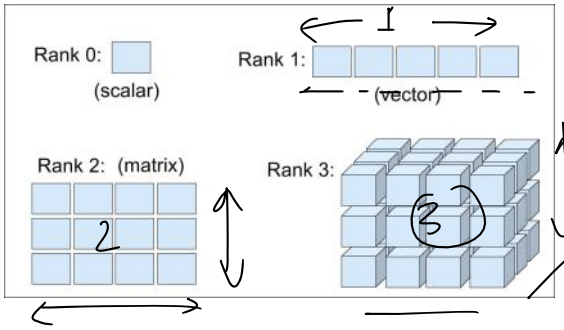
20 November 2024 17:55

1. Boring but useful
2. Knowledge of Deep learning
3. Very similar to NumPy
4. Lecture flow
5. Very close to PyTorch official documentation
6. Watch like a lecture

What are Tensors

20 November 2024 17:56

Tensor is a specialized multi-dimensional array designed for mathematical and computational efficiency.

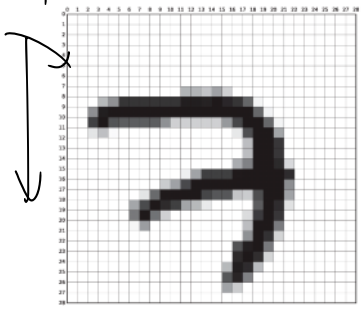


Real-World Examples

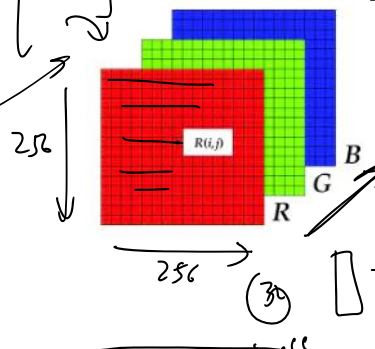
1. Scalars: 0-dimensional tensors (a single number)
   - Represents a single value, often used for simple metrics or constants.
   - Example:
     - Loss value: After a forward pass, the loss function computes a single scalar value indicating the difference between the predicted and actual outputs.
     - Example: 5.0 or -3.14

2. Vectors: 1-dimensional tensors (a list of numbers)
   - Represents a sequence or a collection of values.
   - Example:
     - Feature vector: In natural language processing, each word in a sentence may be represented as a 1D vector using embeddings.
     - Example: [0.12, -0.84, 0.33] (a word embedding vector from a pre-trained model like Word2Vec or Glove).

3. Matrices: 2-dimensional tensors (a 2D grid of numbers)
   - Represents tabular or grid-like data.
   - Example:
     - Grayscale images: A grayscale image can be represented as a 2D tensor, where each entry corresponds to the pixel intensity.
     - Example:
       [[0, 255, 128],
       [34, 90, 180]]

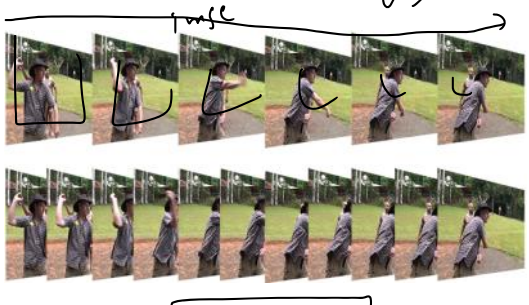


4. 3D Tensors: Coloured images
   - Adds a third dimension, often used for stacking data.
   - Example:
     - RGB Images: A single RGB image is represented as a 3D tensor (width × height × channels).
     - Examples:
       - RGB Image (e.g., 256x256): Shape [256, 256, 3]



5. 4D Tensors: Batches of RGB images
   - Adds the batch size as an additional dimension to 3D data.
   - Example:
     - Batches of RGB Images: A dataset of coloured images is represented as a 4D tensor (batch size × width × height × channels).
     - Example: A batch of 32 images, each of size 128x128 with 3 colour channels (RGB), would have shape [32, 128, 128, 3].

6. 5D Tensors: Video data
   - Adds a time dimension for data that changes over time (e.g., video frames).
   - Example:
     - Video Clips: Represented as a sequence of frames, where each frame is an RGB image.
     - Example: A batch of 10 video clips, each with 16 frames of size 64x64 and 3 channels (RGB), would have shape [10, 16, 64, 64, 3].

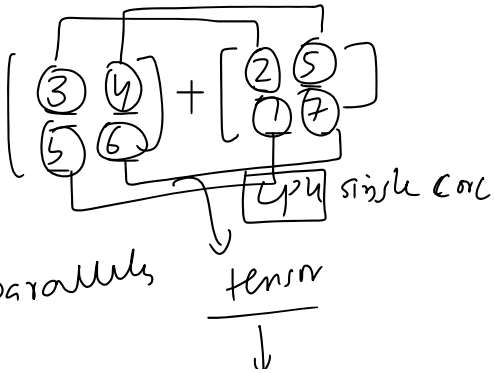


Why Are Tensors Useful?

1. Mathematical Operations
   - Tensors enable efficient mathematical computations (addition, multiplication, dot product, etc.) necessary for neural network operations.

2. Representation of Real-world Data
   - Data like images, audio, videos, and text can be represented as tensors:
     - Images: Represented as 3D tensors (width × height × channels).
     - Text: Tokenized and represented as 2D or 3D tensors (sequence length × embedding size).

3. Efficient Computations



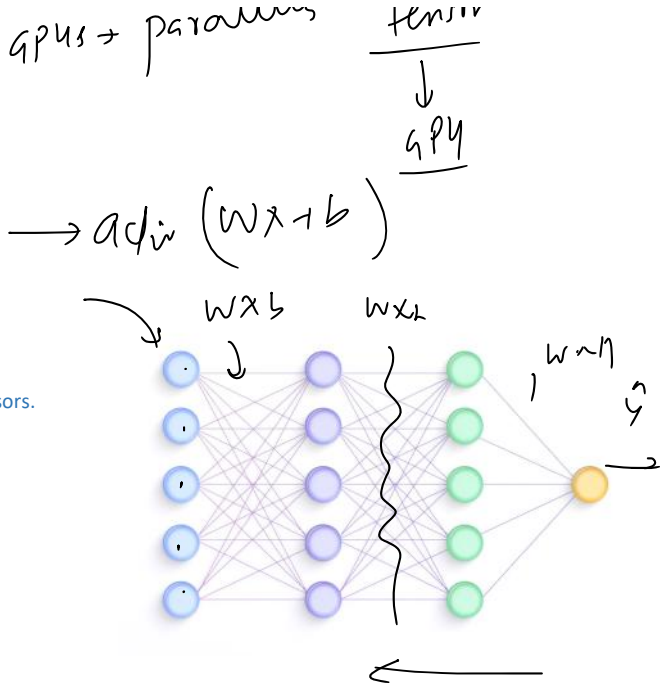
size).

3. Efficient Computations

- Tensors are optimized for hardware acceleration, allowing computations on GPUs or TPUs, which are crucial for training deep learning models.

Where Are Tensors Used in Deep Learning?

1. Data Storage
   - Training data (images, text, etc.) is stored in tensors.
2. Weights and Biases
   - The learnable parameters of a neural network (weights, biases) are stored as tensors.
3. Matrix Operations
   - Neural networks involve operations like matrix multiplication, dot products, and broadcasting—all performed using tensors.
4. Training Process
   - During forward passes, tensors flow through the network.
   - Gradients, represented as tensors, are calculated during the backward pass.



The Why

28 November 2024 16:52

$y = x^2 \rightarrow$ program $\rightarrow (x) \rightarrow x \rightarrow \frac{dy}{dx}$

$\downarrow$

$\frac{dy}{dx} = \boxed{2x} \rightarrow$ code

$x \rightarrow 2 \quad \frac{dy}{dx} = 4$

$x \rightarrow 3 \quad \frac{dy}{dx} = 6$

$y = x^2 \rightarrow \frac{dy}{dx} = 2x$

$z = \sin(y)$

$x \rightarrow \frac{dz}{dx}$ ② >> ①

$\frac{dz}{dy} = \cos y$

$\boxed{\frac{dz}{dx}} = \frac{dz}{dy}\frac{dy}{dx}$

$\rightarrow 2x\cos(y) = \boxed{2x\cos(x^2)}$

$$\begin{cases} y = x^2 \\ z = \sin y \\ u = e^z \end{cases}$$

$\frac{du}{dx} = \frac{du}{dz}\frac{dz}{dy}\frac{dy}{dx}$

$\downarrow$

code

[Nested function → complex → derivative → code difficult.]

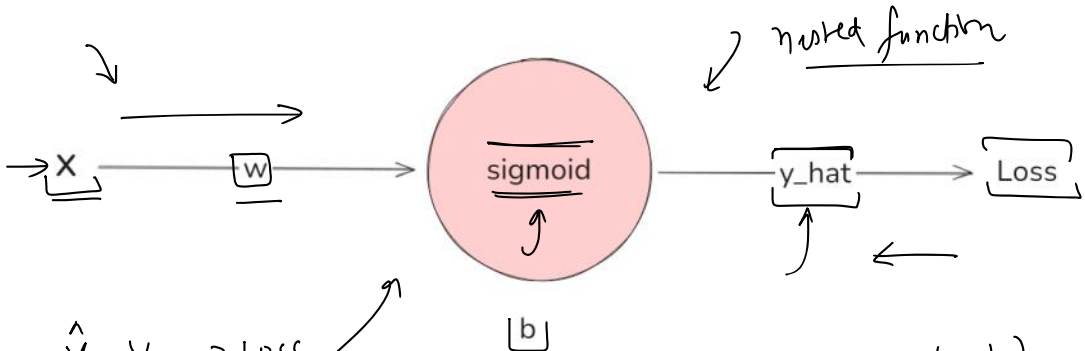
Nested function → derivative

$\downarrow$

[Deep learning]

$\frac{\partial L}{\partial w}$, $\frac{\partial L}{\partial b}$

cgpa	placed
9.11	1
8.9	1
7	0
6.56	1
4.56	0



nested function

$\hat{y}$ y → Loss

$L \rightarrow (w, b)$

Training process

1. Forward pass - Compute the output of the network given an input.
2. Calculate loss - Calculate the loss function to quantify the error.
3. Backward pass - Compute gradients of the loss with respect to the parameters.
4. Update gradients - Adjust the parameters using an optimization algorithm (e.g., gradient descent).

Forward Pass Computation

1. Linear Transformation:

$$z = w \cdot x + b$$

2. Activation (Sigmoid Function):

$$y_{pred} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

3. Loss Function (Binary Cross-Entropy Loss):

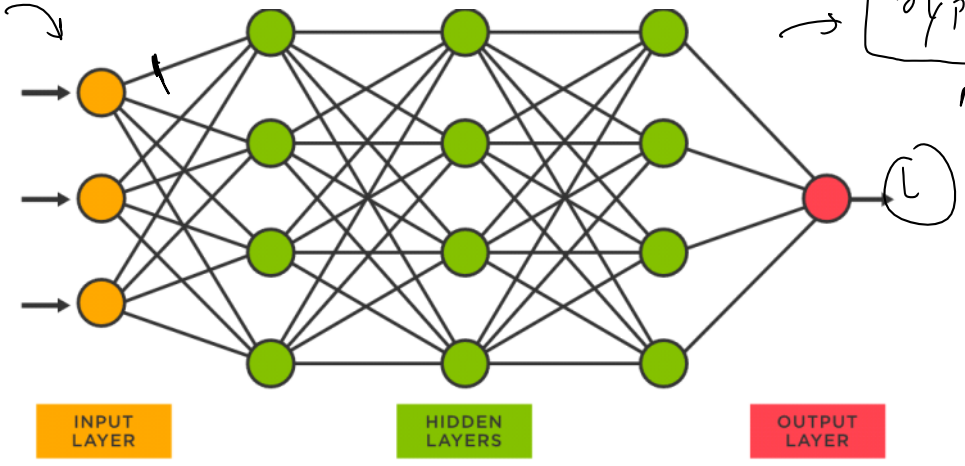
$$L = -[y_{target} \cdot \ln(y_{pred}) + (1 - y_{target}) \cdot \ln(1 - y_{pred})]$$

chain rule

$\frac{\partial L}{\partial b}$ $\frac{\partial L}{\partial w}$

$\frac{\partial L}{\partial y_{pred}} \times \frac{\partial y_{pred}}{\partial z} \frac{\partial z}{\partial w}$

$\frac{\partial z}{\partial b}$



$\frac{\partial L}{\partial w}$

Autograd

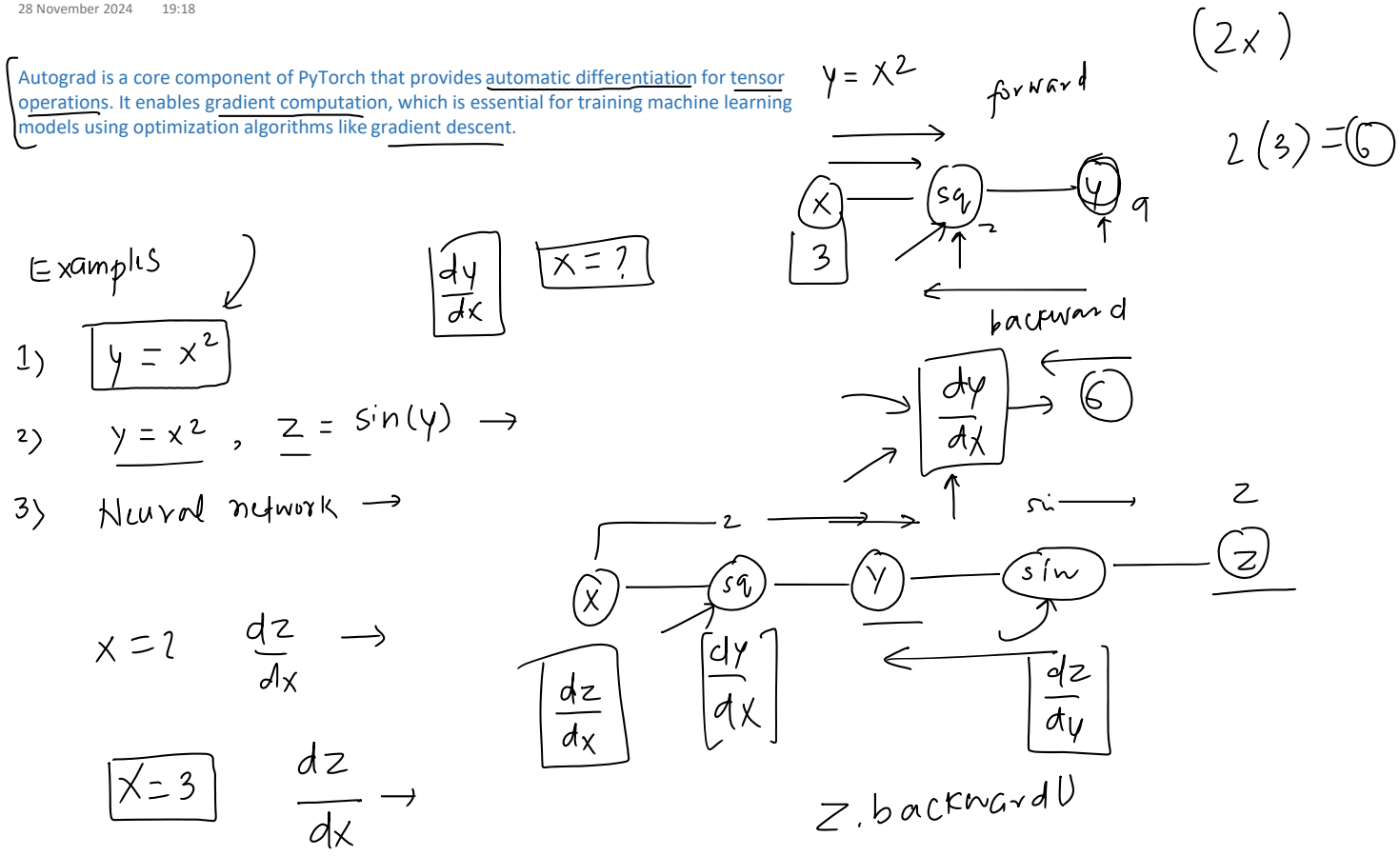
NN → nested function

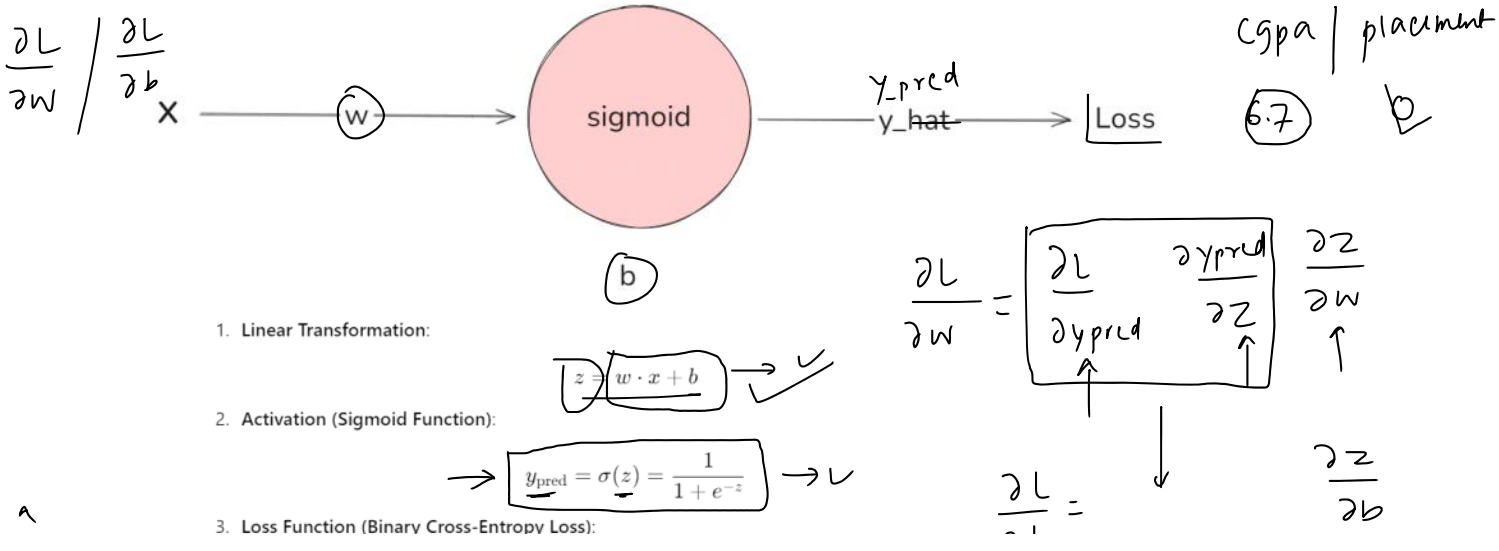
↳ derivatives manually impossible

What is Autograd

28 November 2024 19:18

Autograd is a core component of PyTorch that provides automatic differentiation for tensor operations. It enables gradient computation, which is essential for training machine learning models using optimization algorithms like gradient descent.





$$y_{\text{pred}} = \sigma(z) = \frac{1}{1+e^{-z}}$$

3. **Loss Function (Binary Cross-Entropy Loss):**

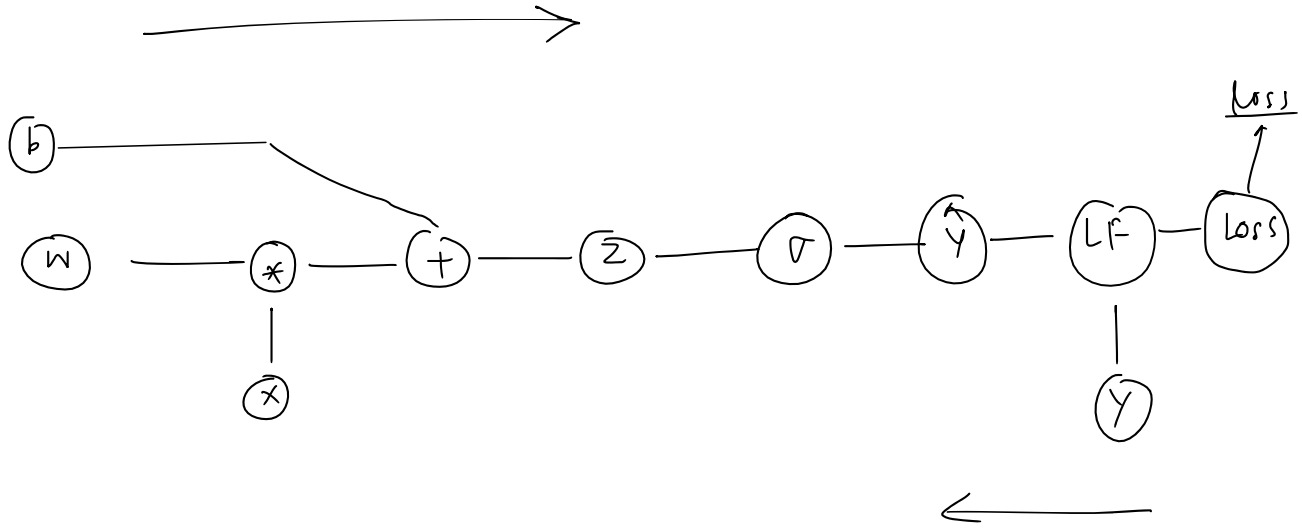
$$L = -[y_{\text{target}} \cdot \ln(y_{\text{pred}}) + (1 - y_{\text{target}}) \cdot \ln(1 - y_{\text{pred}})]$$

$\hat{y}$ y_pred

$\frac{\partial L}{\partial b} = \downarrow \frac{\partial z}{\partial b}$

$\frac{\partial L}{\partial y\_pred} = \frac{(\hat{y} - y)}{\hat{y}(1-\hat{y})}$ $\quad \frac{\partial \hat{y}}{\partial z} = \hat{y}(1-\hat{y})$ $\quad \frac{\partial z}{\partial w} = x$ $\quad \frac{\partial z}{\partial b} = 1$

$$\frac{\partial L}{\partial w} = (\hat{y} - y) * x \qquad \frac{\partial L}{\partial b} = (\hat{y} - y) * 1$$



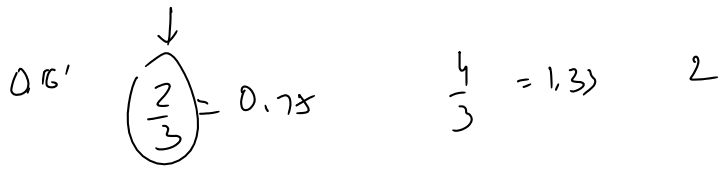
$$x = [x_1 \; x_2 \; x_3] = [1 \; 2 \; 3]$$

$y = x^2.\text{mean}()$ multivar func.

$$y = \frac{x_1^2 + x_2^2 + x_3^3}{3}$$

$y = f(x_1, x_2, x_3)$

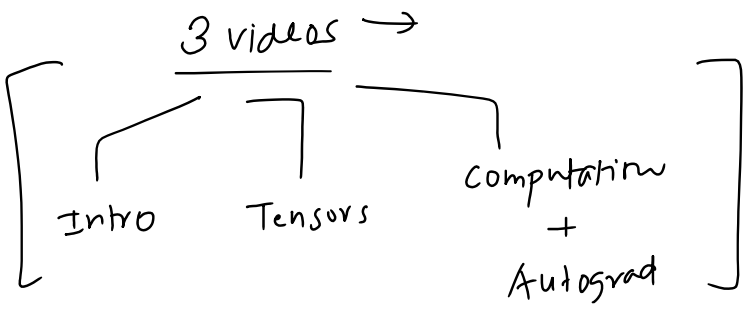
$\frac{dy}{dx_1} = \frac{2x_1}{3}$ $\quad \frac{\partial y}{dx_2} = \frac{2x_2}{3}$ $\quad \frac{\partial y}{\partial x_3} = \frac{2x_3}{3}$


0.6
2/3 = 0.75
4/3 = 1.33
2

Plan of Attack

02 December 2024 18:41

1. We will build a simple neural network ✓
2. Train it on a real world dataset ✓
3. Will mimic the PyTorch workflow ✓
4. Will have a lot of manual elements ✓
5. End result is not important ✓



Breast cancer

Dataset → NN

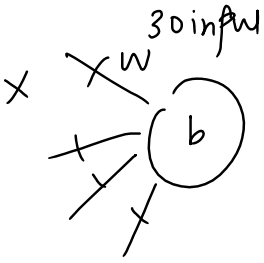
↓

Training pipeline → foundation

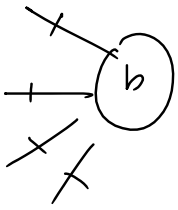
Code flow

02 December 2024 18:41

1. Load the dataset ✓
2. Basic preprocessing ✓
3. Training Process
   a. Create the model ✓
   b. Forward pass ✓
   c. Loss calculation ✓
   d. Backprop ✓
   e. Parameters update ✓
4. Model evaluation → accuracy



W, b



$z = wx + b$

$\sigma(z)$

$$w_{new} = w_{old} - lr \frac{\partial L}{\partial w}$$

$$b_{new} = b_{old} - \eta \frac{\partial L}{\partial b}$$

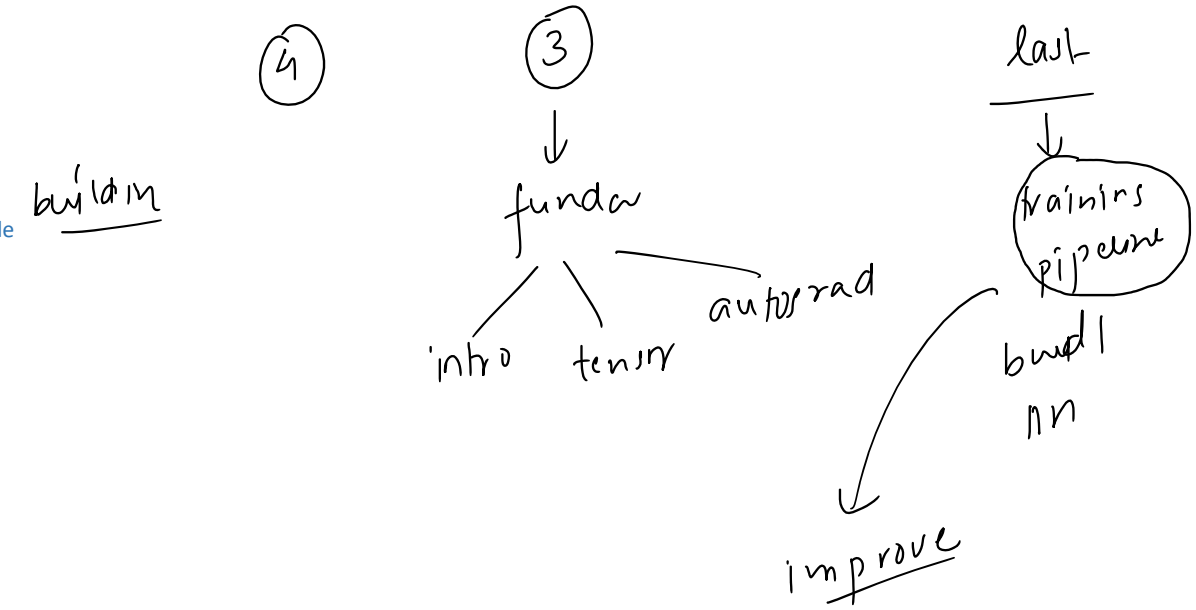
Next steps

02 December 2024 18:42

Plan of Action

03 December 2024 19:05

1. Revision
2. Improvements
3. The nn module
4. The torch.optim module



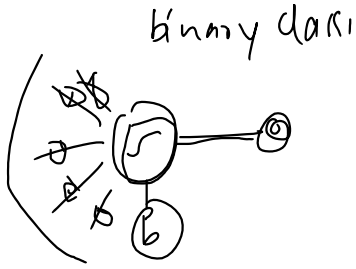
Improvements

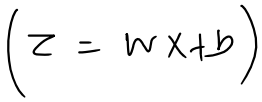
03 December 2024 19:11

1. Building the neural network using nn module
2. Using built-in activation function
3. Using built-in loss function
4. Using built-in optimizer

The nn module

03 December 2024 19:14

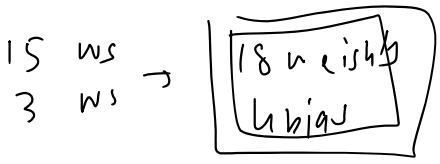


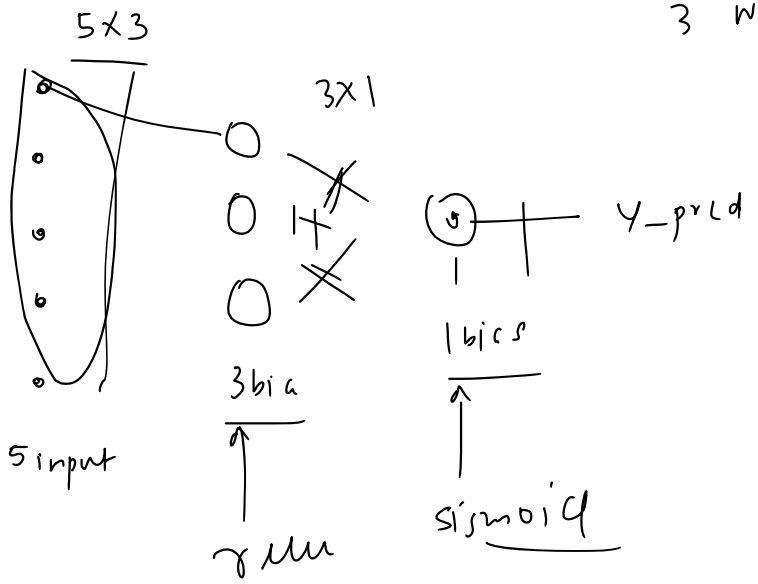


The torch.nn module in PyTorch is a core library that provides a wide array of classes and functions designed to help developers build neural networks efficiently and effectively. It abstracts the complexity of creating and training neural networks by offering pre-built layers, loss functions, activation functions, and other utilities, enabling you to focus on designing and experimenting with model architectures.

Key Components of torch.nn:

1. **Modules (Layers)**:
   - nn.Module: The base class for all neural network modules. Your custom models and layers should subclass this class.
   - Common Layers: Includes layers like nn.Linear (fully connected layer), nn.Conv2d (convolutional layer), nn.LSTM (recurrent layer), and many others.
2. **Activation Functions**:
   - Functions like nn.ReLU, nn.Sigmoid, and nn.Tanh introduce non-linearities to the model, allowing it to learn complex patterns.
3. **Loss Functions**:
   - Provides loss functions such as nn.CrossEntropyLoss, nn.MSELoss, and nn.NLLLoss to quantify the difference between the model's predictions and the actual targets.
4. **Container Modules**:
   - nn.Sequential: A sequential container to stack layers in order.
5. **Regularization and Dropout**:
   - Layers like nn.Dropout and nn.BatchNorm2d help prevent overfitting and improve the model's ability to generalize to new data.





Improved Code v1

03 December 2024 19:16

The torch.optim module

03 December 2024 19:16

.

torch.optim is a module in PyTorch that provides a variety of optimization algorithms used to update the parameters of your model during training.

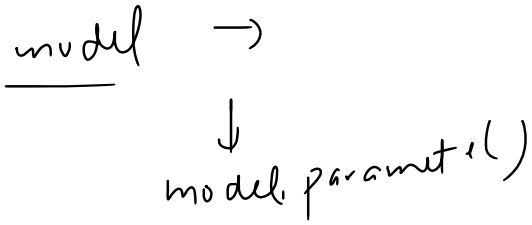
It includes common optimizers like Stochastic Gradient Descent (SGD), Adam, RMSprop, and more.

It handles weight updates efficiently, including additional features like learning rate scheduling and weight decay (regularization).

The model.parameters() method in PyTorch retrieves an iterator over all the trainable parameters (weights and biases) in a model. These parameters are instances of torch.nn.Parameter and include:

- Weights: The weight matrices of layers like nn.Linear, nn.Conv2d, etc.
- Biases: The bias terms of layers (if they exist).

The optimizer uses these parameters to compute gradients and update them during training.

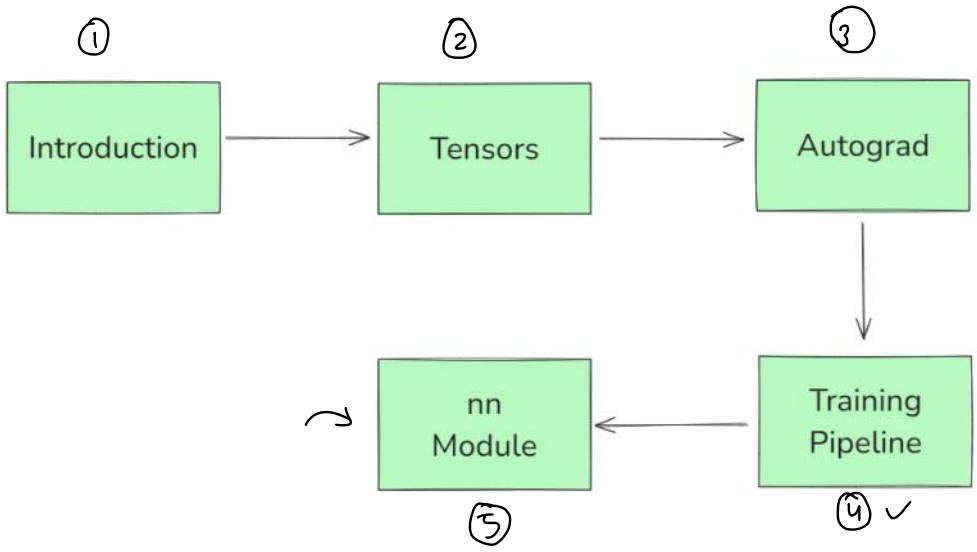


Improved Code v2

03 December 2024 19:16

Recap

10 December 2024 17:04



Problems

1. Memory inefficient
2. Better Convergence

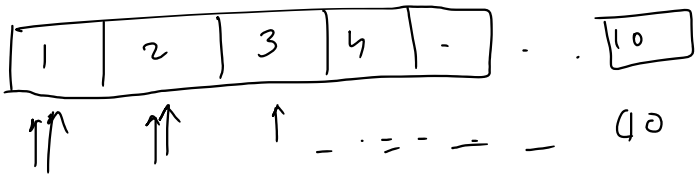
Solution - using batches of data to train the model

1000 rows
↳ 10 batches
↓
100 rows → forward / backward / grad / grad
100 rows →

Simple Solution

12 December 2024 08:55

$\frac{320}{32} = 10$




```python
batch_size = 32
epochs = 25
n_samples = len(X_train_tensor)

for epoch in range(epochs):
    # Simply loop over the dataset in chunks of `batch_size`
    for start_idx in range(0, n_samples, batch_size):
        end_idx = start_idx + batch_size
        X_batch = X_train_tensor[start_idx:end_idx]
        y_batch = y_train_tensor[start_idx:end_idx]

        # Forward pass
        y_pred = model(X_batch)
        loss = loss_function(y_pred, y_batch.view(-1, 1))

        # Update step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

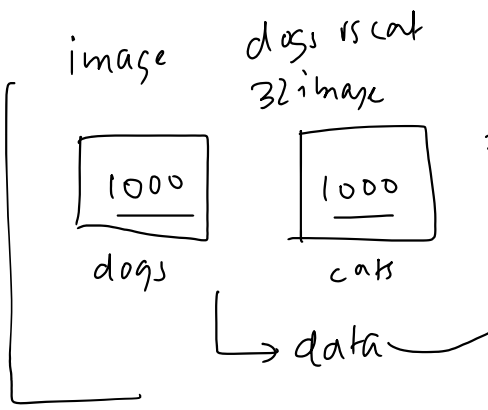
epoch 1 2 3 . .

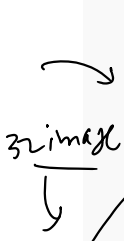
problems →

Problem with this approach!

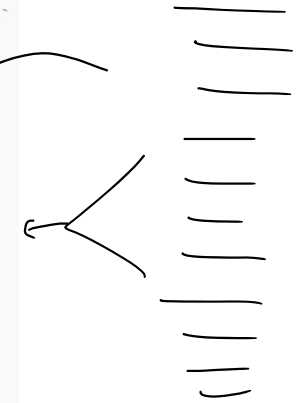
12 December 2024 16:22

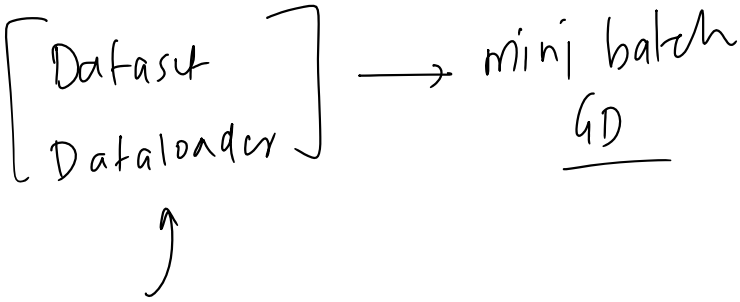
1. No standard interface for data
2. No easy way to apply transformations
3. Shuffling and sampling
4. Batch management & Parallelization






```python
batch_size = 32
epochs = 25
n_samples = len(X_train_tensor)

for epoch in range(epochs):
    # Simply loop over the dataset in chunks of `batch_size`
    for start_idx in range(0, n_samples, batch_size):
        end_idx = start_idx + batch_size
        X_batch = X_train_tensor[start_idx:end_idx]
        y_batch = y_train_tensor[start_idx:end_idx]

        # Forward pass
        y_pred = model(X_batch)
        loss = loss_function(y_pred, y_batch.view(-1, 1))

        # Update step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

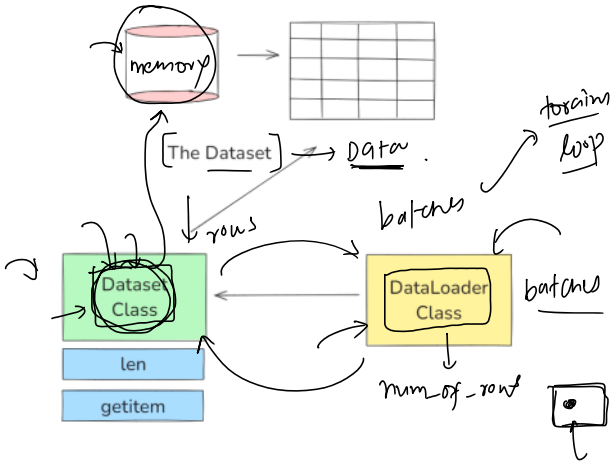




The Dataset and Dataloader Classes

12 December 2024 08:47

Dataset and DataLoader are core abstractions in PyTorch that decouple how you define your data from how you efficiently iterate over it in training loops.



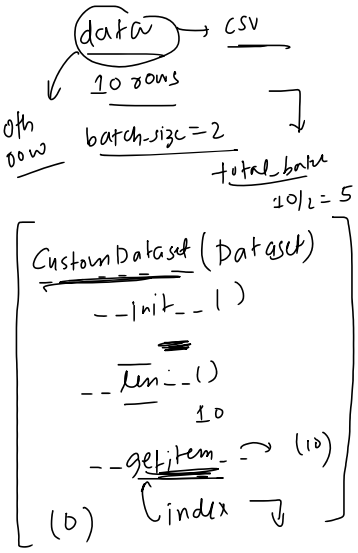
abstract class

Dataset Class

The Dataset class is essentially a blueprint. When you create a custom Dataset, you decide how data is loaded and returned.

It defines:

- __init__() which tells how data should be loaded.
- __len__() which returns the total number of samples.
- __getitem__(index) which returns the data (and label) at the given index.

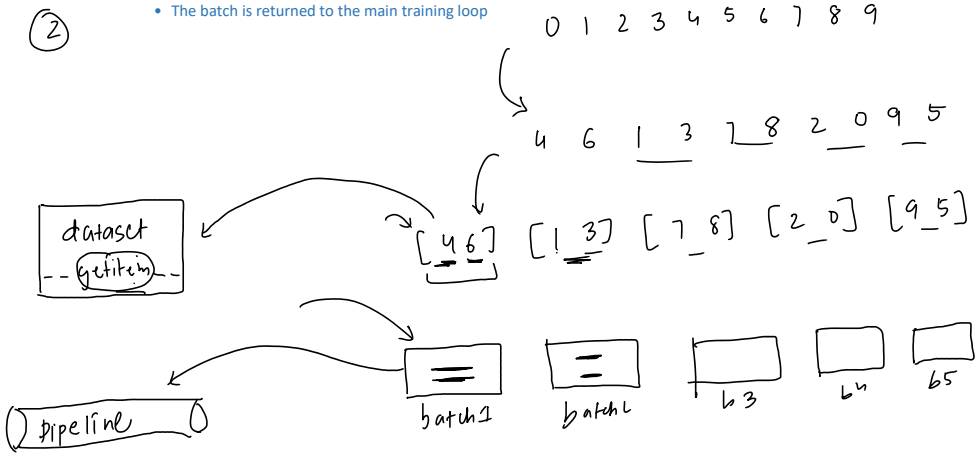


DataLoader Class

The DataLoader wraps a Dataset and handles batching, shuffling, and parallel loading for you.

DataLoader Control Flow:

- At the start of each epoch, the DataLoader (if shuffle=True) shuffles indices(using a sampler).
- It divides the indices into chunks of batch_size.
- for each index in the chunk, data samples are fetched from the Dataset object
- The samples are then collected and combined into a batch (using collate_fn)
- The batch is returned to the main training loop



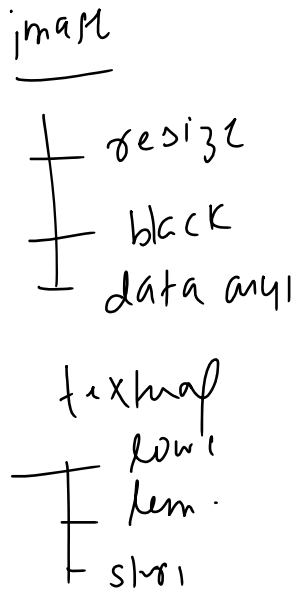
A Simple Example

12 December 2024 08:48

A note about data transformations

12 December 2024 17:51

```python
class CustomDataset(Dataset):

  def __init__(self, features, labels):

    self.features = features
    self.labels = labels

  def __len__(self):

    return self.features.shape[0]

  def __getitem__(self, index):

    return self.features[index], self.labels[index]
```



A note about Parallelization

12 December 2024 18:00

Imagine the entire data loading and training process for one epoch with num_workers=4:

Assumptions:

- Total samples: 10,000
- Batch size: 32
- Workers (num_workers): 4
- Approximately 312 full batches per epoch (10000 / 32 ≈ 312).

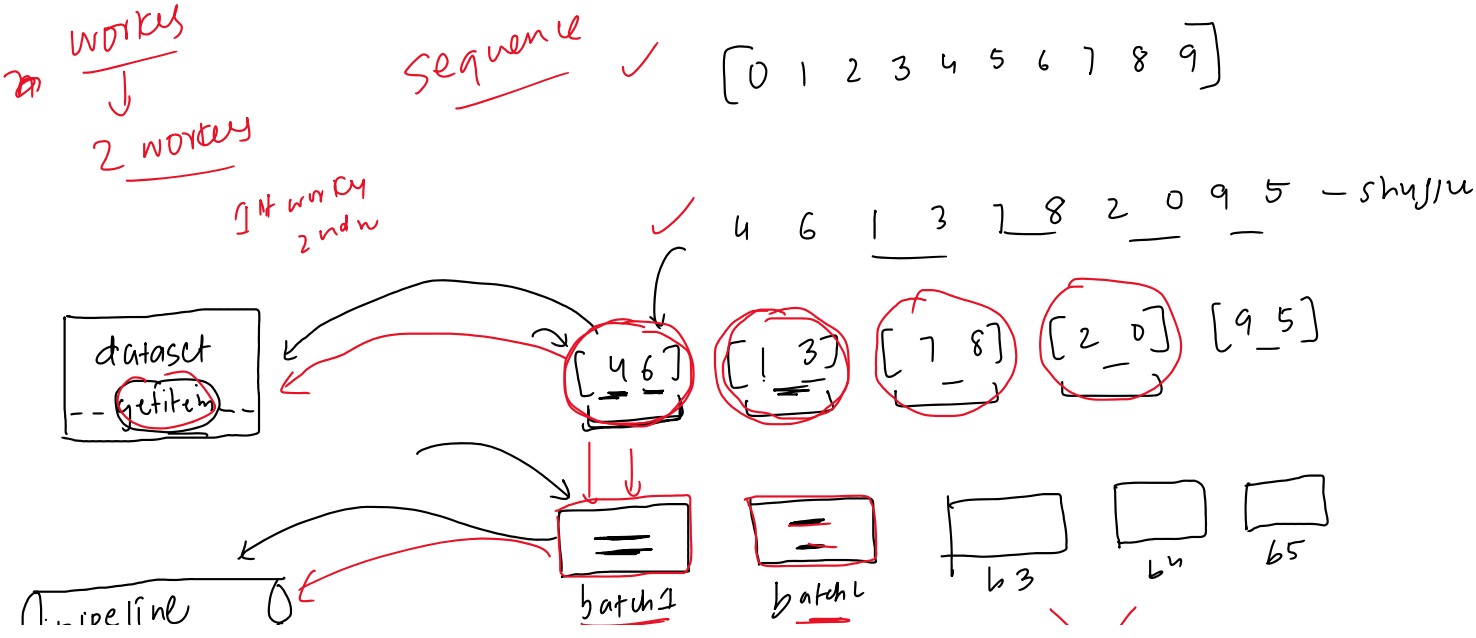
Workflow:

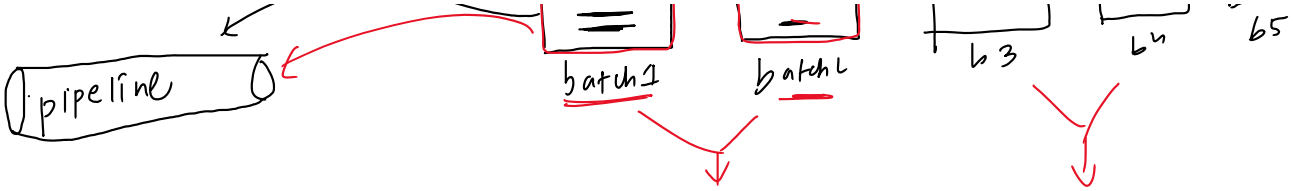
1. **Sampler and Batch Creation (Main Process):**
   Before training starts for the epoch, the DataLoader's sampler generates a shuffled list of all 10,000 indices. These are then grouped into 312 batches of 32 indices each. All these batches are queued up, ready to be fetched by workers.
2. **Parallel Data Loading (Workers):**
   - At the start of the training epoch, you run a training loop like:

     python
     Copy code
     for batch_data, batch_labels in dataloader:
       # Training logic
   - Under the hood, as soon as you start iterating over dataloader, it dispatches the first four batches of indices to the four workers:
     - Worker #1 loads batch 1 (indices [batch_1_indices])
     - Worker #2 loads batch 2 (indices [batch_2_indices])
     - Worker #3 loads batch 3 (indices [batch_3_indices])
     - Worker #4 loads batch 4 (indices [batch_4_indices])

       Each worker:
   - Fetches the corresponding samples by calling __getitem__ on the dataset for each index in that batch.
   - Applies any defined transforms and passes the samples through collate_fn to form a single batch tensor.
3. **First Batch Returned to Main Process:**
   - Whichever worker finishes first sends its fully prepared batch (e.g., batch 1) back to the main process.
   - As soon as the main process gets this first prepared batch, it yields it to your training loop, so your codefor batch_data, batch_labels in dataloader: receives (batch_data, batch_labels) for the first batch.
4. **Model Training on the Main Process:**
   - While you are now performing the forward pass, computing loss, and doing backpropagation on the first batch, the other three workers are still preparing their batches in parallel.
   - By the time you finish updating your model parameters for the first batch, the DataLoader likely has the second, third, or even more batches ready to go (depending on processing speed and hardware).
5. **Continuous Processing:**
   - As soon as a worker finishes its batch, it grabs the next batch of indices from the queue.
   - For example, after Worker #1 finishes with batch 1, it immediately starts on batch 5. After Worker #2 finishes batch 2, it takes batch 6, and so forth.
   - This creates a pipeline effect: at any given moment, up to 4 batches are being prepared concurrently.
6. **Loop Progression:**
   - Your training loop simply sees:

     python
     Copy code
     for batch_data, batch_labels in dataloader:
       # forward pass
       # loss computation
       # backward pass
       # optimizer step
   - Each iteration, it gets a new, ready-to-use batch without long I/O waits, because the workers have been pre-loading and processing data in parallel.
7. **End of the Epoch:**
   - After ~312 iterations, all batches have been processed. All indices have been consumed, so the DataLoader has no more batches to yield.
   - The epoch ends. If shuffle=True, on the next epoch, the sampler reshuffles indices, and the whole process repeats with workers again loading data in parallel.




pipeline
batch 1
batch 2
b 3
b 4
65

A note about samplers

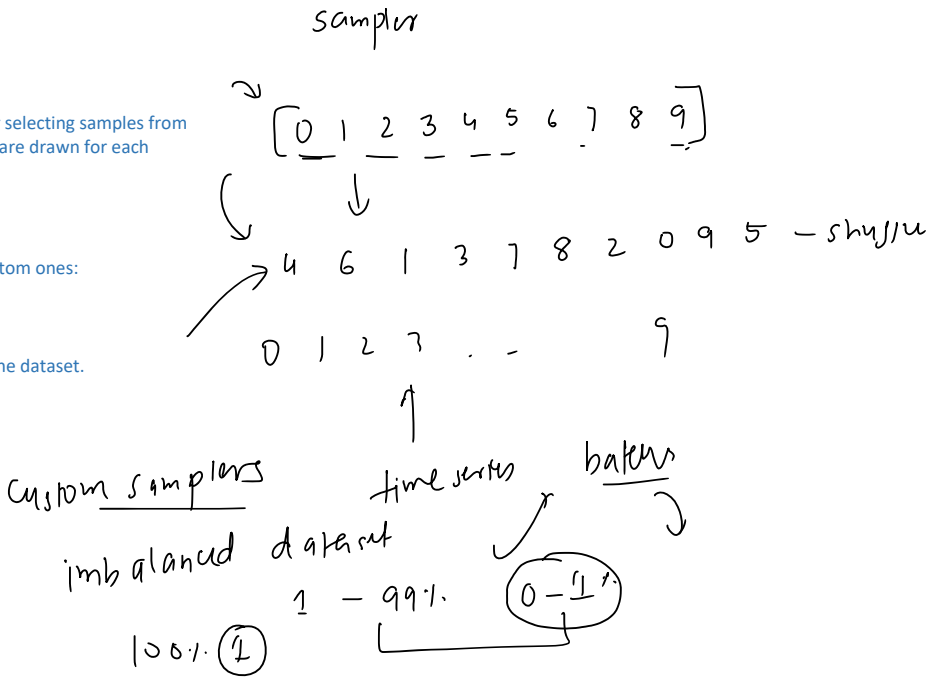
12 December 2024 17:51

In PyTorch, the sampler in the DataLoader determines the strategy for selecting samples from the dataset during data loading. It controls how indices of the dataset are drawn for each batch.

Types of Samplers

PyTorch provides several predefined samplers, and you can create custom ones:

1. SequentialSampler:
   - Samples elements sequentially, in the order they appear in the dataset.
   - Default when shuffle=False.
2. RandomSampler:
   - Samples elements randomly without replacement.
   - Default when shuffle=True.



A note about collate_fn

12 December 2024 17:51

The collate_fn in PyTorch's DataLoader is a function that specifies how to combine a list of samples from a dataset into a single batch. By default, the DataLoader uses a simple batch collation mechanism, but collate_fn allows you to customize how the data should be processed and batched.

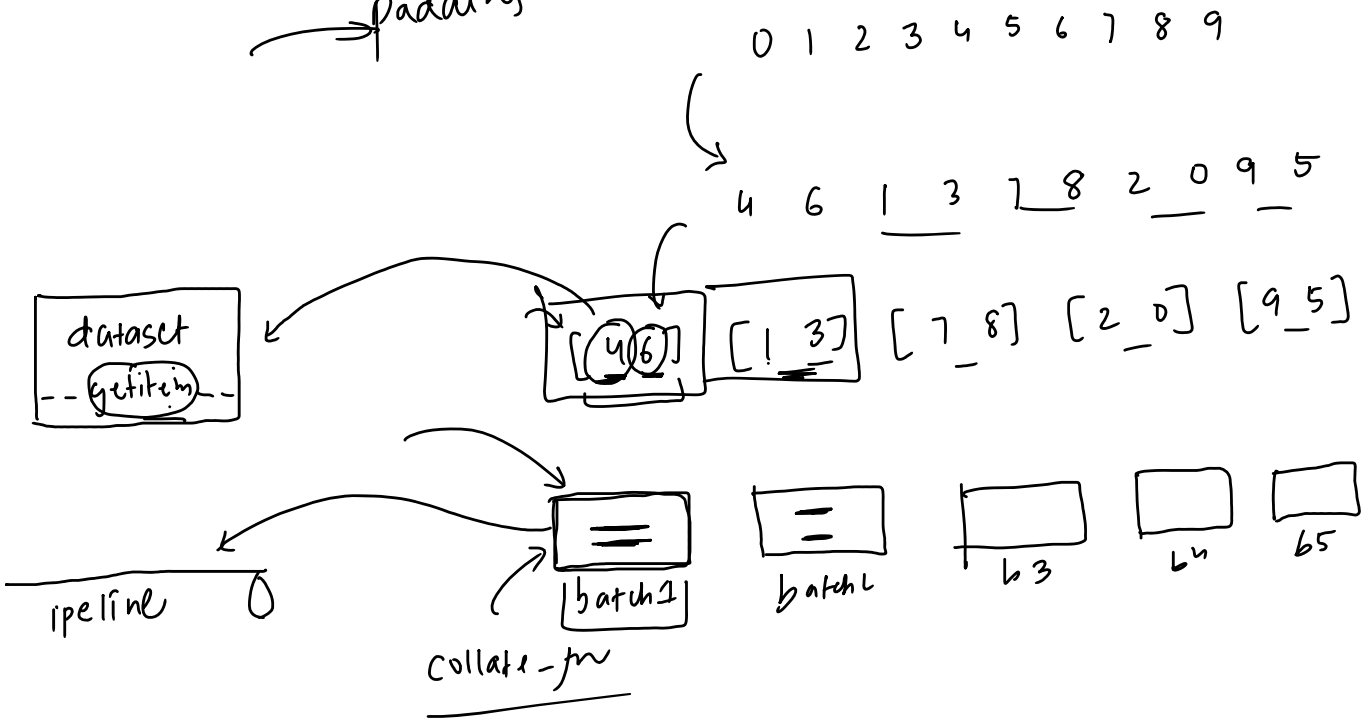
batch_size = 2

text

Sentence	Tokenized (Integer IDs)	Label
"I love coding"	[1, 2, 3]	0
"Deep learning rocks"	[4, 5, 6]	1
"Transformers are fun"	[7, 8, 9, 10]	1
"Hello world"	[11, 12] 0 0	0

padding

collate_fn

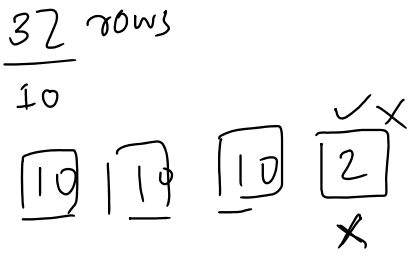


DataLoader Important Parameters

12 December 2024 18:01

The DataLoader class in PyTorch comes with several parameters that allow you to customize how data is loaded, batched, and preprocessed. Some of the most commonly used and important parameters include:

1. dataset (mandatory):
   - The Dataset from which the DataLoader will pull data.
   - Must be a subclass of torch.utils.data.Dataset that implements __getitem__ and __len__.
2. batch_size:
   - How many samples per batch to load.
   - Default is 1.
   - Larger batch sizes can speed up training on GPUs but require more memory.
3. shuffle:
   - If True, the DataLoader will shuffle the dataset indices each epoch.
   - Helpful to avoid the model becoming too dependent on the order of samples.
4. num_workers:
   - The number of worker processes used to load data in parallel.
   - Setting num_workers > 0 can speed up data loading by leveraging multiple CPU cores, especially if I/O or preprocessing is a bottleneck.
5. pin_memory: — spw
   - If True, the DataLoader will copy tensors into pinned (page-locked) memory before returning them.
   - This can improve GPU transfer speed and thus overall training throughput, particularly on CUDA systems.
6. drop_last
   - If True, the DataLoader will drop the last incomplete batch if the total number of samples is not divisible by the batch size.
   - Useful when exact batch sizes are required (for example, in some batch normalization scenarios).
7. collate_fn:
   - A callable that processes a list of samples into a batch (the default simply stacks tensors).
   - Custom collate_fn can handle variable-length sequences, perform custom batching logic, or handle complex data structures.
8. sampler:
   - sampler defines the strategy for drawing samples (e.g., for handling imbalanced classes, or custom sampling strategies).
   - batch_sampler works at the batch level, controlling how batches are formed.
   - Typically, you don't need to specify these if you are using batch_size and shuffle. However, they provide lower-level control if you have advanced requirements.



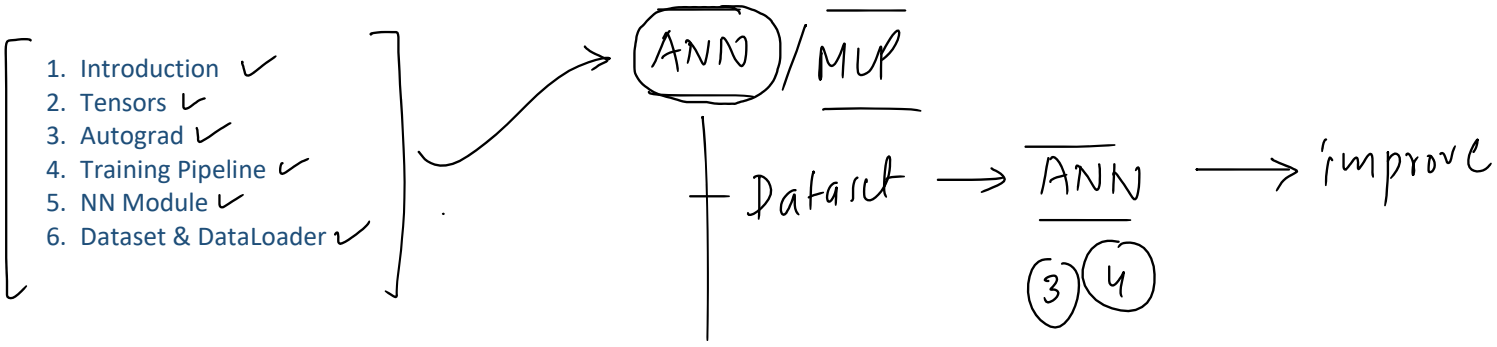
Improving our existing code

12 December 2024 08:49

Recap & Plan of Action

24 December 2024 08:37

1. Introduction ✓
2. Tensors ✓
3. Autograd ✓
4. Training Pipeline ✓
5. NN Module ✓
6. Dataset & DataLoader ✓



The Dataset

24 December 2024 08:38

kaggle → fasim MNIST

Code

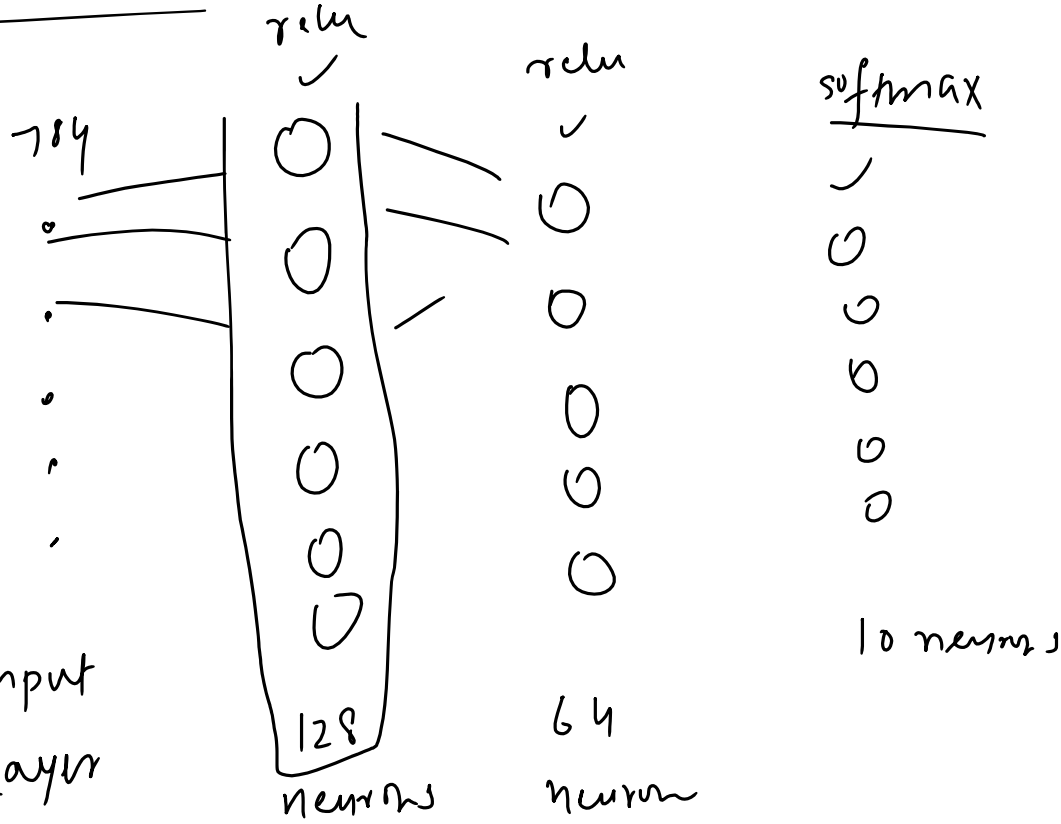
24 December 2024 08:38

70,000 images → cpu

gpu

↓

60000 images → ANN build



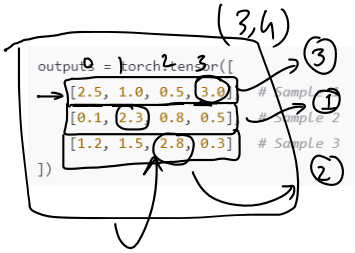
3> workflow

- → Dataloader objects
- → training loop
- → evaluation

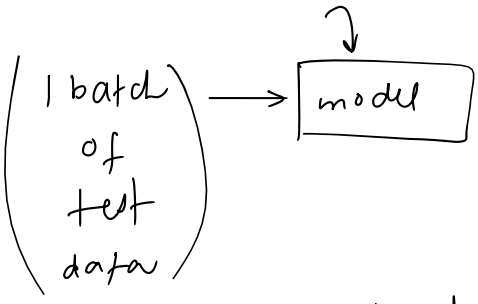
Improvements

24 December 2024 08:38

24 December 2024 08:38

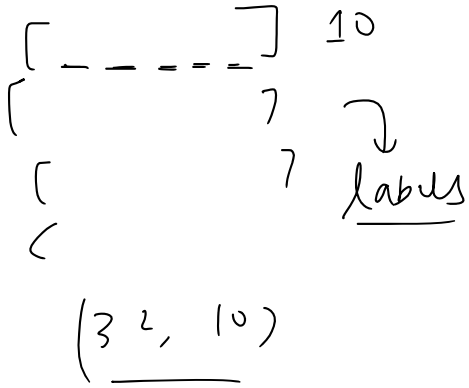


[3, 1, 2]



[32 images] → 1 output

↑ ↘ label

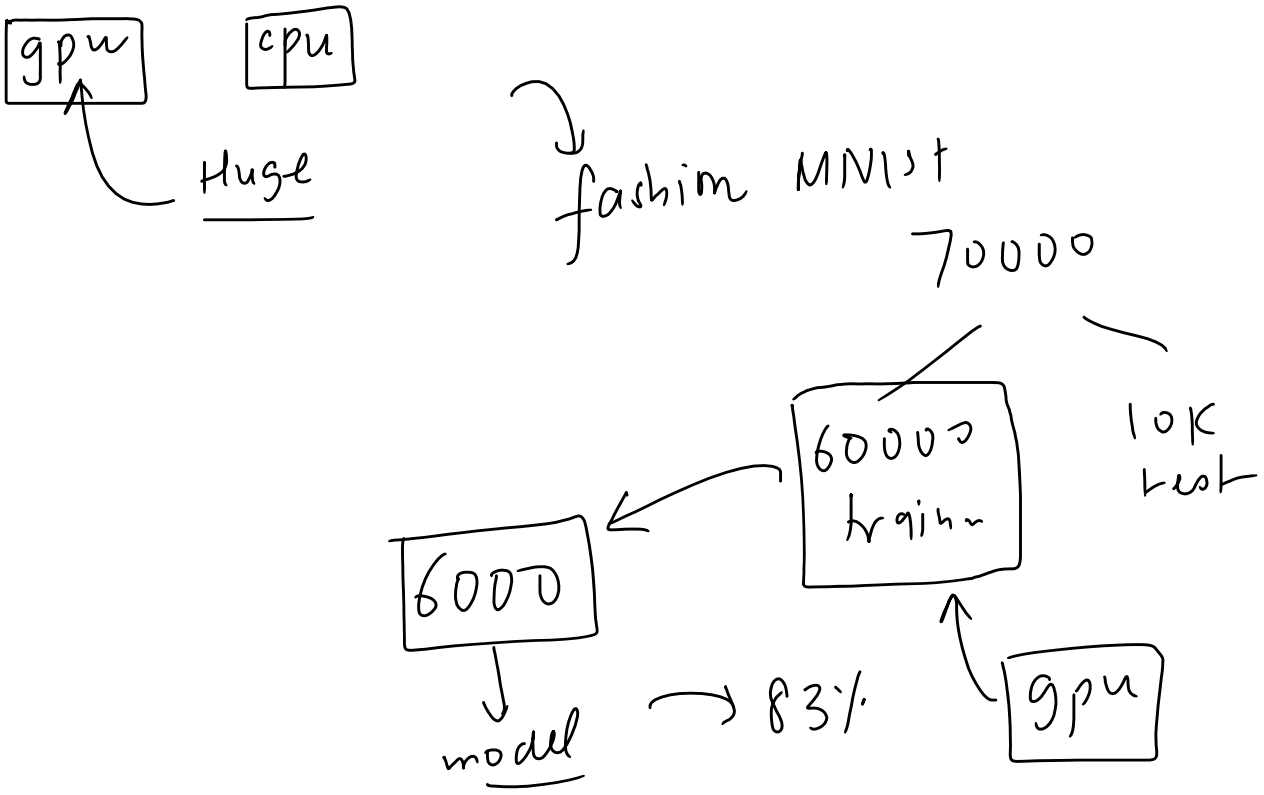


(32, 10)

- → use full dataset (6000 image)
  - ↳ GPU →
- → optimizer, lr, epochs, weight_int.
- → hPT

Recap

24 December 2024 17:37



Steps for GPU Training

24 December 2024 17:37

1. Check GPU Availability

Before starting, verify if a GPU is available. If available, select it; otherwise, use the CPU.

```python
import torch

# Check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

2. Move the Model to GPU

Move your model to the selected device (`cuda` for GPU or `cpu`) so that all computations occur on the same device.

```python
model = MyModel()  # Replace with your model
model = model.to(device)  # Move model to GPU
```

3. Modify the Training Loop by Moving Data to GPU

Ensure that each batch of data (features and labels) is moved to the GPU before processing. This ensures that both the model and data are on the same device.

```
for batch_features, batch_labels in train_loader:
    # Move data to GPU
    batch_features, batch_labels = batch_features.to(device), batch_labels.to(device)
```

4. Modify the Evaluation Loop by Moving Data to GPU

Similarly, ensure test data is moved to the GPU during evaluation. Disable gradient calculations using `torch.no_grad()` for efficiency.

```
with torch.no_grad():
    for batch_features, batch_labels in test_loader:
        # Move data to GPU
        batch_features, batch_labels = batch_features.to(device), batch_labels.to(device)
```

5. Optimize the GPU Usage

To make the best use of GPU resources, apply the following optimizations:

a. Use Larger Batch Sizes

Larger batch sizes can better utilize GPU memory and reduce computation time per epoch (if memory allows).

```python
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False, pin_memory=True)
```

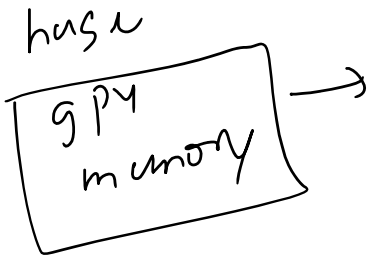
```
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False, pin_memory=True)
```

b. Enable DataLoader Pinning

Use `pin_memory=True` in `DataLoader` to speed up data transfer from CPU to GPU.

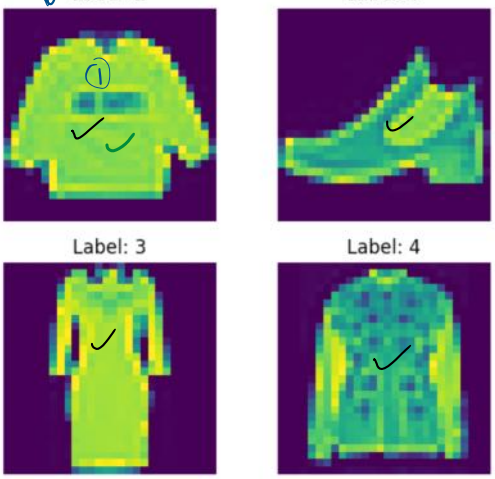
python Copy code

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, pin_memory=True)
```



Recap

02 January 2025 16:49



flip? rotate

fashion MNIST

→ pytorch

↓

ANN

60000 images

stage 1

cpu → train

5000 images → 82-83%

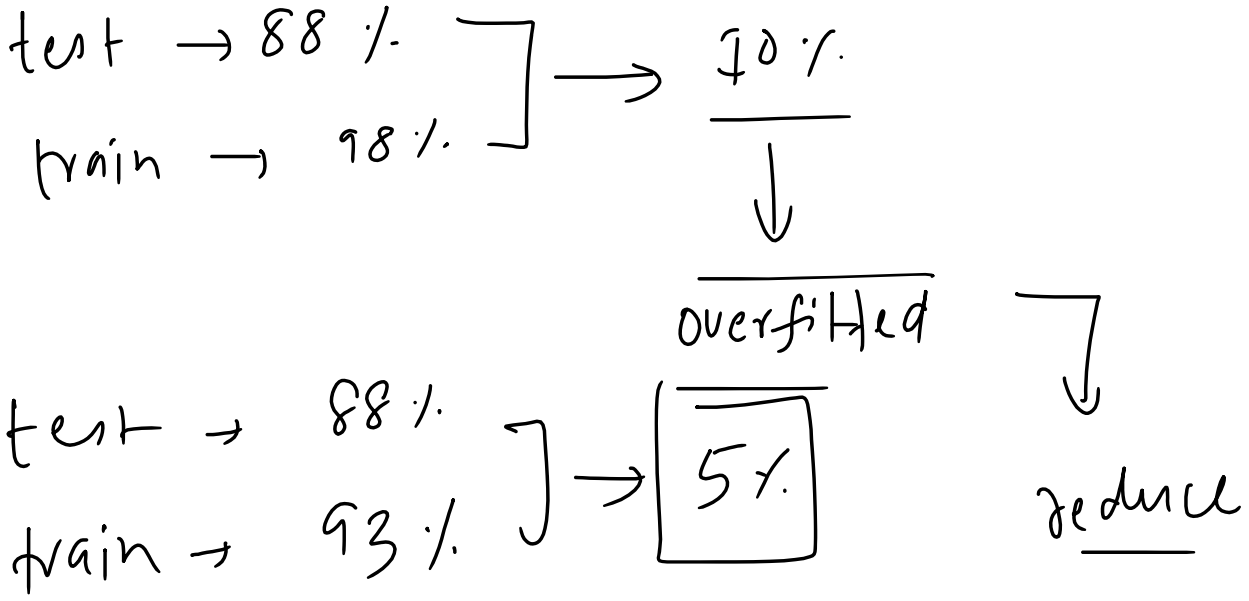
stage 2

gpu - train

60000 → 88

The Problem

02 January 2025 16:50

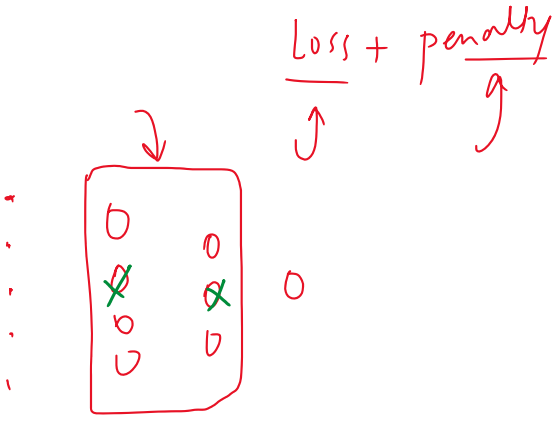


Solution

02 January 2025 16:50

60000

1. Adding more data ✓ ✗
2. Reducing the complexity of NN architecture ✗
3. Regularization ✓
4. Dropouts ✓
5. Data Augmentation → CNN ✗
6. Batch Normalization ✓
7. Early Stopping ✗



optimize

→ Regularization

→ Dropouts

→ Batch Norm

→ train → train / test → gap reduce

Dropout

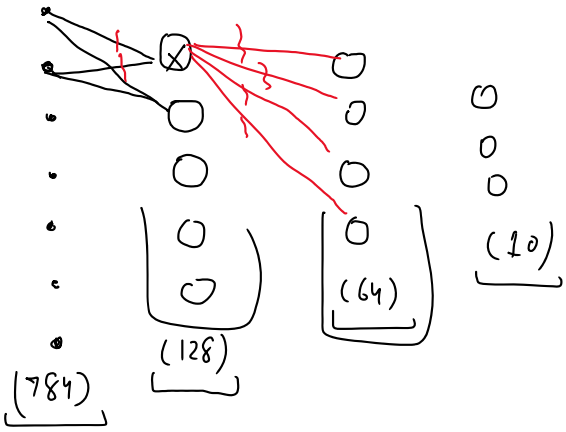
03 January 2025 18:22

[p = 0.5] 50%

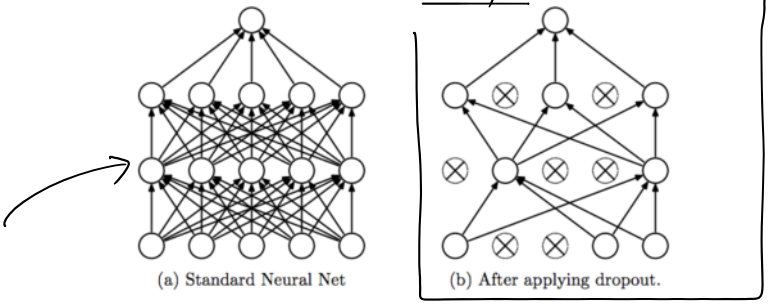
0.3

1. Applied to the hidden layers
2. Applied after the ReLU activation function
3. Randomly turns off p% neurons in the hidden layer during each forward pass
4. This has a regularization effect ✓
5. During evaluation dropout is not used

each forward pass



simplify

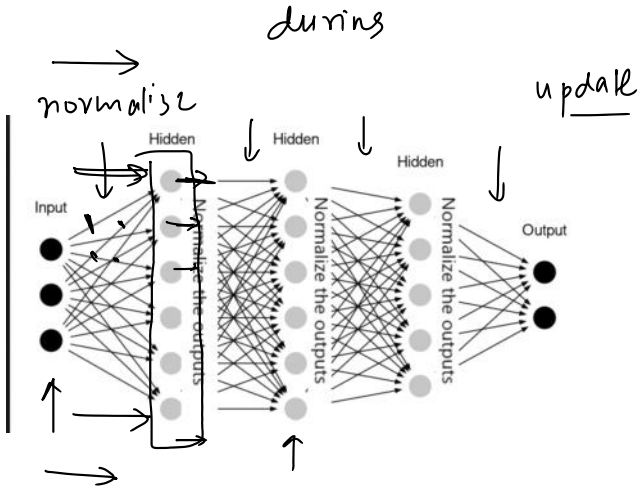


(a) Standard Neural Net

(b) After applying dropout.

Batch Norm

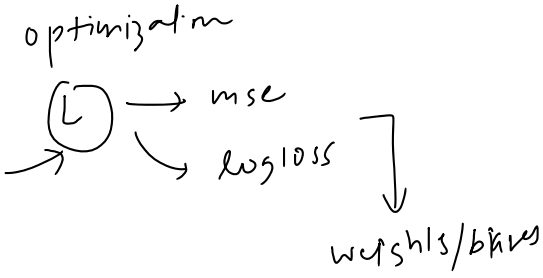
03 January 2025 18:40



- Applied to Hidden Layers:
  - Typically applied to the hidden layers of a neural network, but not to the output layer.
- Applied After Linear Layers and Before Activation Functions:
  - Normalizes the output of the preceding layer (e.g., after nn.Linear) and is usually followed by an activation function (e.g., ReLU).
- Normalizes Activations:
  - Computes the mean and variance of the activations within a mini-batch and uses these statistics to normalize the activations.
- Includes Learnable Parameters:
  - Introduces two learnable parameters, gamma (scaling) and beta (shifting), which allow the network to adjust the normalized outputs.
- Improves Training Stability:
  - Reduces internal covariate shift, stabilizing the training process and allowing the use of higher learning rates.
- Regularization Effect:
  - Introduces some regularization because the statistics are computed over a mini-batch, adding noise to the training process.
- Consistent During Evaluation:
  - During evaluation, BatchNorm uses the running mean and variance accumulated during training, rather than recomputing them from the mini-batch.

L2 Regularization

03 January 2025 18:58



Applied to Model Weights:

- Regularization is applied to the weights of the model to penalize large values and encourage smaller, more generalizable weights.

Introduced via Loss Function or Optimizer:

- Adds a penalty term λ∑wi2 to the loss function in L2 regularization.

$$\text{Loss}_{\text{reg}} = \text{Loss}_{\text{original}} + \lambda \sum w_i^2$$

$\lambda \left(w_1^2 + w_2^2 + w_3^2 + w_4^2 \right)$

- In weight decay, directly modifies the gradient update rule to include λwi, effectively shrinking weights during training.

$$w \leftarrow w - \eta(\nabla \text{Loss} + \lambda w)$$

weight_decay

Penalizes Large Weights:

- Encourages the network to distribute learning across multiple parameters, avoiding reliance on a few large weights.

Reduces Overfitting:

- Helps the model generalize better to unseen data by discouraging overly complex representations.

Controlled by a Hyperparameter:

- A regularization coefficient (λ often set via weight_decay in optimizers) controls the strength of the penalty. Larger values lead to stronger regularization.

No Effect on Bias Terms:

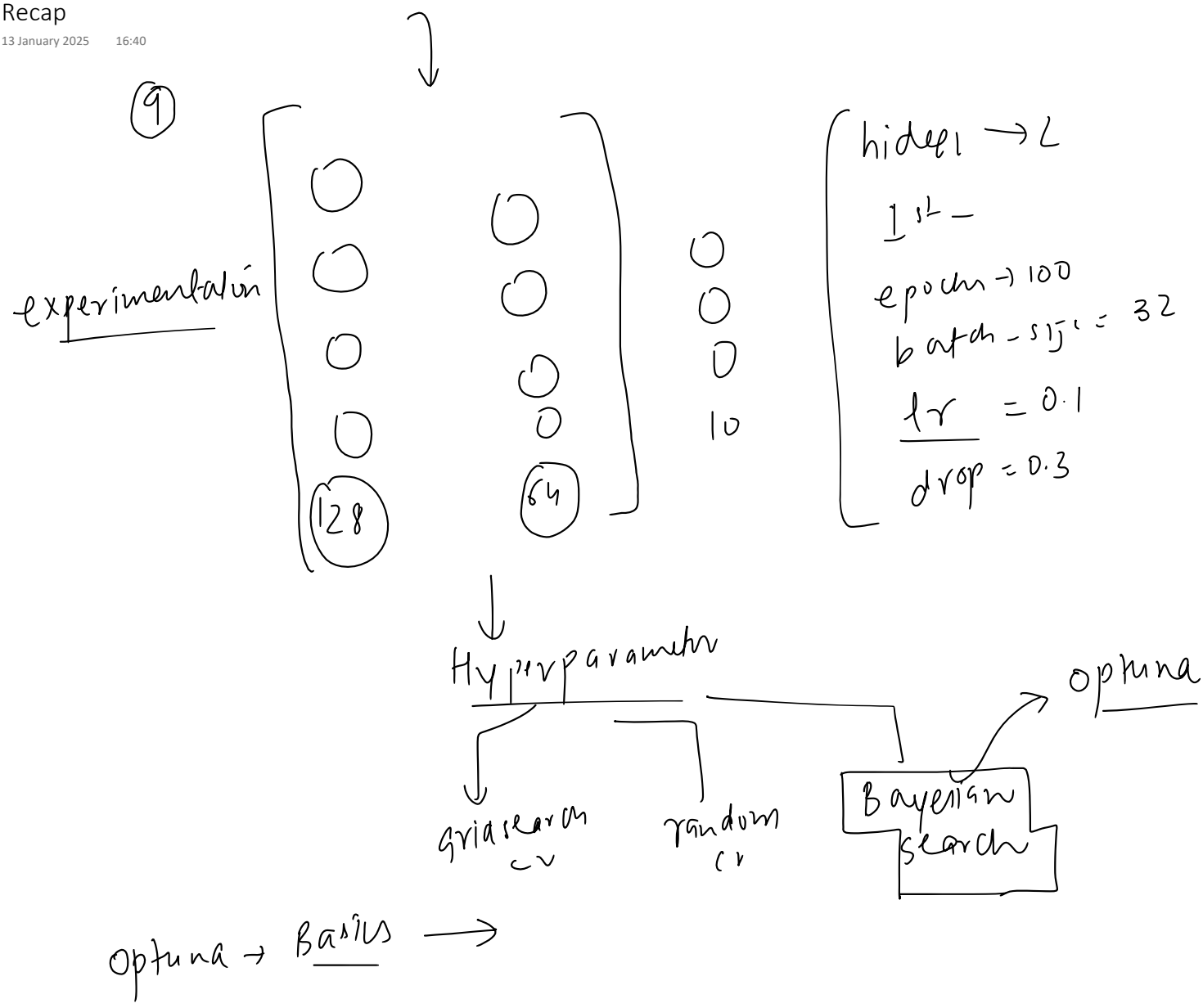
- Regularization is typically applied only to weights, not biases, as biases don't directly affect model complexity.

Active During Training:

- Regularization affects weight updates only during training. It does not explicitly influence the model during inference.

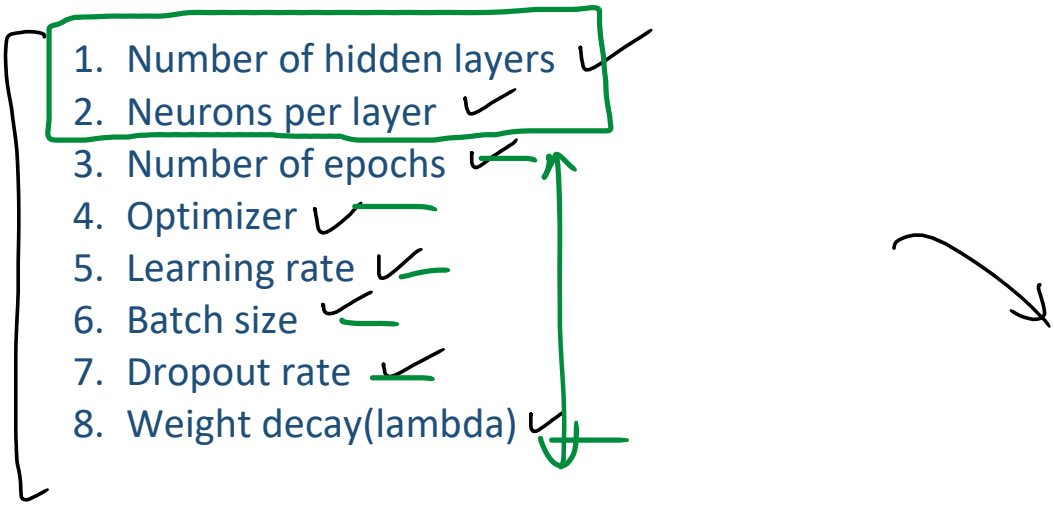
Recap

13 January 2025 16:40



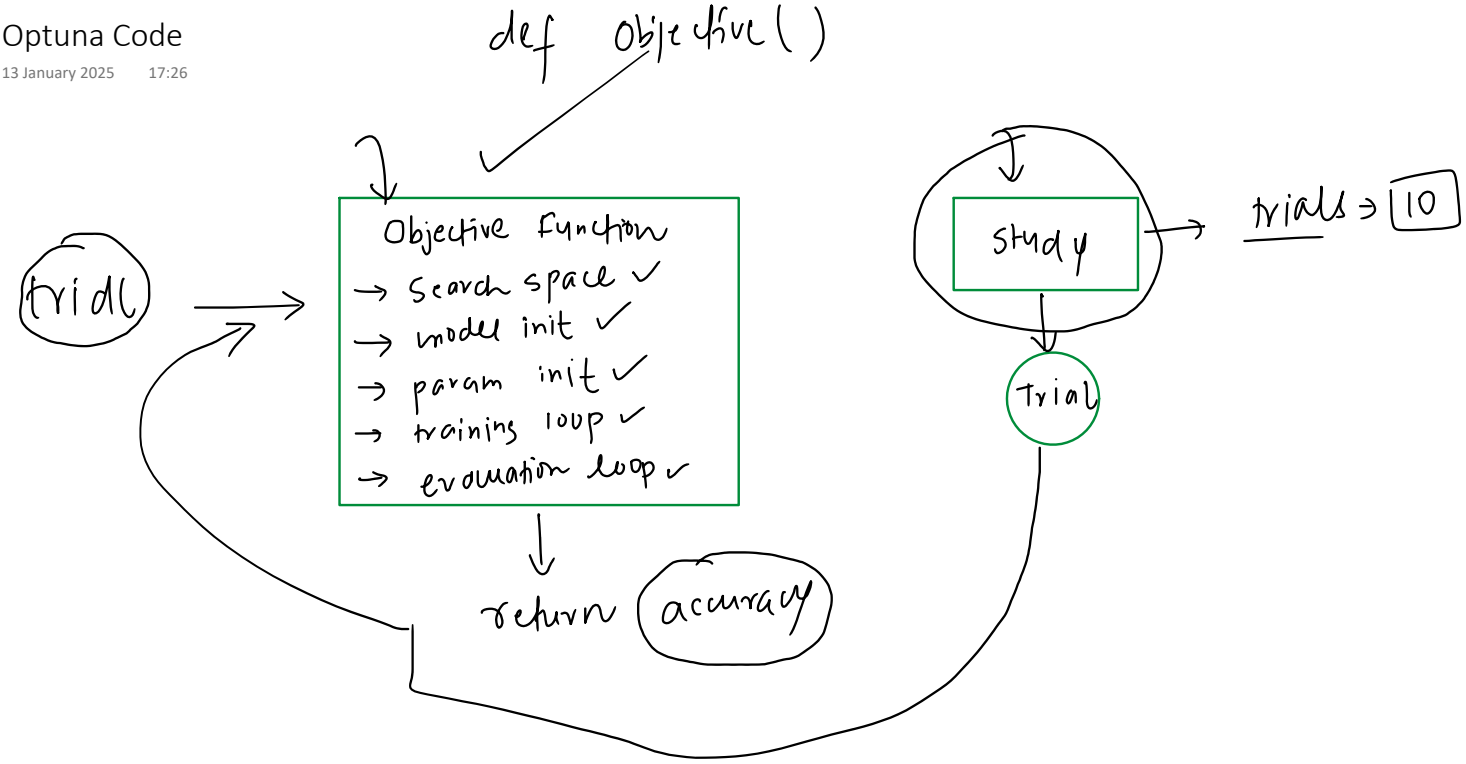
Plan of Action

13 January 2025 16:40



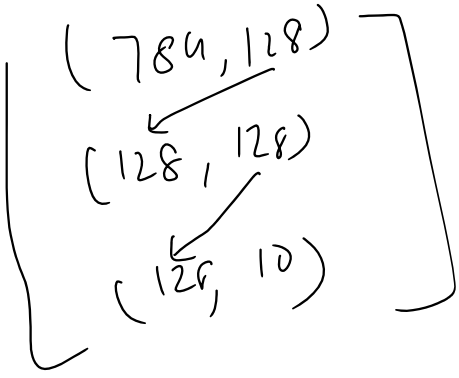
Optuna Code

13 January 2025 17:26



num_hidden_layers = 2

neurons_per_layer = 128

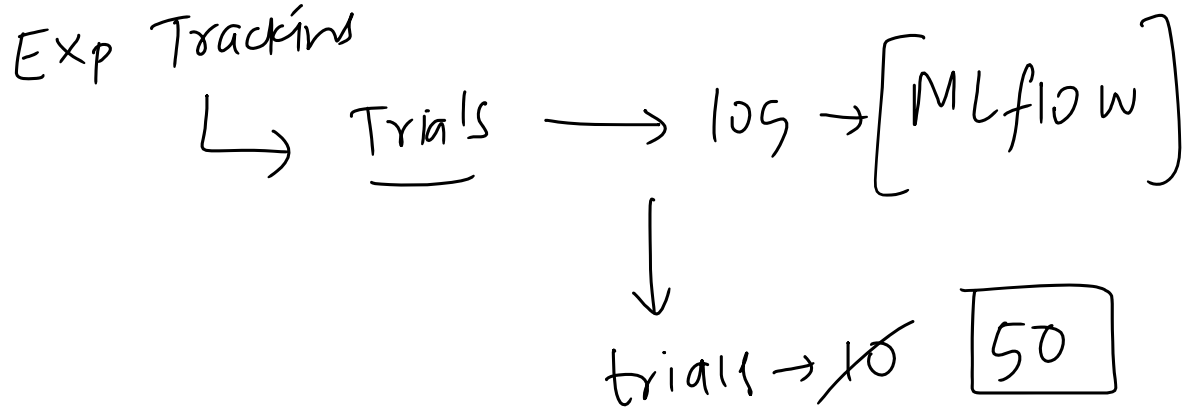


Round 1 → optuna

Round 2 → 6

Optuna x MLflow

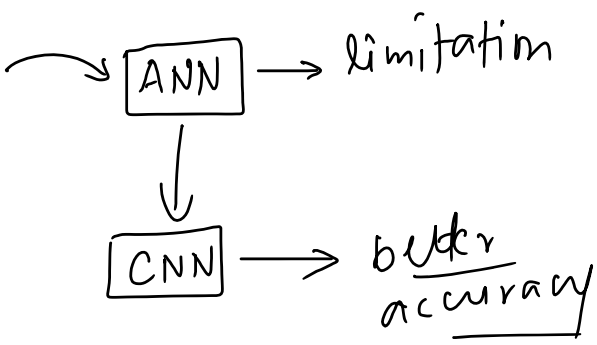
13 January 2025 19:02



Recap

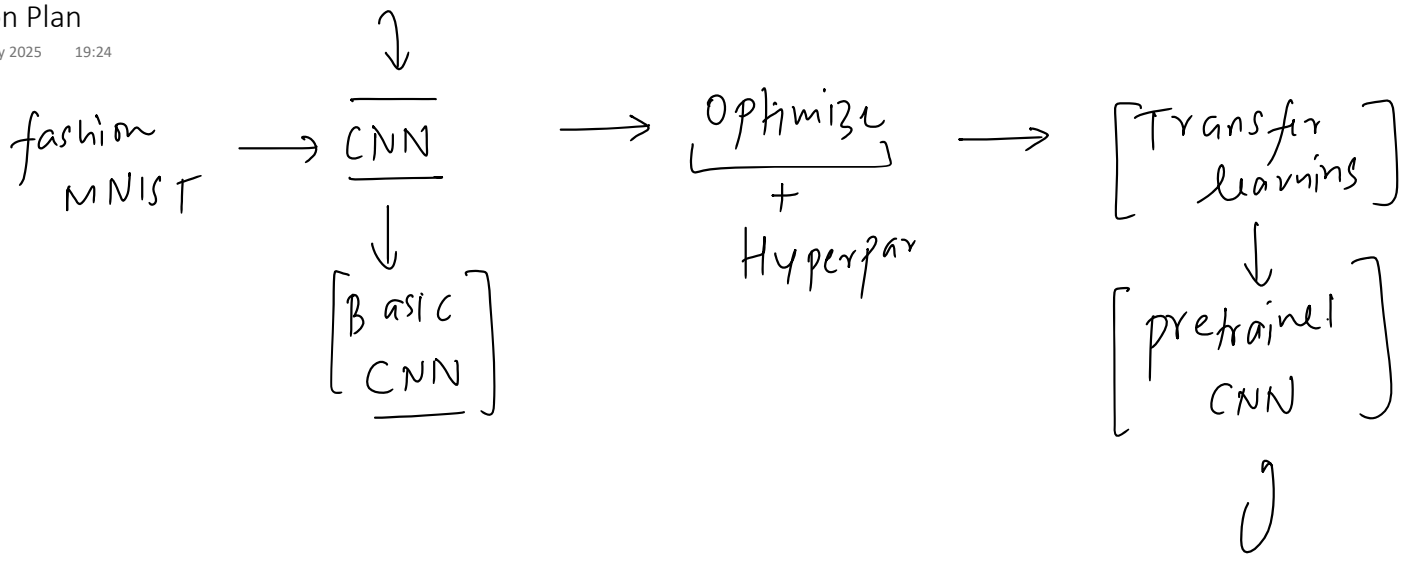
18 January 2025 18:37

S. No.	Model	Accuracy
1	ANN(partial dataset on CPU)	83.2%
2	ANN(full dataset on GPU)	88.9% (Overfitting)
3	ANN (optimized)	88.9% (Reduced Overfitting)
4	ANN (Hyperparameter Tuned)	90.2%



Action Plan

18 January 2025 19:24



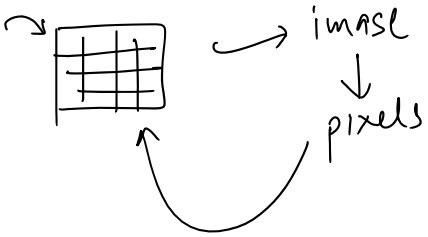
Prerequisite

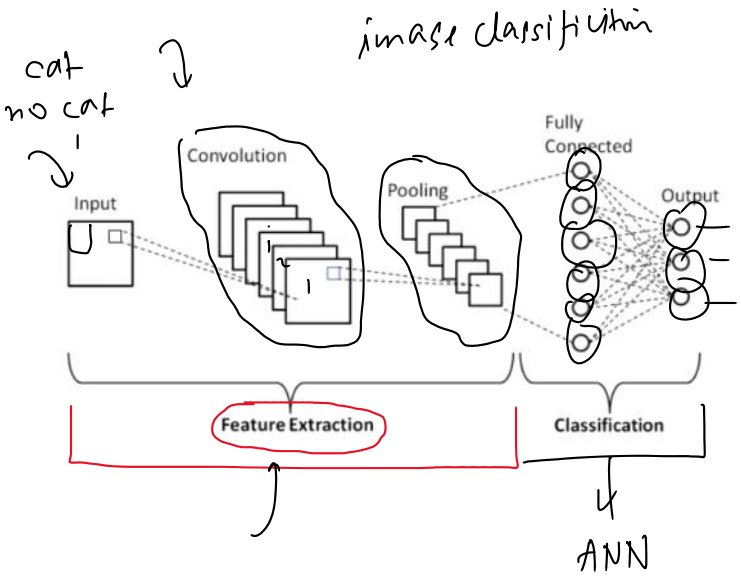
18 January 2025 19:24

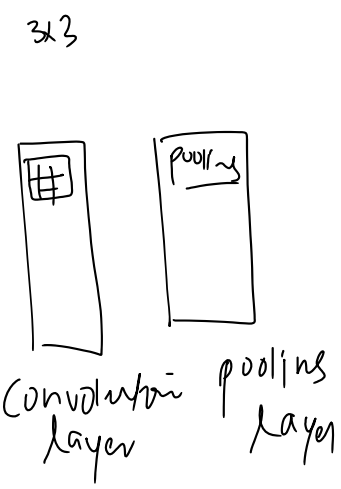
What is CNN

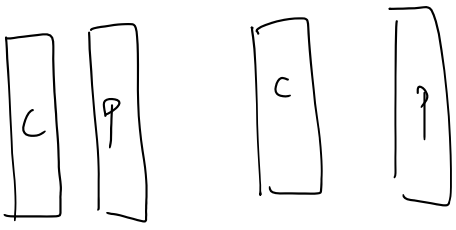
18 January 2025 18:38

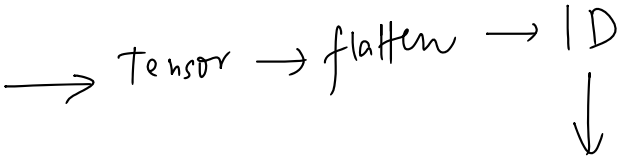
Convolutional Neural Networks (CNNs) are a type of deep learning model specifically designed to process and analyse structured data, such as images or videos. They are particularly effective in tasks like image classification, image recognition and object detection due to their ability to automatically and efficiently learn spatial hierarchies of features.





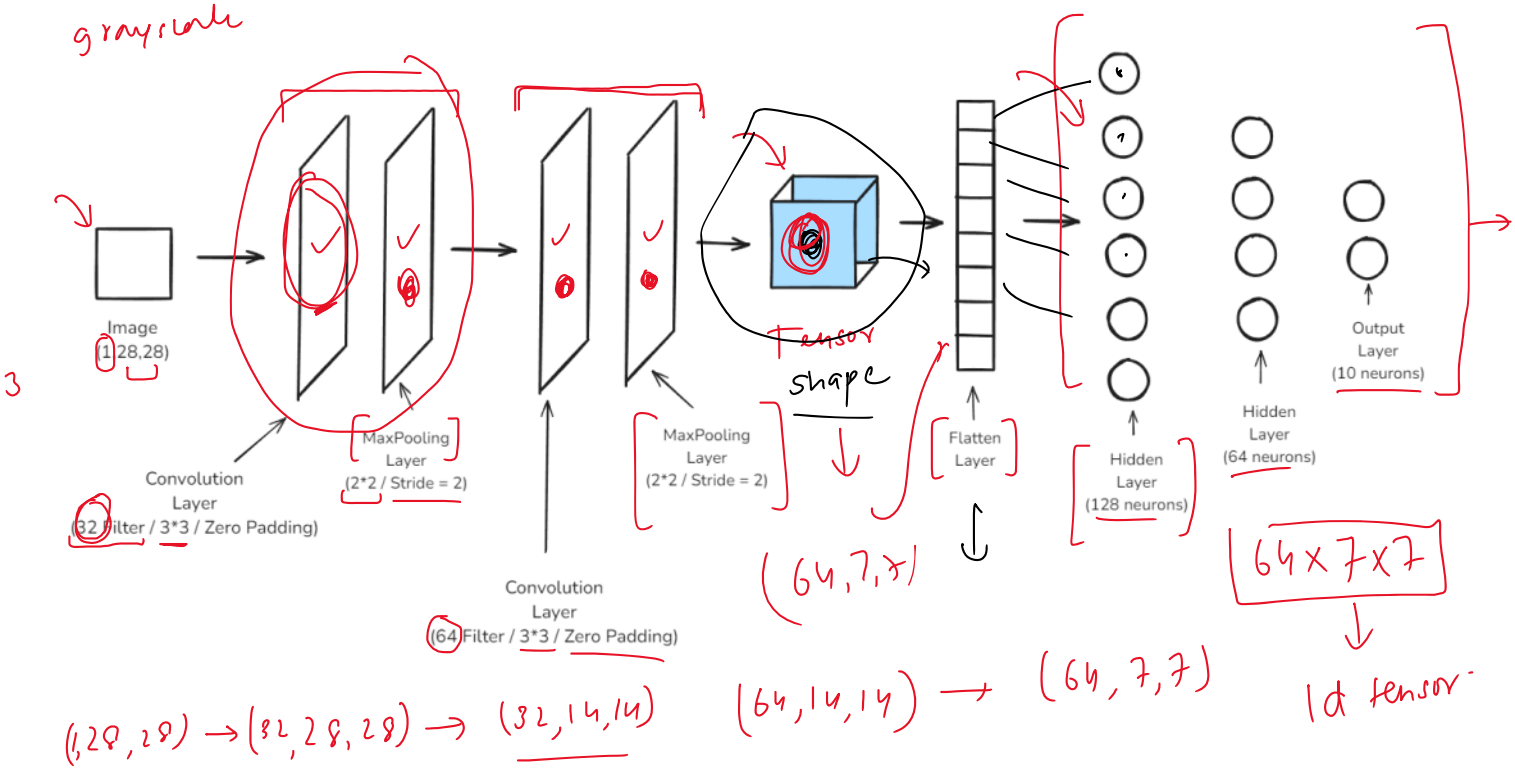






Architecture

18 January 2025 18:38

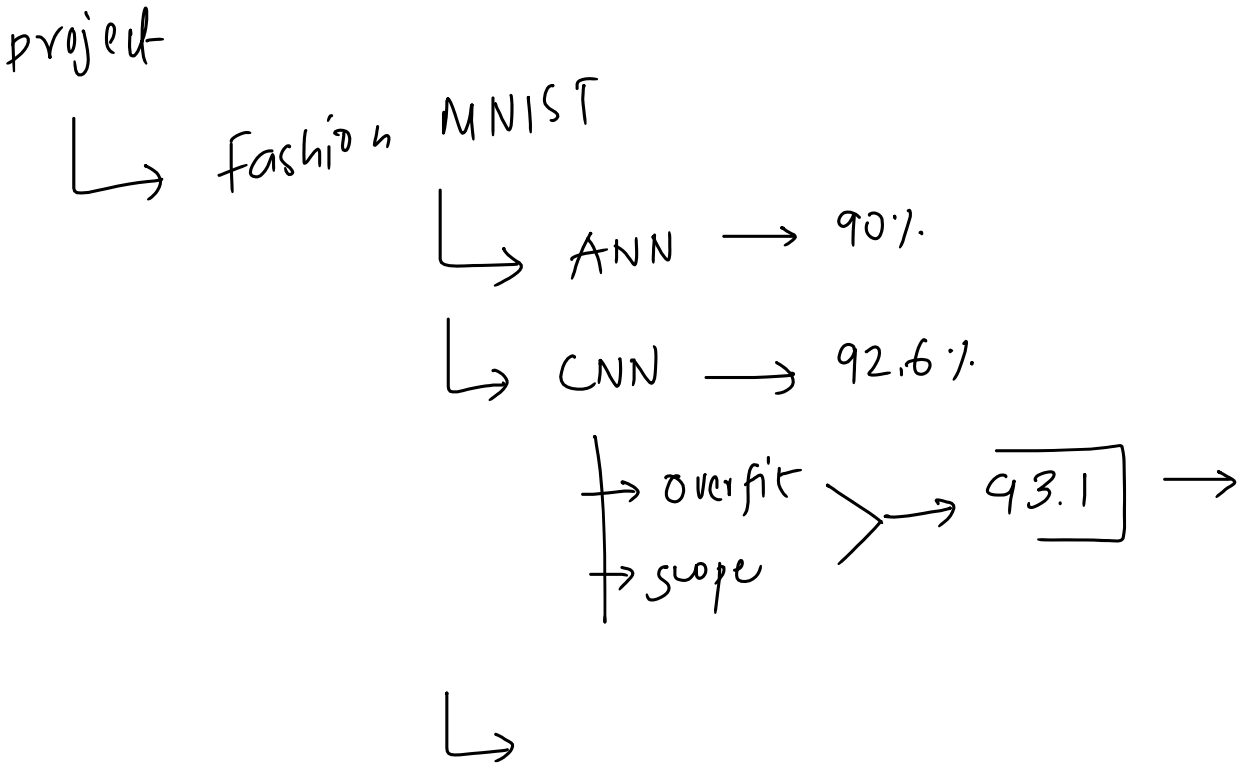


Improvements

18 January 2025 19:20

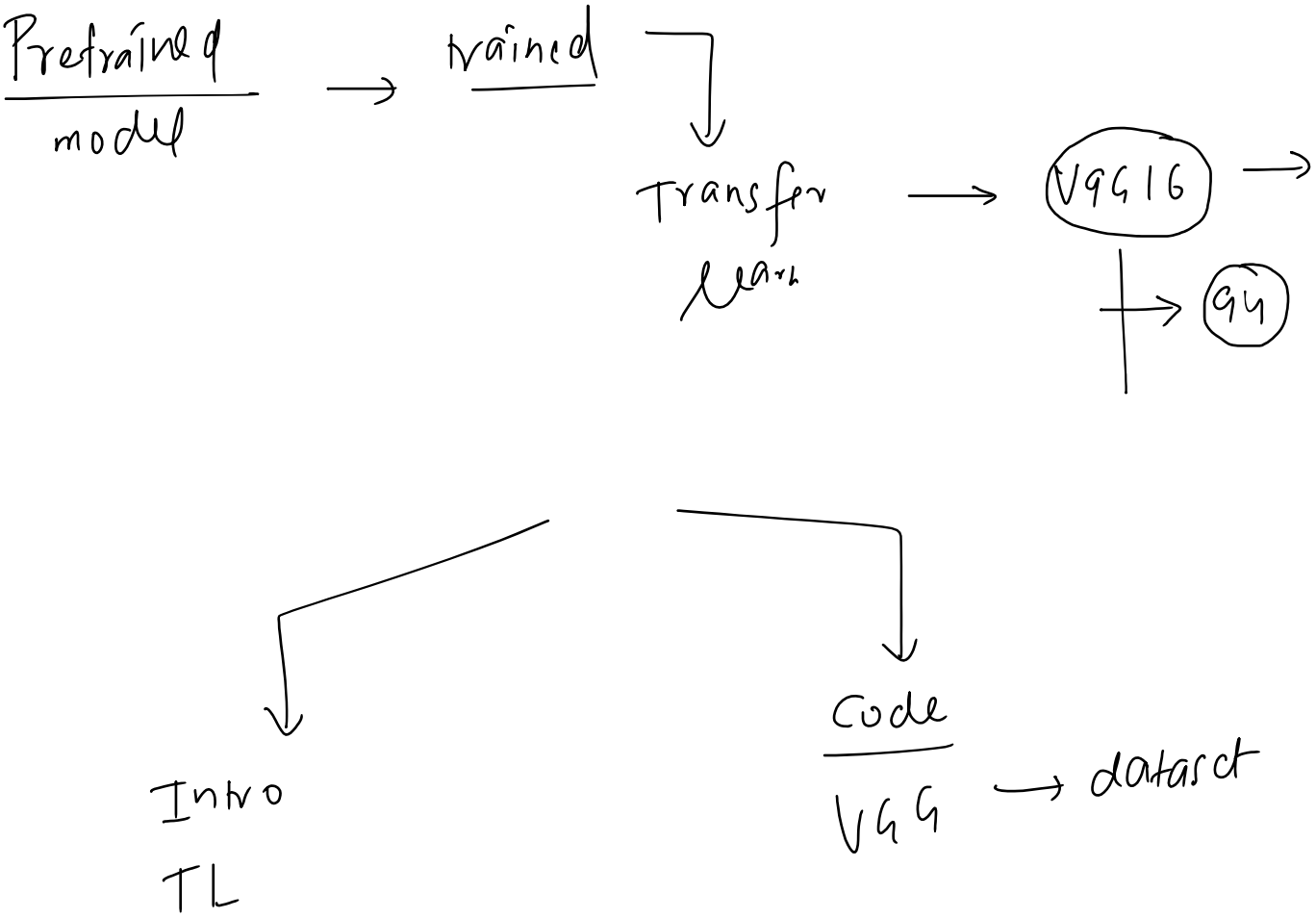
Recap

31 January 2025 11:11



Plan of Action

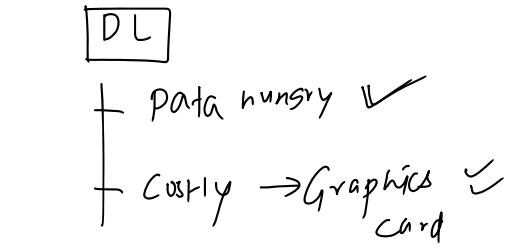
31 January 2025 11:12



What is Transfer Learning

31 January 2025 11:12

Transfer learning is a machine learning technique where a model trained on one task is reused (partially or fully) for a different but related task. Instead of training a model from scratch, which can be computationally expensive and require large datasets, transfer learning leverages knowledge from a pre-trained model to improve learning efficiency and performance.

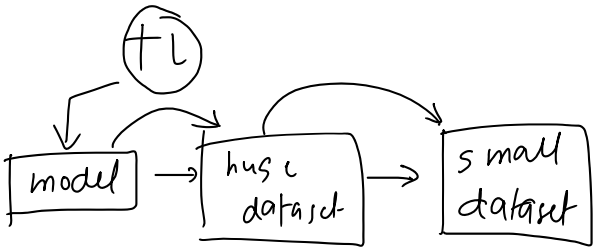


How Transfer Learning Works

1. Pretraining on a Large Dataset
   - A model is first trained on a large dataset (e.g., ImageNet for images, GPT for text).
   - The model learns general features, such as edges and shapes in images or syntax and semantics in text.
2. Fine-Tuning for a New Task
   - The pre-trained model is then adapted to a new, often smaller, dataset.
   - Some layers may be frozen (not updated), while others are fine-tuned for the specific task.

1.4 million

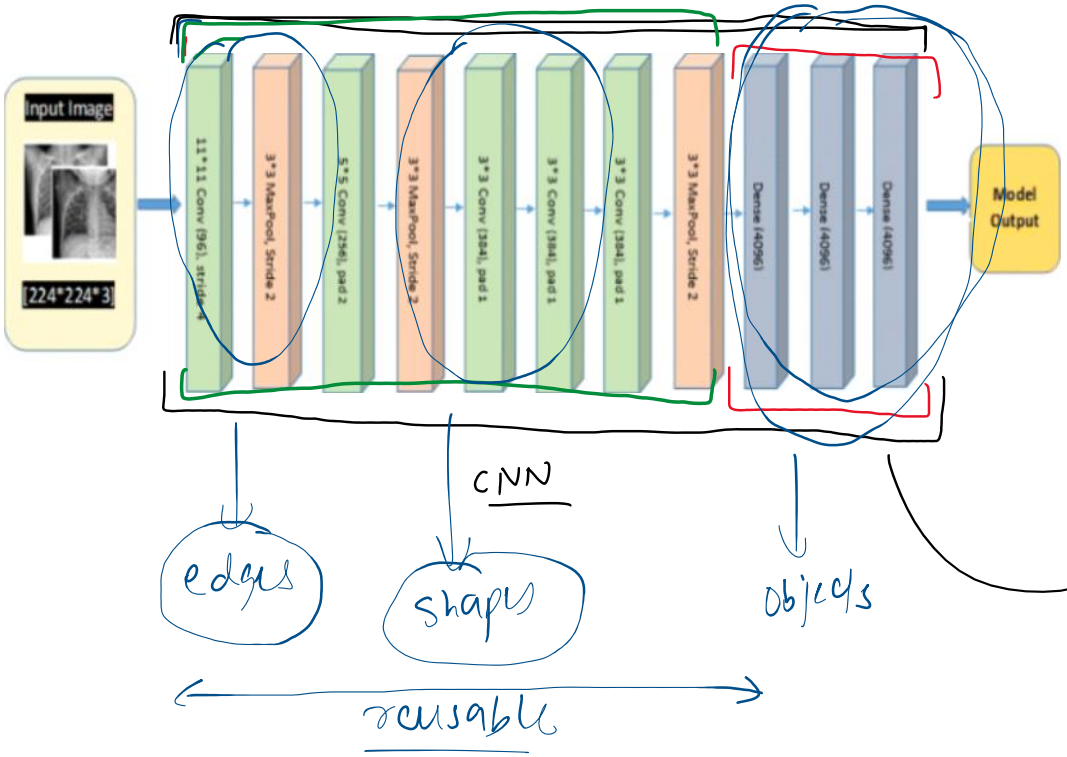
day to day



Why does Transfer Learning work?

pretraining

31 January 2025 16:49



Large dataset

Small dataset

CNN

edges

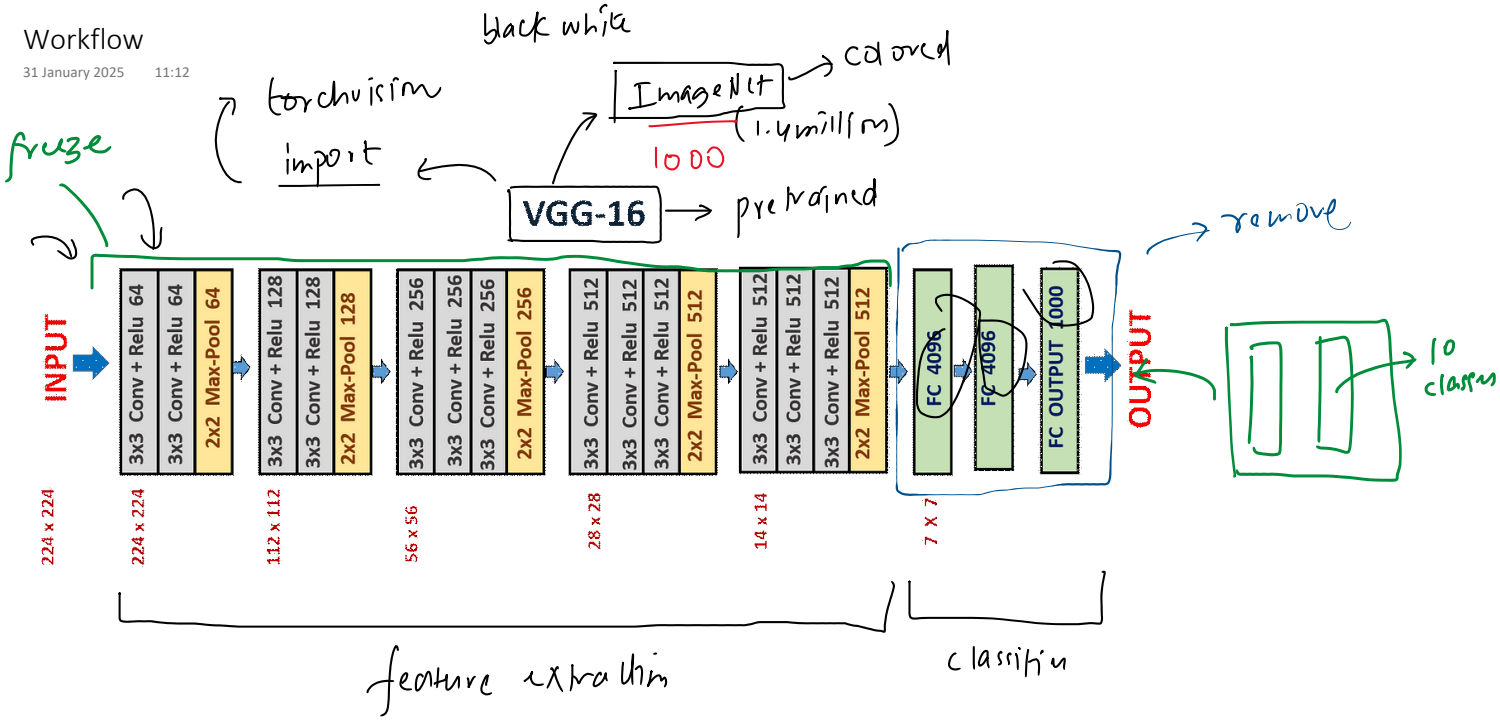
shapes

objects

reusable

Workflow

31 January 2025 11:12



1) import
2) detach classifier
3) attach classifier
4) freeze fe layers
5) train

Data prepare

[modify]

tabular (1, 784)

pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783	pixel784
0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

(1, 28, 28) 1) ~~reshape (28, 28)~~ → 2d (matrix) → PIL

(1,28,28) 1) ~~reshape (28,28)~~ → 2d (matrix) → PIL

→ 2) ~~datatype~~ → ~~(np.uint8)~~

→ 3) ~~1D → 3D~~ → (3,28,28)

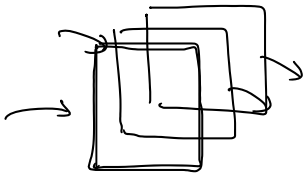
→ 4) tensor → PIL image (3,28,28)

5) ~~resize (3,256,256)~~ →

6) ~~center crop~~ (3,224,224) →

7) ~~tensor (scale)~~ → (0,1) →

8) ~~normalize~~ →



$$p = \frac{p - \mu}{\sigma}$$

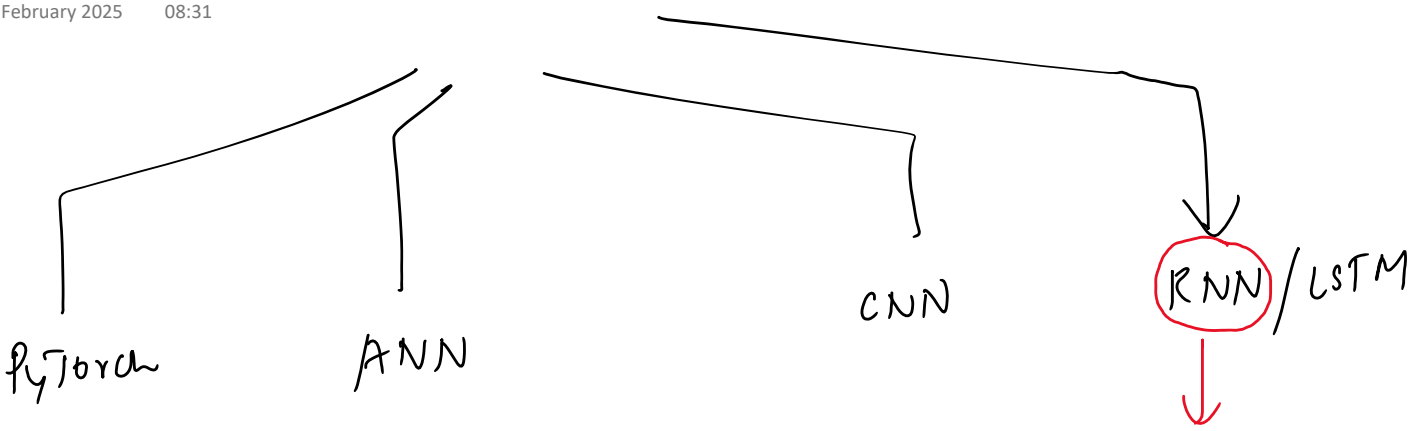
(3,224,224)

Code

31 January 2025 11:13

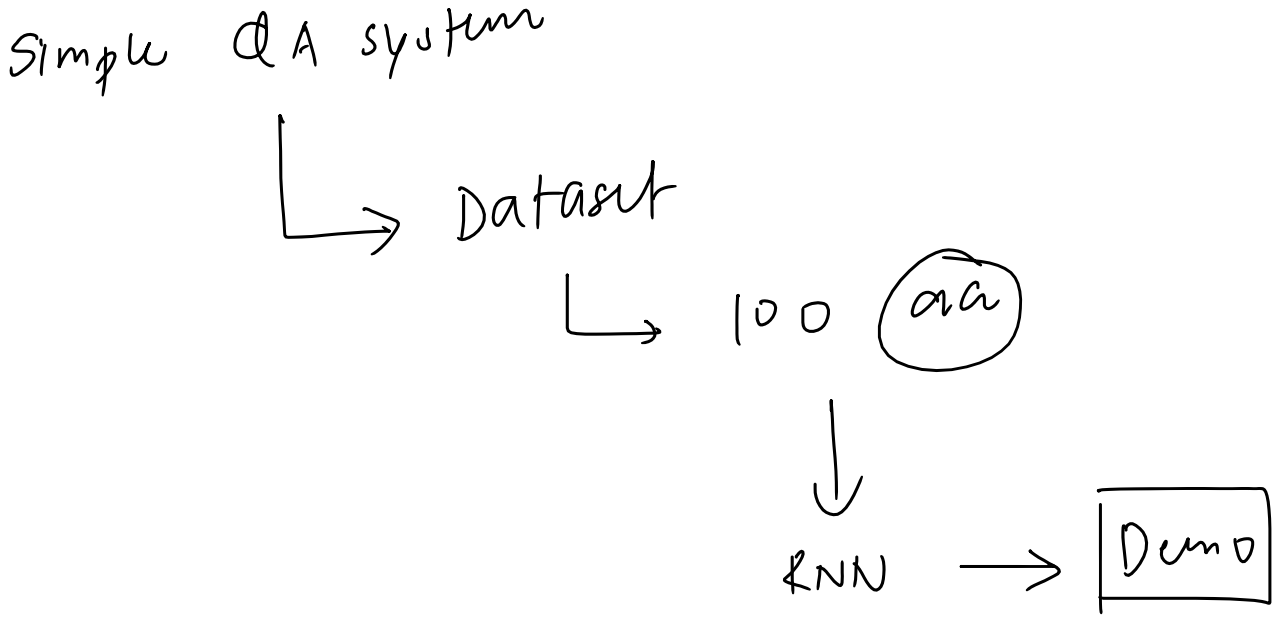
Recap

05 February 2025 08:31



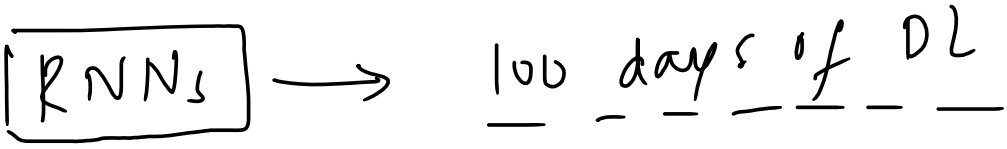
Plan of Action

05 February 2025 08:32



Prerequisites

05 February 2025 08:52



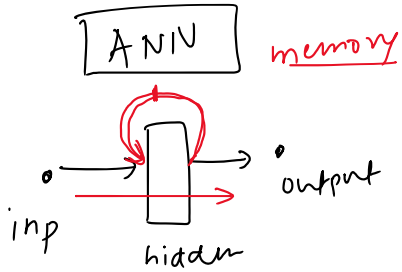
What is RNN

05 February 2025 08:32

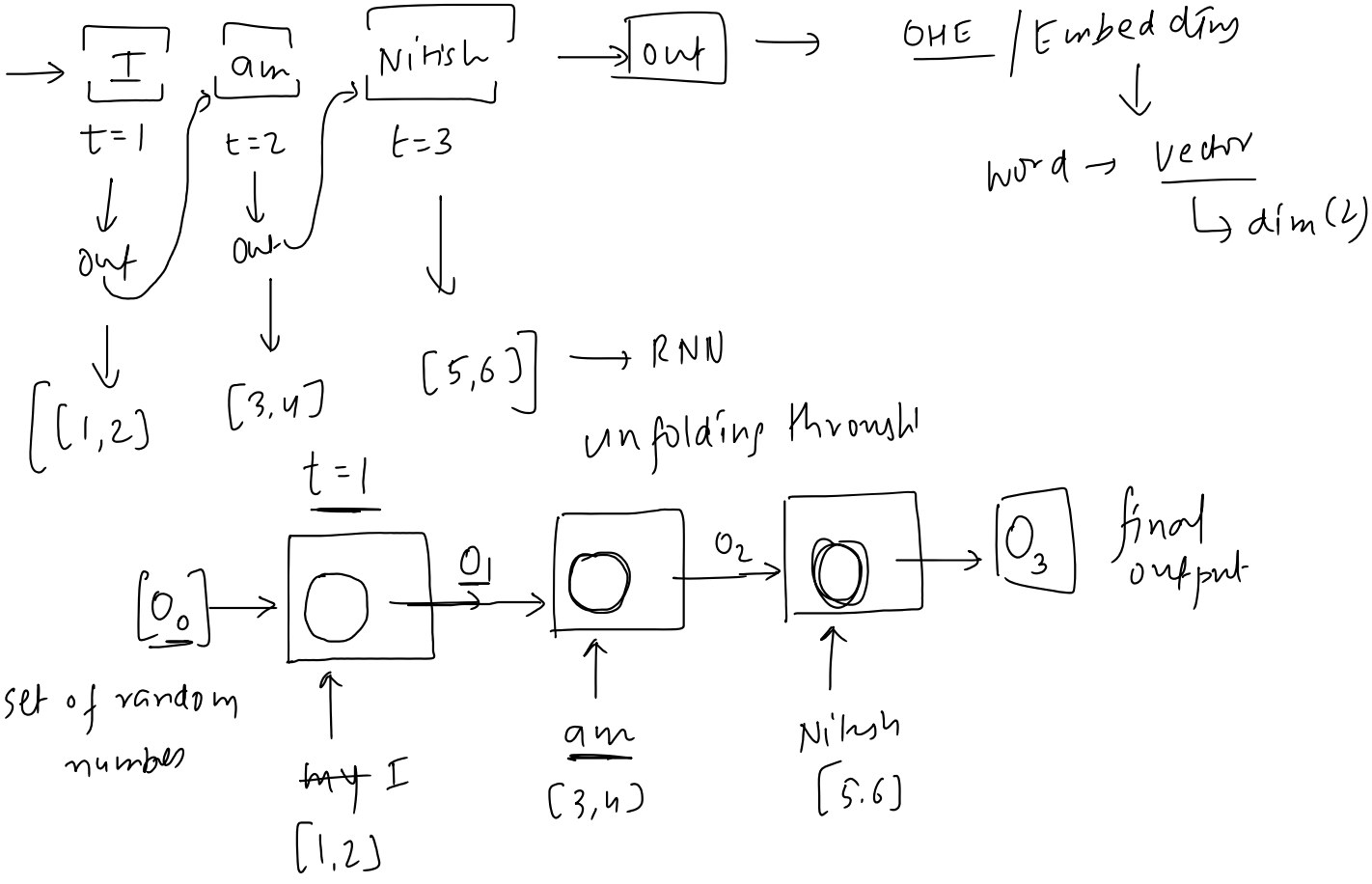
A Recurrent Neural Network (RNN) is a type of neural network designed for processing sequential data. Unlike traditional feedforward networks, which process inputs independently, RNNs maintain a memory of previous inputs by using loops in their architecture.

This makes them well-suited for tasks where context and order matter, such as time series forecasting, speech recognition and text generation.

RNN → NN



RNNs → sequential → textual

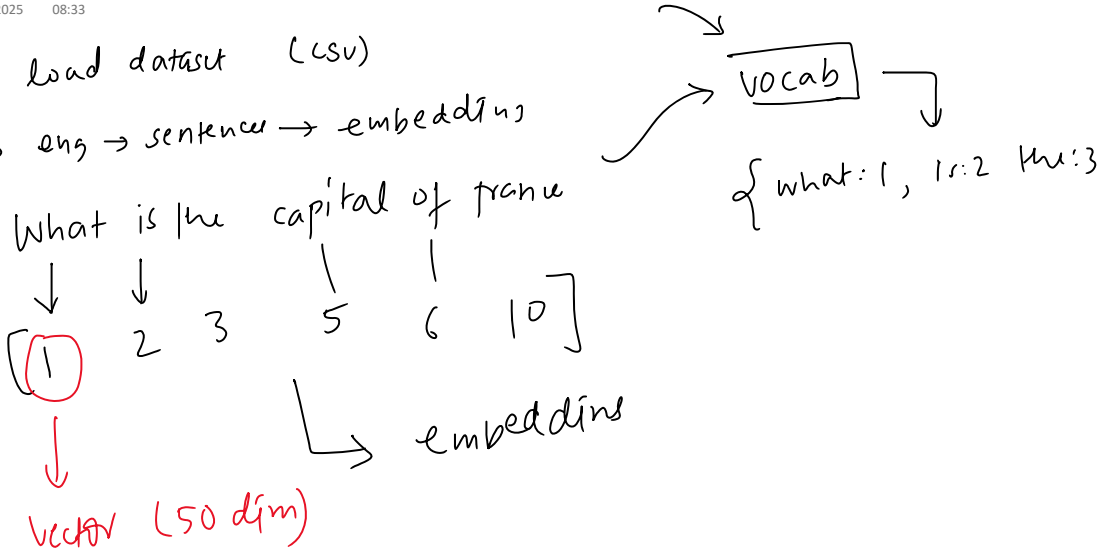


Strategy

05 February 2025 08:33

1) Load dataset (csv)

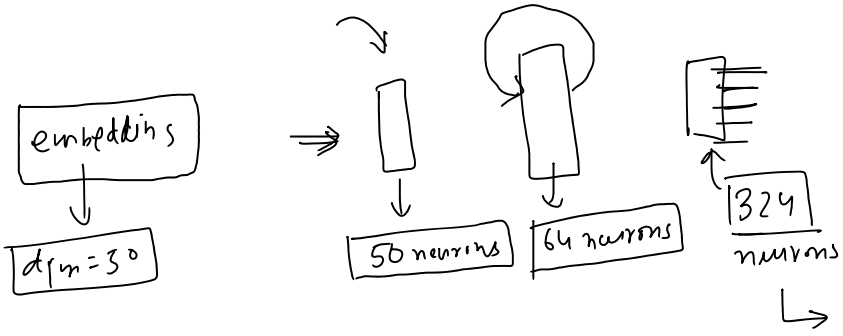
2) eng → sentence → embeddings



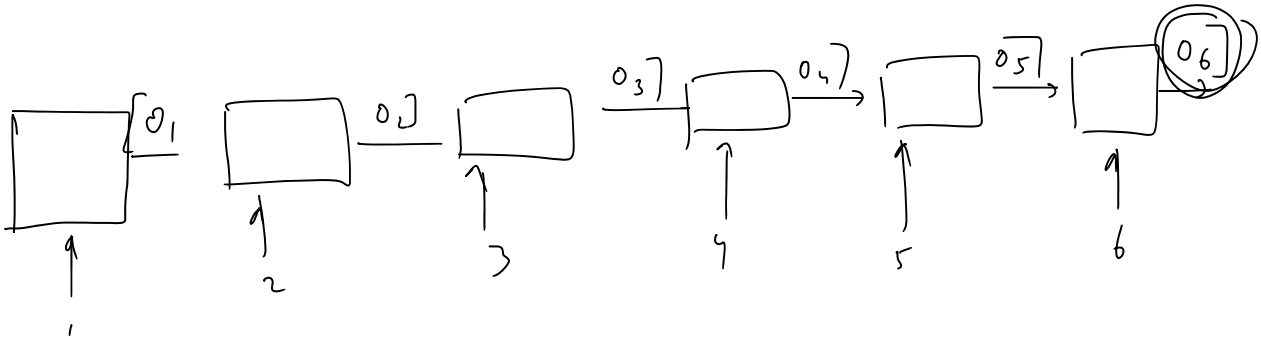
3) build RNN

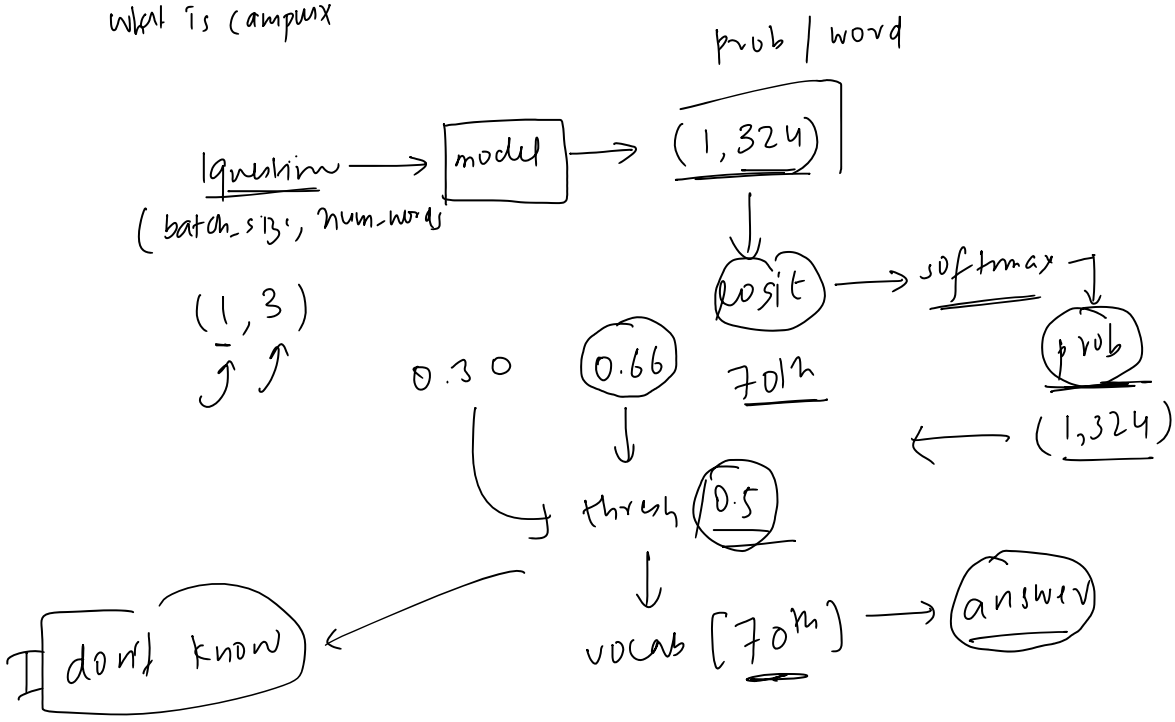
4) train

5) Prediction



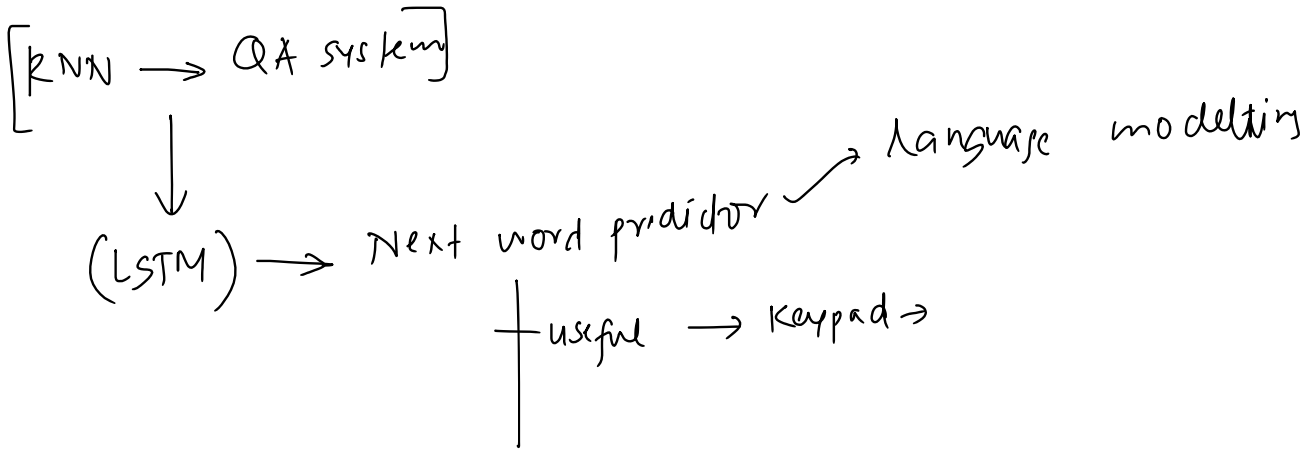
[1, 2, 3, 4, 5, 1]




what is campus
prob / word
question
model
(1,324)
(batch_size, num_words
(1,3)
posit
softmax
prob
(1,324)
0.30
0.66
70k
through
0.5
don't know
vocab [70k]
answer

Plan of Attack

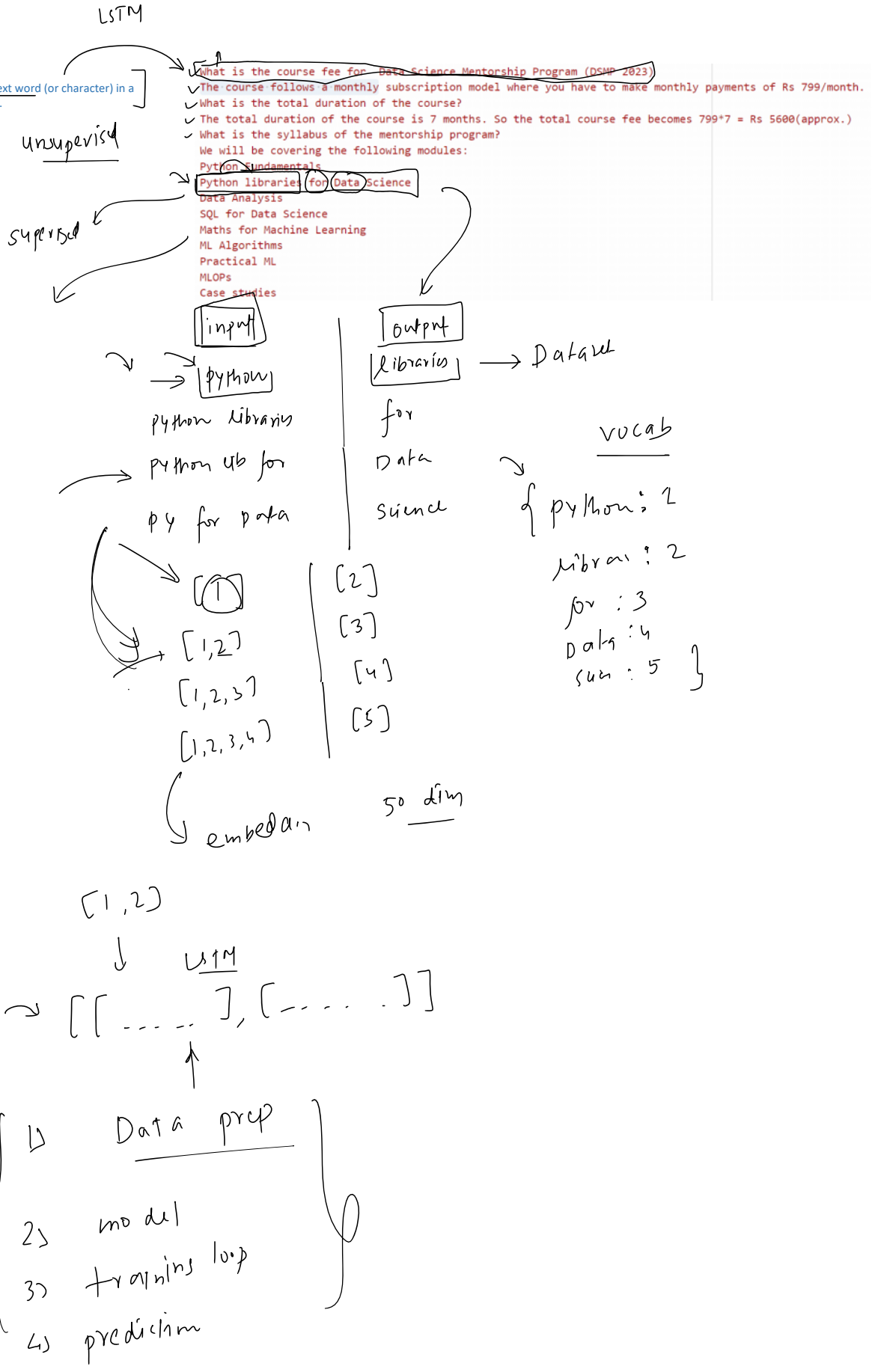
06 February 2025 09:17

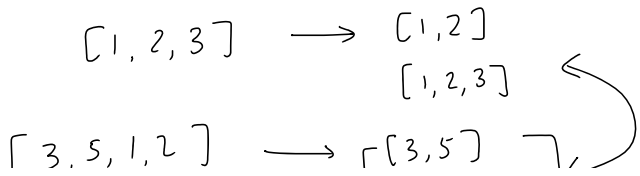


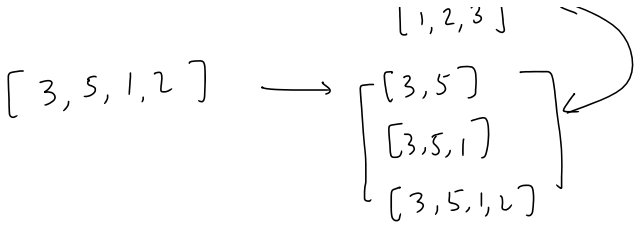
Strategy

06 February 2025 09:18

Language Modeling is the task of predicting the next word (or character) in a sequence based on the context of previous words.







942 rows → lstm

↳ in batches (32)

32 samples → size

lstm → paddins

pre

[1,2] → [0,1,2]

[1,2,3] [1,2,3]

942 → diff

↳ same

↳ find longest sequence

LSTM Architecture

06 February 2025 09:18