Neural Network Optimization

Dataset Preparation

1. Data Splitting

- **Training Set**: Used to update model weights (60-80%).
- **Validation Set**: Tune hyperparameters (10-20%).
- **Test Set**: Final evaluation (10-20%).

2. Normalization/Standardization

- **Min-Max Scaling**:

$$X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

- **Z-Score Standardization**:

$$X_{\text{std}} = \frac{X - \mu}{\sigma}$$

3. Data Augmentation

- Techniques: Rotation, flipping, cropping (for images).
- Goal: Increase dataset diversity and reduce overfitting.

4. Shuffling & Batching

- **Mini-batch Gradient Descent**:
  Split data into batches (e.g., 32, 64, 128 samples) for efficient training.

Weight Initialization

1. Zero Initialization (Avoid!)

- Fails to break symmetry: All neurons learn the same features.

2. Random Initialization

- **Normal Distribution**:

$$W_{ij} \sim \mathcal{N}(0, \sigma^2)$$

- **Uniform Distribution**:

$$W_{ij} \sim U(-a, a)$$

3. Xavier/Glorot Initialization

- For **tanh/sigmoid** activations:

$$\sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

where $n_{\text{in}}, n_{\text{out}}$ are input/output units.

4. He Initialization

- For **ReLU** activations:

$$\sigma = \sqrt{\frac{2}{n_{\text{in}}}}$$

Optimization Methods

1. Stochastic Gradient Descent (SGD)

- **Weight Update**:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

  - α: Learning rate.

2. SGD with Momentum

- Accumulates velocity in directions of persistent reduction:
  Velocity Update:

$$v_t := \beta v_{t-1} + (1 - \beta) \nabla_{\mathbf{w}} J(\mathbf{w})$$

  Weight Update:

$$\mathbf{w} := \mathbf{w} - \alpha v_t$$

  - β: Momentum coefficient (e.g., 0.9).

3. AdaGrad (Adaptive Gradient)

- Adapts learning rates per parameter:
  Cache Accumulation:

$$c_t := c_{t-1} + (\nabla_{\mathbf{w}} J(\mathbf{w}))^2$$

  Weight Update:

$$\mathbf{w} := \mathbf{w} - \frac{\alpha}{\sqrt{c_t} + \epsilon} \nabla_{\mathbf{w}} J(\mathbf{w})$$

  - ϵ: Small constant (e.g., 10^{-8}) for numerical stability.

4. RMSProp

- Improves AdaGrad by decaying cache:
  Cache Update:

$$c_t := \beta c_{t-1} + (1-\beta)(\nabla_{\mathbf{w}} J(\mathbf{w}))^2$$

  Weight Update:

$$\mathbf{w} := \mathbf{w} - \frac{\alpha}{\sqrt{c_t} + \epsilon} \nabla_{\mathbf{w}} J(\mathbf{w})$$

5. Adam (Adaptive Moment Estimation)

- Combines momentum and RMSProp:
  Moment Update:

$$m_t := \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\mathbf{w}} J(\mathbf{w})$$

  Cache Update:

$$v_t := \beta_2 v_{t-1} + (1-\beta_2)(\nabla_{\mathbf{w}} J(\mathbf{w}))^2$$

  Bias Correction:

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

  Weight Update:

$$\mathbf{w} := \mathbf{w} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

  - Defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Optimization Challenges & Solutions

Challenge	Solution
Vanishing Gradients	ReLU, Residual Connections
Exploding Gradients	Gradient Clipping, Batch Norm
Local Minima	Momentum, Adam
Overfitting	Dropout, L2 Regularization

Hyperparameter Tuning

- **Learning Rate (α)**: Use learning rate schedules (e.g., decay over epochs).
- **Batch Size**: Affects convergence speed and gradient noise.
- **Regularization Strength (λ)**: Balances loss and weight magnitudes.

Hardware & Software for Neural Networks

Hardware Components

1. **CPUs (Central Processing Units)**

- **Role**: General-purpose computation, data preprocessing.
- **Example**: Intel i9, AMD Ryzen.
- **Limitation**: Limited parallelization for large matrix operations.

2. **GPUs (Graphics Processing Units)**

- **Role**: Accelerate matrix operations via parallel processing (CUDA cores).
- **Example**: NVIDIA RTX 4090 (24GB VRAM), A100 (80GB VRAM).
- **Key Features**:
  - High memory bandwidth.
  - CUDA/ROCm support for deep learning frameworks.

3. **TPUs (Tensor Processing Units)**

- **Role**: Google's custom ASIC optimized for tensor operations.
- **Example**: TPU v4 (Google Cloud).
- **Use Case**: Large-scale training in frameworks like TensorFlow/JAX.

4. **FPGAs & ASICs**

- **Role**: Custom hardware for specific neural network tasks.
- **Example**: Intel Stratix (FPGA), Cerebras Wafer-Scale Engine (ASIC).

Hardware Comparison

Component	Pros	Cons
CPU	Versatile, low latency	Slow for large NN training
GPU	Massive parallelism	High cost, power consumption
TPU	Optimized for tensor math	Limited framework support

Software Frameworks

1. **PyTorch**

- **Developed by**: Meta (Facebook).
- **Key Features**:
  - Dynamic computation graph (define-by-run).
  - GPU acceleration via CUDA.
  - Autograd for automatic differentiation.
  - Rich ecosystem (TorchVision, TorchText).

Example: Simple Neural Network in PyTorch

```
import torch
import torch.nn as nn

# Define a model
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)  # Input: 28x28=784
        self.fc2 = nn.Linear(128, 10)    # Output: 10 classes

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

model = Net().to('cuda')  # Move to GPU
```

2. **TensorFlow**

- **Developed by**: Google.
- **Key Features**:
  - Static computation graph (define-and-run).
  - TensorFlow Serving for deployment.

3. **JAX**

- **Developed by**: Google.
- **Key Features**:
  - Autograd + XLA (Accelerated Linear Algebra).
  - Functional programming style.

PyTorch Workflow

1. **Installation**

```
# Install PyTorch with CUDA (check https://pytorch.org/)
conda install pytorch torchvision cudatoolkit=11.8 -c pytorch
```

2. **Core Components**

- **Tensors**: GPU-accelerated arrays.

```
x = torch.randn(3, 3).cuda()  # Create tensor on GPU
```

- **Autograd**: Automatic differentiation.

```
x.requires_grad_(True)
y = x.sum()
y.backward()  # Computes gradients ∇y/∂x
```

- **DataLoader**: Batch processing.

```
from torch.utils.data import DataLoader
loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

3. **Training Loop**

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

for epoch in range(10):
    for inputs, labels in loader:
        inputs, labels = inputs.to('cuda'), labels.to('cuda')
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Hardware-Software Integration

1. **GPU Acceleration**

- **Move tensors to GPU**:

```
tensor = tensor.to('cuda')
```

- **Check GPU availability**:

```
torch.cuda.is_available()  # Returns True/False
```

2. **Distributed Training**

- **Data Parallelism**:

```
model = nn.DataParallel(model)  # Split batch across GPUs
```

- **DistributedDataParallel (DDP)**:
  Multi-node/multi-GPU training (advanced).

Tools & Libraries

Tool	Purpose	Example
CUDA	GPU programming (NVIDIA)	`torch.cuda`
cuDNN	Optimized deep learning kernels	Used by PyTorch/TensorFlow
TensorBoard	Visualization	`torch.utils.tensorboard`
ONNX	Model interoperability	`torch.onnx.export`
Docker	Environment reproducibility	Prebuilt PyTorch images

Challenges

1. **Hardware Cost**: High-end GPUs/TPUs are expensive.
2. **Software Compatibility**: CUDA versions vs framework support.
3. **Scalability**: Distributed training requires expertise.

Resources

- PyTorch Documentation
- NVIDIA CUDA Toolkit
- Google Colab: Free GPU access.

Fully Connected Neural Network (FCNN)

Definition

A neural network where every neuron in one layer is connected to every neuron in the next layer. Also called a **Dense Network**.

Mathematical Formulation

- **Forward Pass for Layer l**:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

$$\mathbf{a}^{[l]} = g^{[l]}(\mathbf{z}^{[l]})$$

  - $\mathbf{W}^{[l]}$: Weight matrix
  - $\mathbf{a}^{[l-1]}$: Output of previous layer
  - $g^{[l]}$: Activation function (e.g., ReLU, Sigmoid)

Advantages

- Simple to implement and interpret.
- Universal approximator: Can model any continuous function.

Drawbacks

- **Parameter Explosion**: High-dimensional inputs (e.g., images) require too many weights.
- **No Spatial Awareness**: Ignores local patterns in structured data (e.g., pixels).

Convolutional Neural Network (CNN)

Definition

A neural network designed for **grid-like data** (e.g., images, audio) using convolutional layers to extract spatial hierarchies of features.

Key Components

1. **Convolutional Layer**: Applies filters to detect local patterns.
2. **Activation Layer**: Introduces nonlinearity (e.g., ReLU).
3. **Pooling Layer**: Reduces spatial dimensions (e.g., max pooling).
4. **Fully Connected Layer**: Final classification/regression.

Mathematical Formulation

1. Convolution Operation

For input $\mathbf{X}$ (2D image) and filter $\mathbf{W}$:
$$ (\mathbf{X} * \mathbf{W})*{i,j} = \sum*{m} \sum_{n} \mathbf{X}*{i+m, j+n} \cdot \mathbf{W}*{m,n} $$

- **Stride (s)**: Step size of filter movement.
- **Padding (p)**: Adds zeros around input to preserve size.

2. Activation (ReLU)

$$\mathbf{A}^{[l]} = \max(0, \mathbf{Z}^{[l]})$$

3. Max Pooling

$$ \mathbf{P}_{i,j} = \max(\mathbf{A}_{2i:2i+k, 2j:2j+k}) $$

- k: Pooling window size (e.g., 2x2).

CNN Structure Example

```
Input → Conv2D (32 filters, 3x3) → ReLU → MaxPool (2x2)
→ Conv2D (64 filters, 3x3) → ReLU → MaxPool (2x2)
→ Flatten → Dense (128 units) → ReLU → Dense (10 units) → Softmax
```

Classic Neural Networks

---

1. AlexNet (2012)

- **Key Contribution**: Popularized deep CNNs for ImageNet classification.
- **Architecture**:
  - 8 layers (5 conv, 3 dense).
  - ReLU activation, dropout, data augmentation.
  - Parallel GPU training.
- **Formula**:

$$\text{ReLU}(x) = \max(0, x)$$

2. VGG-16 (2014)

- **Key Contribution**: Demonstrated depth improves performance with small 3×3 filters.
- **Architecture**:
  - 16 layers (13 conv, 3 dense).
  - Uniform 3×3 conv blocks, max-pooling.

3. ResNet (2015)

- **Key Contribution**: Residual learning with skip connections solves vanishing gradients.
- **Architecture**:
  - Residual Block:

$$\mathbf{a}^{[l+2]} = g(\mathbf{W}^{[l+2]}\mathbf{a}^{[l+1]} + \mathbf{b}^{[l+2]}) + \mathbf{a}^{[l]}$$

  - Variants: ResNet-50, ResNet-152.

4. Inception (GoogLeNet, 2014)

- **Key Contribution**: Parallel filters (1×1, 3×3, 5×5) in "Inception modules".
- **Architecture**:
  - 22 layers with inception blocks.
  - 1×1 conv for dimensionality reduction.

Classic Datasets

1. MNIST

- **Type**: Handwritten digits (0-9).
- **Size**: 60k training + 10k test grayscale 28×28 images.
- **Use Case**: Basic image classification benchmark.

2. ImageNet

- **Type**: Natural images (1k classes).
- **Size**: ~1.2M training images (224×224 RGB).
- **Use Case**: Large-scale classification (ILSVRC challenge).

3. PASCAL VOC

- **Type**: Object detection/segmentation.
- **Size**: ~11k images with 20 classes.
- **Annotations**: Bounding boxes, segmentation masks.

4. COCO (Common Objects in Context)

- **Type**: Object detection/segmentation/captioning.
- **Size**: 330k images, 80 classes.
- **Annotations**: Bounding boxes, masks, captions.

Network Comparison

Network	Depth	Key Idea	Top-1 Accuracy (ImageNet)
AlexNet	8	ReLU + Dropout	63.3%
VGG-16	16	Deep 3×3 conv stacks	71.5%
ResNet	50+	Skip connections	76.5% (ResNet-50)
Inception	22	Multi-scale filters	74.8%

Dataset Comparison

Dataset	Task	Classes	Image Size
MNIST	Classification	10	28×28 (gray)
ImageNet	Classification	1,000	224×224 (RGB)
COCO	Detection/Segmentation	80	Variable
VOC	Detection	20	Variable

Legacy

- **AlexNet**: Sparked the deep learning revolution.
- **ResNet**: Enabled training of 1000+ layer networks.
- **COCO**: Standard benchmark for modern detection models (YOLO, Mask R-CNN).