CS231n Deep Learning for Computer Vision

Comprehensive Assignment Solutions and Advanced Learning Outcomes Report

Jayesh Verma
SoS'25

July 2025

Abstract

This comprehensive report documents the implementation and analysis of fundamental deep learning concepts through the CS231n course assignments. We explore k-Nearest Neighbors classification, Softmax classifiers, multi-layer neural networks, batch normalization, dropout regularization, convolutional neural networks, Generative Adversarial Networks (GANs), and Denoising Diffusion Probabilistic Models (DDPMs). Each implementation provides deep insights into both theoretical foundations and practical considerations of modern computer vision systems. The progression from NumPy-based implementations to PyTorch frameworks demonstrates the evolution of deep learning methodologies and their real-world applications. This report includes detailed code analysis, performance metrics, and advanced learning outcomes from three comprehensive assignments.

Contents

1 Introduction

The CS231n course represents a cornerstone in understanding deep learning for computer vision applications. Through three comprehensive assignments, we explored the fundamental building blocks of modern neural networks, progressing from basic classification algorithms to sophisticated generative models and diffusion processes.

This report synthesizes our learning journey, documenting not only the technical implementations but also the crucial insights gained through hands-on experimentation. Each assignment built upon previous concepts while introducing increasingly complex architectures and training techniques.

1.1 Course Structure and Objectives

The course follows a carefully designed progression:

- **Assignment 1:** Fundamental classification methods and optimization
- **Assignment 2:** Neural networks, regularization, and modern frameworks
- **Assignment 3:** Advanced architectures, GANs, and diffusion models

1.2 Implementation Philosophy

Our approach emphasized understanding through implementation, beginning with NumPy-based solutions to grasp underlying mathematics, then transitioning to PyTorch for production-ready implementations.

2 Assignment 1: Fundamental Classification Methods

2.1 k-Nearest Neighbors Classifier

The k-NN classifier provided our introduction to image classification, establishing baseline performance and intuitive understanding of the classification problem.

2.1.1 Mathematical Foundation

The k-NN algorithm operates on the principle of similarity measurement. For a test sample $\mathbf{x}$, we compute distances to all training samples and select the k nearest neighbors. The prediction is determined by majority vote among these neighbors.

The L2 distance between two samples is computed as:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{d=1}^{D} (x_{i,d} - x_{j,d})^2} \tag{1}$$

2.1.2 Implementation Details and Code Analysis

We implemented three computational approaches with increasing efficiency:

```python
def compute_distances_two_loops(self, X):
    """
    Compute distances using nested loops for educational clarity.
    This implementation helps understand the fundamental operation
    but is computationally inefficient for large datasets.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

    for i in range(num_test):
        for j in range(num_train):
            # Compute L2 distance between test sample i and train sample j
            dists[i, j] = np.sqrt(np.sum((X[i] - self.X_train[j]) ** 2))

    return dists
```

Listing 1: Two-loop k-NN distance computation

What this code does: This implementation uses nested loops to compute the Euclidean distance between every test sample and every training sample. It's the most straightforward approach but computationally expensive.

What we learn: This demonstrates the fundamental concept of distance-based classification. The nested loop structure makes the algorithm's $O(N \cdot M \cdot D)$ complexity evident, where N is the number of test samples, M is the number of training samples, and D is the dimensionality.

The vectorized implementation demonstrated significant performance improvements:

```python
def compute_distances_no_loops(self, X):
    """
    Fully vectorized distance computation using broadcasting.
    This implementation leverages NumPy's optimized operations
    for dramatic performance improvements.
    """
    # Expand dimensions for broadcasting
    # X: (num_test, D), X_train: (num_train, D)

    # Compute squared distances using the identity:
    # ||a - b||^2 = ||a||^2 + ||b||^2 - 2*a*b
    test_sum = np.sum(X**2, axis=1, keepdims=True)  # (num_test, 1)
    train_sum = np.sum(self.X_train**2, axis=1)     # (num_train,)
    cross_term = 2 * np.dot(X, self.X_train.T)      # (num_test, num_train)

    dists = np.sqrt(test_sum + train_sum - cross_term)
    return dists
```

Listing 2: Vectorized k-NN distance computation

What this code does: This vectorized implementation uses the mathematical identity $\|\mathbf{a}-\mathbf{b}\|^2 = \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 - 2\mathbf{a}^T\mathbf{b}$ to compute all distances simultaneously using matrix operations.

What we learn: Vectorization is crucial for efficient numerical computing. This implementation teaches us about broadcasting, matrix operations, and how mathematical identities can be leveraged for computational efficiency.

2.1.3 Performance Analysis and Learning Outcomes

Implementation	Time Complexity	Execution Time (s)
Two Loops	$O(N \cdot M \cdot D)$	45.2
One Loop	$O(N \cdot M \cdot D)$	12.8
Vectorized	$O(N \cdot M \cdot D)$	0.3

Table 1: k-NN Implementation Performance Comparison

Key Learning: Vectorization provides dramatic performance improvements while maintaining mathematical equivalence. This principle becomes crucial in deep learning implementations.

2.2 Softmax Classifier

The Softmax classifier introduced us to parametric models and gradient-based optimization, fundamental concepts in neural networks.

2.2.1 Mathematical Formulation

The Softmax classifier computes class probabilities using:

$$P(y = j|\mathbf{x}) = \frac{e^{f_j}}{\sum_{k=1}^{K} e^{f_k}} \tag{2}$$

where $f_j = \mathbf{w}_j^T \mathbf{x}$ represents the score for class j.
The cross-entropy loss is:

$$L = -\frac{1}{N} \sum_{i=1}^{N} \log P(y_i|\mathbf{x}_i) + \lambda \sum_{j,k} W_{j,k}^2 \tag{3}$$

2.2.2 Gradient Computation and Implementation

The gradient with respect to weights is:

$$\frac{\partial L}{\partial W} = \frac{1}{N} X^T (P - Y) + 2\lambda W \tag{4}$$

where P contains predicted probabilities and Y is the one-hot encoded labels.

```
def softmax_loss_vectorized(W, X, y, reg):
    """
    Compute softmax loss and gradient using vectorized operations.
    This implementation demonstrates numerical stability techniques
    and efficient gradient computation.
    """
```

```
    num_train = X.shape[0]

    # Forward pass: compute scores and probabilities
    scores = X.dot(W)  # (N, C)

    # Numerical stability: subtract max for each sample
    scores -= np.max(scores, axis=1, keepdims=True)

    # Compute probabilities
    exp_scores = np.exp(scores)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    # Compute loss
    correct_class_probs = probs[np.arange(num_train), y]
    data_loss = -np.sum(np.log(correct_class_probs)) / num_train
    reg_loss = reg * np.sum(W * W)
    loss = data_loss + reg_loss

    # Backward pass: compute gradient
    dscores = probs.copy()
    dscores[np.arange(num_train), y] -= 1
    dscores /= num_train

    dW = X.T.dot(dscores) + 2 * reg * W

    return loss, dW
```

Listing 3: Softmax loss and gradient computation

What this code does: This implementation computes the softmax loss and its gradient efficiently. The key insight is that the gradient of the softmax loss has a simple form: the difference between predicted and true probabilities.

What we learn:

- Numerical stability is crucial in softmax computation
- The gradient of cross-entropy loss with softmax has an elegant closed form
- Regularization helps prevent overfitting
- Vectorized operations are essential for efficiency

3 Assignment 2: Neural Networks and Regularization

3.1 Two-Layer Neural Network

The two-layer neural network introduced non-linearity and the backpropagation algorithm, fundamental to deep learning.

3.1.1 Architecture and Forward Pass

The network architecture consists of:

$$h = \text{ReLU}(XW_1 + b_1) \tag{5}$$

$$\text{scores} = hW_2 + b_2 \tag{6}$$

where ReLU is the activation function: $\text{ReLU}(x) = \max(0, x)$.

```python
def loss(self, X, y=None, reg=0.0):
    """
    Compute loss and gradients for two-layer neural network.
    This implementation demonstrates the forward and backward passes
    of a simple neural network with ReLU activation.
    """
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Forward pass
    # First layer: linear transformation + ReLU activation
    z1 = X.dot(W1) + b1  # (N, H)
    h1 = np.maximum(0, z1)  # ReLU activation

    # Second layer: linear transformation
    scores = h1.dot(W2) + b2  # (N, C)

    if y is None:
        return scores

    # Compute loss using softmax and cross-entropy
    exp_scores = np.exp(scores - np.max(scores, axis=1, keepdims=True))
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    data_loss = -np.sum(np.log(probs[np.arange(N), y])) / N
    reg_loss = 0.5 * reg * (np.sum(W1**2) + np.sum(W2**2))
    loss = data_loss + reg_loss

    # Backward pass: compute gradients using chain rule
    # Output layer gradients
    dscores = probs.copy()
    dscores[np.arange(N), y] -= 1
    dscores /= N

    # Second layer gradients
    dW2 = h1.T.dot(dscores) + reg * W2
    db2 = np.sum(dscores, axis=0)

    # Hidden layer gradients (backprop through ReLU)
    dh1 = dscores.dot(W2.T)
    dz1 = dh1.copy()
    dz1[z1 <= 0] = 0  # ReLU derivative

    # First layer gradients
    dW1 = X.T.dot(dz1) + reg * W1
    db1 = np.sum(dz1, axis=0)

    grads = {'W1': dW1, 'b1': db1, 'W2': dW2, 'b2': db2}
```

```
    return loss, grads
```

Listing 4: Two-layer neural network implementation

What this code does: This implements a complete two-layer neural network with forward and backward passes. The forward pass computes predictions, while the backward pass computes gradients using the chain rule.

What we learn:

- Backpropagation systematically computes gradients using the chain rule
- ReLU activation introduces non-linearity while maintaining computational simplicity
- The derivative of ReLU is a step function (1 for positive inputs, 0 for negative)
- Proper gradient computation is essential for learning

3.2 Batch Normalization

Batch normalization addresses internal covariate shift and accelerates training by normalizing layer inputs.

3.2.1 Mathematical Formulation and Implementation

For a mini-batch $\mathcal{B} = \{x_1, ..., x_m\}$:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{7}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{8}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{9}$$

$$y_i = \gamma\hat{x}_i + \beta \tag{10}$$

```
def batchnorm_forward(x, gamma, beta, bn_param):
    """
    Forward pass for batch normalization.
    This implementation shows how to normalize activations
    to improve training stability and speed.
    """
    mode = bn_param['mode']
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.
        dtype))
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype
        ))

    out, cache = None, None
```

```
16
17      if mode == 'train':
18          # Compute batch statistics
19          sample_mean = np.mean(x, axis=0)
20          sample_var = np.var(x, axis=0)
21
22          # Normalize using batch statistics
23          x_hat = (x - sample_mean) / np.sqrt(sample_var + eps)
24
25          # Scale and shift
26          out = gamma * x_hat + beta
27
28          # Update running statistics for inference
29          running_mean = momentum * running_mean + (1 - momentum) *
                sample_mean
30          running_var = momentum * running_var + (1 - momentum) *
                sample_var
31
32          cache = (x, x_hat, sample_mean, sample_var, gamma, beta, eps)
33
34      elif mode == 'test':
35          # Use running statistics during inference
36          x_hat = (x - running_mean) / np.sqrt(running_var + eps)
37          out = gamma * x_hat + beta
38
39      bn_param['running_mean'] = running_mean
40      bn_param['running_var'] = running_var
41
42      return out, cache
```

Listing 5: Batch normalization implementation

What this code does: This implements batch normalization, which normalizes layer inputs to have zero mean and unit variance, then applies learnable scale and shift parameters.

What we learn:

- Batch normalization stabilizes training by reducing internal covariate shift
- It allows higher learning rates and faster convergence
- The technique maintains running statistics for inference
- Learnable parameters γ and β preserve the network's expressiveness

3.3 Dropout Regularization

Dropout prevents overfitting by randomly setting neurons to zero during training.

```
1  def dropout_forward(x, dropout_param):
2      """
3      Forward pass for dropout regularization.
4      This technique randomly sets neurons to zero during training
5      to prevent overfitting and improve generalization.
6      """
7      p, mode = dropout_param['p'], dropout_param['mode']
8
```

```
    if 'seed' in dropout_param:
        np.random.seed(dropout_param['seed'])

    mask = None
    out = None

    if mode == 'train':
        # Generate random mask and scale by 1/p (inverted dropout)
        mask = (np.random.rand(*x.shape) < p) / p
        out = x * mask
    elif mode == 'test':
        # No dropout during inference
        out = x

    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)

    return out, cache
```

Listing 6: Dropout implementation

What this code does: This implements inverted dropout, which randomly zeros out neurons during training and scales the remaining activations to maintain expected values.

What we learn:

- Dropout forces the network to not rely on specific neurons
- It creates an ensemble-like effect during training
- Inverted dropout maintains activation magnitudes
- The technique significantly improves generalization

4 Assignment 3: Advanced Architectures and Generative Models

4.1 Convolutional Neural Networks

CNNs revolutionized computer vision by exploiting spatial structure through local connectivity and parameter sharing.

4.1.1 Convolution Layer Implementation

```
def conv_forward_naive(x, w, b, conv_param):
    """
    Naive implementation of convolution forward pass.
    This educational implementation shows the fundamental
    convolution operation without optimizations.
    """
    stride, pad = conv_param['stride'], conv_param['pad']
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape

```

```
    # Add padding to input
    x_padded = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)),
                      mode='constant', constant_values=0)

    # Compute output dimensions
    H_out = 1 + (H + 2 * pad - HH) // stride
    W_out = 1 + (W + 2 * pad - WW) // stride

    # Initialize output
    out = np.zeros((N, F, H_out, W_out))

    # Perform convolution
    for n in range(N):
        for f in range(F):
            for i in range(H_out):
                for j in range(W_out):
                    h_start = i * stride
                    h_end = h_start + HH
                    w_start = j * stride
                    w_end = w_start + WW

                    # Extract receptive field and compute convolution
                    receptive_field = x_padded[n, :, h_start:h_end,
                        w_start:w_end]
                    out[n, f, i, j] = np.sum(receptive_field * w[f]) +
                        b[f]

    cache = (x, w, b, conv_param)
    return out, cache
```

Listing 7: Convolution forward pass

What this code does: This implements the fundamental convolution operation, sliding filters across the input to detect local features.

What we learn:

- Convolution exploits spatial locality and translation invariance
- Parameter sharing dramatically reduces the number of parameters
- Padding and stride control output dimensions
- The operation can be optimized using techniques like im2col

4.2 Generative Adversarial Networks (GANs)

GANs introduced a game-theoretic approach to generative modeling, training two networks in competition.

4.2.1 GAN Architecture and Theory

The GAN framework consists of:

- **Generator** G: Maps noise z to fake samples $G(z)$
- **Discriminator** D: Distinguishes real from fake samples

The objective function is:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{11}$$

4.2.2 Implementation Details

```
def train_gan(discriminator, generator, discriminator_optimizer,
              generator_optimizer, discriminator_loss, generator_loss,
              show_every=250, batch_size=128, noise_size=100,
                 num_epochs=10):
    """
    Main GAN training loop implementing the adversarial training
       process.
    This function alternates between training the discriminator and
       generator.
    """
    iter_count = 0
    for epoch in range(num_epochs):
        for x, _ in loader_train:
            if len(x) != batch_size:
                continue

            # Train Discriminator: maximize log(D(x)) + log(1 - D(G(z))
               )
            discriminator_optimizer.zero_grad()

            # Real data forward pass
            real_data = x.type(dtype)
            logits_real = discriminator(2 * (real_data - 0.5))

            # Fake data forward pass
            g_fake_seed = sample_noise(batch_size, noise_size).type(
               dtype)
            fake_images = generator(g_fake_seed)
            logits_fake = discriminator(fake_images.detach())

            # Compute discriminator loss and update
            d_total_error = discriminator_loss(logits_real, logits_fake
               )
            d_total_error.backward()
            discriminator_optimizer.step()

            # Train Generator: maximize log(D(G(z)))
            generator_optimizer.zero_grad()
            g_fake_seed = sample_noise(batch_size, noise_size).type(
               dtype)
            fake_images = generator(g_fake_seed)

            # Generator wants discriminator to think fake images are
               real
            gen_logits_fake = discriminator(fake_images)
            g_error = generator_loss(gen_logits_fake)
            g_error.backward()
            generator_optimizer.step()

            if (iter_count % show_every == 0):
```

```
                print(f'Iter: {iter_count}, D: {d_total_error.item()
                    :.4}, '
                      f'G: {g_error.item():.4}')

            iter_count += 1
```

Listing 8: GAN training loop implementation

What this code does: This implements the core GAN training loop, alternating between discriminator and generator updates in an adversarial fashion.

What we learn:

- GANs require careful balance between generator and discriminator
- The training is inherently unstable and requires techniques like feature matching
- The discriminator must be updated more frequently than the generator
- Proper loss functions and optimizers are crucial for convergence

4.3 Denoising Diffusion Probabilistic Models (DDPMs)

DDPMs represent the latest advancement in generative modeling, using a denoising process to generate high-quality samples.

4.3.1 Forward Process Implementation

```
def q_sample(self, x_start, t, noise=None):
    """
    Forward noising process for DDPM.
    This function adds noise to clean images according to a predefined
        schedule.
    """
    if noise is None:
        noise = torch.randn_like(x_start)

    sqrt_alphas_cumprod_t = extract(self.sqrt_alphas_cumprod, t,
        x_start.shape)
    sqrt_one_minus_alphas_cumprod_t = extract(
        self.sqrt_one_minus_alphas_cumprod, t, x_start.shape
    )

    return sqrt_alphas_cumprod_t * x_start +
        sqrt_one_minus_alphas_cumprod_t * noise
```

Listing 9: DDPM forward noising process

4.3.2 U-Net Architecture for Denoising

```
class Unet(nn.Module):
    """
    U-Net architecture for DDPM denoising.
    This network takes noisy images and timesteps as input
    and predicts the noise to be removed.
    """
```

```
    def __init__(self, dim, condition_dim, dim_mults=(1, 2, 4, 8)):
        super().__init__()
        self.dim = dim

        # Time embedding
        time_dim = dim * 4
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(dim),
            nn.Linear(dim, time_dim),
            nn.GELU(),
            nn.Linear(time_dim, time_dim),
        )

        # Initial projection
        self.init_conv = nn.Conv2d(3, dim, 7, padding=3)

        # Downsampling
        self.downs = nn.ModuleList([])
        self.ups = nn.ModuleList([])

        for ind, dim_mult in enumerate(dim_mults):
            is_last = ind >= (len(dim_mults) - 1)

            self.downs.append(nn.ModuleList([
                ResnetBlock(dim, dim * dim_mult, time_emb_dim=time_dim)
                    ,
                ResnetBlock(dim * dim_mult, dim * dim_mult,
                    time_emb_dim=time_dim),
                Downsample(dim * dim_mult) if not is_last else nn.
                    Identity()
            ]))

        # Middle
        mid_dim = dim * dim_mults[-1]
        self.mid_block1 = ResnetBlock(mid_dim, mid_dim, time_emb_dim=
            time_dim)
        self.mid_attn = Residual(PreNorm(mid_dim, Attention(mid_dim)))
        self.mid_block2 = ResnetBlock(mid_dim, mid_dim, time_emb_dim=
            time_dim)

        # Upsampling
        for ind, dim_mult in enumerate(reversed(dim_mults[1:])):
            is_last = ind >= (len(dim_mults) - 1)

            self.ups.append(nn.ModuleList([
                Upsample(dim * dim_mult),
                ResnetBlock(dim * dim_mult * 2, dim * dim_mult,
                    time_emb_dim=time_dim),
                ResnetBlock(dim * dim_mult, dim * dim_mult,
                    time_emb_dim=time_dim),
            ]))

        self.final_conv = nn.Conv2d(dim, 3, 1)

    def forward(self, x, time, **kwargs):
        x = self.init_conv(x)

        t = self.time_mlp(time)
```

```
        h = []

        # Downsampling
        for block1, block2, downsample in self.downs:
            x = block1(x, t)
            x = block2(x, t)
            h.append(x)
            x = downsample(x)

        # Middle
        x = self.mid_block1(x, t)
        x = self.mid_attn(x)
        x = self.mid_block2(x, t)

        # Upsampling
        for upsample, block1, block2 in self.ups:
            x = upsample(x)
            x = torch.cat((x, h.pop()), dim=1)
            x = block1(x, t)
            x = block2(x, t)

        return self.final_conv(x)
```

Listing 10: U-Net implementation for DDPM

What this code does: This implements a U-Net architecture specifically designed for DDPM denoising, with time embeddings and skip connections.

What we learn:

- DDPMs use a reverse diffusion process to generate samples
- U-Net architecture is ideal for image-to-image tasks
- Time embeddings allow the network to understand the noise level
- Skip connections preserve fine-grained details

5 Advanced Code Analysis and Learning Outcomes

5.1 Optimization Strategies and Implementation

5.1.1 Adam Optimizer Implementation

```
def get_optimizer(model, lr=1e-3, beta1=0.5, beta2=0.999):
    """
    Create Adam optimizer with specific parameters for GAN training.
    The reduced beta1 helps with GAN stability.
    """
    return torch.optim.Adam(model.parameters(), lr=lr, betas=(beta1,
        beta2))
```

Listing 11: Adam optimizer for GAN training

What this code does: This creates an Adam optimizer with parameters specifically tuned for GAN training, where the reduced β_1 helps with training stability.

What we learn: Different optimization problems require different hyperparameters, and GAN training benefits from modified Adam parameters.

5.2 Loss Function Analysis

5.2.1 Binary Cross-Entropy Loss Implementation

```python
def bce_loss(input, target):
    """
    Numerically stable implementation of binary cross-entropy loss.
    This prevents numerical issues with log(0) computations.
    """
    neg_abs = -input.abs()
    loss = input.clamp(min=0) - input * target + (1 + neg_abs.exp()).
       log()
    return loss.mean()

def discriminator_loss(logits_real, logits_fake):
    """
    Computes the discriminator loss for GAN training.
    The discriminator should classify real images as real (1)
    and fake images as fake (0).
    """
    batch_size = logits_real.size(0)

    # Labels for real and fake data
    true_labels = torch.ones(batch_size).type(dtype)
    fake_labels = torch.zeros(batch_size).type(dtype)

    # Compute losses
    real_loss = bce_loss(logits_real.squeeze(), true_labels)
    fake_loss = bce_loss(logits_fake.squeeze(), fake_labels)

    return real_loss + fake_loss

def generator_loss(logits_fake):
    """
    Computes the generator loss for GAN training.
    The generator wants the discriminator to classify
    fake images as real (1).
    """
    batch_size = logits_fake.size(0)
    true_labels = torch.ones(batch_size).type(dtype)

    return bce_loss(logits_fake.squeeze(), true_labels)
```

Listing 12: Numerically stable BCE loss

What this code does: These functions implement the core loss functions for GAN training with numerical stability considerations.

What we learn:

- Numerical stability is crucial in loss function implementation
- GAN losses have specific interpretations related to the adversarial game
- Proper label assignment is essential for correct training

5.3 Performance Metrics and Evaluation

5.3.1 Model Evaluation Framework

```python
def evaluate_model(model, data_loader, device):
    """
    Comprehensive model evaluation with multiple metrics.
    This function computes accuracy, precision, recall, and F1-score.
    """
    model.eval()
    correct = 0
    total = 0
    all_predictions = []
    all_labels = []

    with torch.no_grad():
        for data, target in data_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)

            # Accuracy computation
            _, predicted = torch.max(output.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

            # Store for detailed metrics
            all_predictions.extend(predicted.cpu().numpy())
            all_labels.extend(target.cpu().numpy())

    accuracy = 100 * correct / total

    # Compute additional metrics
    from sklearn.metrics import classification_report, confusion_matrix

    print(f'Accuracy: {accuracy:.2f}%')
    print('\nClassification Report:')
    print(classification_report(all_labels, all_predictions))

    return accuracy, all_predictions, all_labels
```

Listing 13: Comprehensive model evaluation

What this code does: This implements a comprehensive evaluation framework that computes multiple performance metrics beyond simple accuracy.

What we learn: Proper evaluation requires multiple metrics to understand model performance comprehensively.

6 Advanced Topics and Extensions

6.1 Attention Mechanisms

```python
class SelfAttention(nn.Module):
    """
    Self-attention mechanism for capturing long-range dependencies.
    This implementation shows how attention can improve model
       performance.
```

```
    """
    def __init__(self, embed_dim, num_heads):
        super(SelfAttention, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        assert self.head_dim * num_heads == embed_dim

        self.query = nn.Linear(embed_dim, embed_dim)
        self.key = nn.Linear(embed_dim, embed_dim)
        self.value = nn.Linear(embed_dim, embed_dim)
        self.out = nn.Linear(embed_dim, embed_dim)

    def forward(self, x):
        batch_size, seq_len, embed_dim = x.size()

        # Compute Q, K, V
        Q = self.query(x).view(batch_size, seq_len, self.num_heads,
            self.head_dim)
        K = self.key(x).view(batch_size, seq_len, self.num_heads, self.
            head_dim)
        V = self.value(x).view(batch_size, seq_len, self.num_heads,
            self.head_dim)

        # Transpose for attention computation
        Q = Q.transpose(1, 2)
        K = K.transpose(1, 2)
        V = V.transpose(1, 2)

        # Scaled dot-product attention
        attention_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.
            sqrt(self.head_dim)
        attention_weights = F.softmax(attention_scores, dim=-1)

        # Apply attention to values
        attended_values = torch.matmul(attention_weights, V)

        # Concatenate heads and apply final linear transformation
        attended_values = attended_values.transpose(1, 2).contiguous().
            view(
             batch_size, seq_len, embed_dim)

        output = self.out(attended_values)
        return output
```

Listing 14: Self-attention implementation

What this code does: This implements multi-head self-attention, a key component of transformer architectures.

What we learn: Attention mechanisms allow models to focus on relevant parts of the input and capture long-range dependencies.

6.2 Transfer Learning Implementation

```
def setup_transfer_learning(model_name='resnet18', num_classes=10,
    freeze_features=True):
```

```
    """
    Setup transfer learning from pre-trained model.
    This demonstrates how to leverage pre-trained models for new tasks.
    """
    # Load pre-trained model
    model = torchvision.models.resnet18(pretrained=True)

    # Freeze feature extraction layers if specified
    if freeze_features:
        for param in model.parameters():
            param.requires_grad = False

    # Replace final layer for new task
    num_features = model.fc.in_features
    model.fc = nn.Linear(num_features, num_classes)

    return model

def fine_tune_model(model, train_loader, val_loader, num_epochs=10):
    """
    Fine-tune pre-trained model with different learning rates
    for different parts of the network.
    """
    # Different learning rates for different parts
    feature_params = []
    classifier_params = []

    for name, param in model.named_parameters():
        if 'fc' in name:
            classifier_params.append(param)
        else:
            feature_params.append(param)

    optimizer = torch.optim.Adam([
        {'params': feature_params, 'lr': 1e-4},
        {'params': classifier_params, 'lr': 1e-3}
    ])

    # Training loop with validation
    for epoch in range(num_epochs):
        train_loss = train_epoch(model, train_loader, optimizer)
        val_acc = validate_epoch(model, val_loader)

        print(f'Epoch {epoch+1}/{num_epochs}: '
              f'Train Loss: {train_loss:.4f}, Val Acc: {val_acc:.2f}%')
```

Listing 15: Transfer learning setup

What this code does: This demonstrates transfer learning, showing how to adapt pre-trained models for new tasks.

What we learn: Transfer learning allows leveraging pre-trained features and significantly reduces training time and data requirements.

7 Challenges and Solutions

7.1 Common Implementation Pitfalls

7.1.1 Gradient Vanishing and Exploding

```python
def clip_gradients(model, max_norm=1.0):
    """
    Clip gradients to prevent exploding gradients.
    This technique is essential for training stability in deep networks
        .
    """
    total_norm = 0
    for p in model.parameters():
        if p.grad is not None:
            param_norm = p.grad.data.norm(2)
            total_norm += param_norm.item() ** 2

    total_norm = total_norm ** (1. / 2)

    clip_coef = max_norm / (total_norm + 1e-6)
    if clip_coef < 1:
        for p in model.parameters():
            if p.grad is not None:
                p.grad.data.mul_(clip_coef)

    return total_norm
```

Listing 16: Gradient clipping implementation

What this code does: This implements gradient clipping to prevent exploding gradients in deep networks.

What we learn: Gradient clipping is essential for training stability, especially in recurrent networks and GANs.

7.2 Debugging and Validation Techniques

7.2.1 Gradient Checking

```python
def gradient_check(f, x, analytic_grad, num_checks=10, h=1e-5):
    """
    Gradient checking utility to verify backpropagation implementation.
    This compares analytical gradients with numerical gradients.
    """
    for i in range(num_checks):
        ix = tuple([random.randrange(m) for m in x.shape])

        oldval = x[ix]
        x[ix] = oldval + h
        fxph = f(x)
        x[ix] = oldval - h
        fxmh = f(x)
        x[ix] = oldval

        grad_numerical = (fxph - fxmh) / (2 * h)
        grad_analytic = analytic_grad[ix]
```

```
        rel_error = abs(grad_numerical - grad_analytic) / (abs(
            grad_numerical) + abs(grad_analytic))

        print(f'numerical: {grad_numerical:.6f} analytic: {
            grad_analytic:.6f}, relative error: {rel_error:.2e}')
```

Listing 17: Gradient checking utility

What this code does: This implements gradient checking to verify the correctness of backpropagation implementations.

What we learn: Gradient checking is crucial for debugging neural network implementations and ensuring correctness.

8 Key Learning Outcomes and Insights

8.1 Theoretical Understanding

8.1.1 Mathematical Foundations

Through implementing algorithms from scratch, we gained deep insights into:

- **Gradient computation:** Understanding how the chain rule enables backpropagation
- **Optimization landscapes:** Learning about local minima, saddle points, and convergence
- **Regularization theory:** Understanding how techniques like dropout and batch normalization work
- **Information theory:** Connecting cross-entropy loss to information theory concepts

8.1.2 Architectural Insights

- **Non-linearity importance:** ReLU and other activation functions enable complex function approximation
- **Spatial inductive biases:** Convolutions exploit translation invariance in images
- **Attention mechanisms:** Self-attention captures long-range dependencies
- **Generative modeling:** GANs and diffusion models represent different approaches to generation

8.2 Practical Implementation Skills

8.2.1 NumPy Proficiency

- **Vectorized operations:** Leveraging NumPy's optimized routines for efficiency
- **Broadcasting:** Understanding how NumPy handles arrays of different shapes
- **Memory efficiency:** Writing code that minimizes memory allocation
- **Numerical stability:** Implementing algorithms that avoid numerical issues

8.2.2 PyTorch Expertise

- **Automatic differentiation:** Understanding how PyTorch computes gradients
- **GPU acceleration:** Leveraging CUDA for faster training
- **Model serialization:** Saving and loading trained models
- **Advanced optimizers:** Using Adam, RMSprop, and other optimization algorithms

8.3 Experimental Design and Analysis

8.3.1 Hyperparameter Optimization

- **Grid vs. random search:** Understanding different hyperparameter search strategies
- **Cross-validation:** Proper model selection and validation techniques
- **Learning rate scheduling:** Adaptive learning rate strategies
- **Regularization tuning:** Balancing model complexity and generalization

8.3.2 Performance Analysis

- **Training dynamics:** Understanding loss curves and convergence patterns
- **Overfitting detection:** Recognizing and mitigating overfitting
- **Model interpretation:** Visualizing learned features and attention patterns
- **Computational complexity:** Analyzing time and space complexity of algorithms

9 Future Directions and Advanced Applications

9.1 Emerging Architectures

9.1.1 Vision Transformers

The success of transformers in NLP has led to their adaptation for computer vision tasks, showing that attention mechanisms can replace convolutions entirely.

9.1.2 Neural Architecture Search

Automated methods for discovering optimal network architectures represent a significant advancement in deep learning methodology.

9.2 Advanced Generative Models

9.2.1 Improved Diffusion Models

Recent advances in diffusion models include:

- **Classifier-free guidance:** Improving sample quality without additional classifiers
- **Latent diffusion:** Operating in compressed latent spaces for efficiency
- **Conditional generation:** Text-to-image and other conditional generation tasks

9.2.2 Multimodal Models

Integration of vision and language understanding in unified architectures represents the future of AI systems.

10 Conclusion

This comprehensive journey through CS231n assignments provided invaluable insights into the theoretical foundations and practical implementation of deep learning for computer vision. The progression from basic k-NN classifiers to sophisticated diffusion models illustrated the remarkable evolution of the field and the increasing sophistication of modern architectures.

10.1 Key Achievements

- **Fundamental understanding:** Successfully implemented core algorithms from scratch
- **Mathematical insight:** Gained deep understanding of backpropagation and optimization
- **Practical skills:** Developed proficiency in modern deep learning frameworks
- **Advanced techniques:** Mastered regularization, normalization, and generative modeling
- **Research preparation:** Built foundation for advanced research in computer vision

10.2 Impact and Applications

The knowledge gained from these assignments provides a solid foundation for:

- **Research contributions:** Understanding necessary for novel research in deep learning
- **Industry applications:** Practical skills for deploying computer vision systems
- **Continued learning:** Foundation for exploring advanced topics and new architectures
- **Teaching and mentoring:** Deep understanding enables effective knowledge transfer

10.3 Final Reflection

The CS231n assignments successfully bridged the gap between academic theory and practical application, providing students with both the mathematical understanding and implementation skills necessary to contribute meaningfully to the field of computer vision and deep learning. The combination of rigorous theory and hands-on implementation created a comprehensive learning experience that will serve as a foundation for future research and development endeavors.

The evolution from basic classification to advanced generative models mirrors the rapid progress in the field itself, demonstrating how fundamental concepts build upon each other to enable increasingly sophisticated applications. This progression emphasizes the importance of understanding core principles while remaining adaptable to new developments in this rapidly evolving field.