

Summer of Science '25

Midterm Report

Jayesh Verma
SoS'2

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	4
1.1	Course Structure and Objectives	4
1.2	Implementation Philosophy	4
2	Assignment 1: Fundamental Classification Methods	4
2.1	k-Nearest Neighbors Classifier	4
2.1.1	Mathematical Foundation	4
2.1.2	Implementation Details and Code Analysis	5
2.1.3	Performance Analysis and Learning Outcomes	6
2.2	Softmax Classifier	6
2.2.1	Mathematical Formulation	6
2.2.2	Gradient Computation and Implementation	6
3	Assignment 2: Neural Networks and Regularization	7
3.1	Two-Layer Neural Network	7
3.1.1	Architecture and Forward Pass	8
3.2	Batch Normalization	9
3.2.1	Mathematical Formulation and Implementation	9
3.3	Dropout Regularization	10

4	Assignment 3: Advanced Architectures and Generative Models	11
4.1	Convolutional Neural Networks	11
4.1.1	Convolution Layer Implementation	11
4.2	Generative Adversarial Networks (GANs)	12
4.2.1	GAN Architecture and Theory	12
4.2.2	Implementation Details	13
4.3	Denoising Diffusion Probabilistic Models (DDPMs)	14
4.3.1	Forward Process Implementation	14
4.3.2	U-Net Architecture for Denoising	14
5	Advanced Code Analysis and Learning Outcomes	16
5.1	Optimization Strategies and Implementation	16
5.1.1	Adam Optimizer Implementation	16
5.2	Loss Function Analysis	17
5.2.1	Binary Cross-Entropy Loss Implementation	17
5.3	Performance Metrics and Evaluation	18
5.3.1	Model Evaluation Framework	18
6	Advanced Topics and Extensions	18
6.1	Attention Mechanisms	18
6.2	Transfer Learning Implementation	19
7	Challenges and Solutions	21
7.1	Common Implementation Pitfalls	21
7.1.1	Gradient Vanishing and Exploding	21
7.2	Debugging and Validation Techniques	21
7.2.1	Gradient Checking	21
8	Key Learning Outcomes and Insights	22
8.1	Theoretical Understanding	22
8.1.1	Mathematical Foundations	22
8.1.2	Architectural Insights	22
8.2	Practical Implementation Skills	22
8.2.1	NumPy Proficiency	22
8.2.2	PyTorch Expertise	23
8.3	Experimental Design and Analysis	23
8.3.1	Hyperparameter Optimization	23
8.3.2	Performance Analysis	23
9	Future Directions and Advanced Applications	23
9.1	Emerging Architectures	23
9.1.1	Vision Transformers	23
9.1.2	Neural Architecture Search	23
9.2	Advanced Generative Models	24
9.2.1	Improved Diffusion Models	24
9.2.2	Multimodal Models	24

10 Conclusion	24
10.1 Key Achievements	24
10.2 Impact and Applications	24
10.3 Final Reflection	25

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:

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

2.1.2 Implementation Details and Code Analysis

We implemented three computational approaches with increasing efficiency:

```

1 def compute_distances_two_loops(self, X):
2     """
3     Compute distances using nested loops for educational clarity.
4     This implementation helps understand the fundamental operation
5     but is computationally inefficient for large datasets.
6     """
7     num_test = X.shape[0]
8     num_train = self.X_train.shape[0]
9     dists = np.zeros((num_test, num_train))
10
11    for i in range(num_test):
12        for j in range(num_train):
13            # Compute L2 distance between test sample i and train
14            # sample j
15            dists[i, j] = np.sqrt(np.sum((X[i] - self.X_train[j]) ** 2))
16
17    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:

```

1 def compute_distances_no_loops(self, X):
2     """
3     Fully vectorized distance computation using broadcasting.
4     This implementation leverages NumPy's optimized operations
5     for dramatic performance improvements.
6     """
7     # Expand dimensions for broadcasting
8     # X: (num_test, D), X_train: (num_train, D)
9
10    # Compute squared distances using the identity:
11    # ||a - b||^2 = ||a||^2 + ||b||^2 - 2*a*b
12    test_sum = np.sum(X**2, axis=1, keepdims=True) # (num_test, 1)
13    train_sum = np.sum(self.X_train**2, axis=1) # (num_train,)
14    cross_term = 2 * np.dot(X, self.X_train.T) # (num_test,
15    num_train)
16
17    dists = np.sqrt(test_sum + train_sum - cross_term)
18    return dists

```

Listing 2: Vectorized k-NN distance computation

What this code does: This vectorized implementation uses the mathematical identity $\|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2a^T b$ 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:

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

where $f_j = \mathbf{w}_j^T \mathbf{x}$ represents the score for class j .

The cross-entropy loss is:

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

2.2.2 Gradient Computation and Implementation

The gradient with respect to weights is:

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

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

```

1 def softmax_loss_vectorized(W, X, y, reg):
2     """
3     Compute softmax loss and gradient using vectorized operations.
4     This implementation demonstrates numerical stability techniques
5     and efficient gradient computation.
6     """

```

```
7 num_train = X.shape[0]
8
9 # Forward pass: compute scores and probabilities
10 scores = X.dot(W) # (N, C)
11
12 # Numerical stability: subtract max for each sample
13 scores -= np.max(scores, axis=1, keepdims=True)
14
15 # Compute probabilities
16 exp_scores = np.exp(scores)
17 probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
18
19 # Compute loss
20 correct_class_probs = probs[np.arange(num_train), y]
21 data_loss = -np.sum(np.log(correct_class_probs)) / num_train
22 reg_loss = reg * np.sum(W * W)
23 loss = data_loss + reg_loss
24
25 # Backward pass: compute gradient
26 dscores = probs.copy()
27 dscores[np.arange(num_train), y] -= 1
28 dscores /= num_train
29
30 dW = X.T.dot(dscores) + 2 * reg * W
31
32 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:

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

$$\text{scores} = hW_2 + b_2 \quad (6)$$

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

```

1 def loss(self, X, y=None, reg=0.0):
2     """
3     Compute loss and gradients for two-layer neural network.
4     This implementation demonstrates the forward and backward passes
5     of a simple neural network with ReLU activation.
6     """
7     W1, b1 = self.params['W1'], self.params['b1']
8     W2, b2 = self.params['W2'], self.params['b2']
9     N, D = X.shape
10
11     # Forward pass
12     # First layer: linear transformation + ReLU activation
13     z1 = X.dot(W1) + b1 # (N, H)
14     h1 = np.maximum(0, z1) # ReLU activation
15
16     # Second layer: linear transformation
17     scores = h1.dot(W2) + b2 # (N, C)
18
19     if y is None:
20         return scores
21
22     # Compute loss using softmax and cross-entropy
23     exp_scores = np.exp(scores - np.max(scores, axis=1, keepdims=True))
24     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
25
26     data_loss = -np.sum(np.log(probs[np.arange(N), y])) / N
27     reg_loss = 0.5 * reg * (np.sum(W1**2) + np.sum(W2**2))
28     loss = data_loss + reg_loss
29
30     # Backward pass: compute gradients using chain rule
31     # Output layer gradients
32     dscores = probs.copy()
33     dscores[np.arange(N), y] -= 1
34     dscores /= N
35
36     # Second layer gradients
37     dW2 = h1.T.dot(dscores) + reg * W2
38     db2 = np.sum(dscores, axis=0)
39
40     # Hidden layer gradients (backprop through ReLU)
41     dh1 = dscores.dot(W2.T)
42     dz1 = dh1.copy()
43     dz1[z1 <= 0] = 0 # ReLU derivative
44
45     # First layer gradients
46     dW1 = X.T.dot(dz1) + reg * W1
47     db1 = np.sum(dz1, axis=0)
48
49     grads = {'W1': dW1, 'b1': db1, 'W2': dW2, 'b2': db2}

```



```
50 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 $\mathcal{B} = \{x_1, \dots, x_m\}$:

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

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

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

$$y_i = \gamma \hat{x}_i + \beta \quad (10)$$

```
1 def batchnorm_forward(x, gamma, beta, bn_param):
2     """
3     Forward pass for batch normalization.
4     This implementation shows how to normalize activations
5     to improve training stability and speed.
6     """
7     mode = bn_param['mode']
8     eps = bn_param.get('eps', 1e-5)
9     momentum = bn_param.get('momentum', 0.9)
10
11     N, D = x.shape
12     running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.
13                                dtype))
14     running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype
15                                ))
16
17     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) *
30             sample_mean
31         running_var = momentum * running_var + (1 - momentum) *
32             sample_var
33
34         cache = (x, x_hat, sample_mean, sample_var, gamma, beta, eps)
35
36     elif mode == 'test':
37         # Use running statistics during inference
38         x_hat = (x - running_mean) / np.sqrt(running_var + eps)
39         out = gamma * x_hat + beta
40
41     bn_param['running_mean'] = running_mean
42     bn_param['running_var'] = running_var
43
44     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

```

```
9     if 'seed' in dropout_param:
10         np.random.seed(dropout_param['seed'])
11
12     mask = None
13     out = None
14
15     if mode == 'train':
16         # Generate random mask and scale by 1/p (inverted dropout)
17         mask = (np.random.rand(*x.shape) < p) / p
18         out = x * mask
19     elif mode == 'test':
20         # No dropout during inference
21         out = x
22
23     cache = (dropout_param, mask)
24     out = out.astype(x.dtype, copy=False)
25
26     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

```
1 def conv_forward_naive(x, w, b, conv_param):
2     """
3     Naive implementation of convolution forward pass.
4     This educational implementation shows the fundamental
5     convolution operation without optimizations.
6     """
7     stride, pad = conv_param['stride'], conv_param['pad']
8     N, C, H, W = x.shape
9     F, C, HH, WW = w.shape
10
```

```

11 # Add padding to input
12 x_padded = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)),
13                    mode='constant', constant_values=0)
14
15 # Compute output dimensions
16 H_out = 1 + (H + 2 * pad - HH) // stride
17 W_out = 1 + (W + 2 * pad - WW) // stride
18
19 # Initialize output
20 out = np.zeros((N, F, H_out, W_out))
21
22 # Perform convolution
23 for n in range(N):
24     for f in range(F):
25         for i in range(H_out):
26             for j in range(W_out):
27                 h_start = i * stride
28                 h_end = h_start + HH
29                 w_start = j * stride
30                 w_end = w_start + WW
31
32                 # Extract receptive field and compute convolution
33                 receptive_field = x_padded[n, :, h_start:h_end,
34                                           w_start:w_end]
35                 out[n, f, i, j] = np.sum(receptive_field * w[f]) +
36                                         b[f]
37
38 cache = (x, w, b, conv_param)
39 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:

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

4.2.2 Implementation Details

```

1 def train_gan(discriminator, generator, discriminator_optimizer,
2               generator_optimizer, discriminator_loss, generator_loss,
3               show_every=250, batch_size=128, noise_size=100,
4               num_epochs=10):
5     """
6     Main GAN training loop implementing the adversarial training
7     process.
8     This function alternates between training the discriminator and
9     generator.
10    """
11    iter_count = 0
12    for epoch in range(num_epochs):
13        for x, _ in loader_train:
14            if len(x) != batch_size:
15                continue
16
17            # Train Discriminator: maximize log(D(x)) + log(1 - D(G(z)))
18            discriminator_optimizer.zero_grad()
19
20            # Real data forward pass
21            real_data = x.type(dtype)
22            logits_real = discriminator(2 * (real_data - 0.5))
23
24            # Fake data forward pass
25            g_fake_seed = sample_noise(batch_size, noise_size).type(
26                dtype)
27            fake_images = generator(g_fake_seed)
28            logits_fake = discriminator(fake_images.detach())
29
30            # Compute discriminator loss and update
31            d_total_error = discriminator_loss(logits_real, logits_fake)
32            d_total_error.backward()
33            discriminator_optimizer.step()
34
35            # Train Generator: maximize log(D(G(z)))
36            generator_optimizer.zero_grad()
37            g_fake_seed = sample_noise(batch_size, noise_size).type(
38                dtype)
39            fake_images = generator(g_fake_seed)
40
41            # Generator wants discriminator to think fake images are
42            # real
43            gen_logits_fake = discriminator(fake_images)
44            g_error = generator_loss(gen_logits_fake)
45            g_error.backward()
46            generator_optimizer.step()
47
48            if (iter_count % show_every == 0):

```

```

43         print(f'Iter: {iter_count}, D: {d_total_error.item()
44               :.4}, '
45               f'G: {g_error.item():.4}')
46         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

```

1 def q_sample(self, x_start, t, noise=None):
2     """
3     Forward noising process for DDPM.
4     This function adds noise to clean images according to a predefined
5     schedule.
6     """
7     if noise is None:
8         noise = torch.randn_like(x_start)
9
10    sqrt_alphas_cumprod_t = extract(self.sqrt_alphas_cumprod, t,
11    x_start.shape)
12    sqrt_one_minus_alphas_cumprod_t = extract(
13        self.sqrt_one_minus_alphas_cumprod, t, x_start.shape
14    )
15
16    return sqrt_alphas_cumprod_t * x_start +
17        sqrt_one_minus_alphas_cumprod_t * noise

```

Listing 9: DDPM forward noising process

4.3.2 U-Net Architecture for Denoising

```

1 class Unet(nn.Module):
2     """
3     U-Net architecture for DDPM denoising.
4     This network takes noisy images and timesteps as input
5     and predicts the noise to be removed.
6     """

```

```

7   def __init__(self, dim, condition_dim, dim_mults=(1, 2, 4, 8)):
8       super().__init__()
9       self.dim = dim
10
11      # Time embedding
12      time_dim = dim * 4
13      self.time_mlp = nn.Sequential(
14          SinusoidalPositionEmbeddings(dim),
15          nn.Linear(dim, time_dim),
16          nn.GELU(),
17          nn.Linear(time_dim, time_dim),
18      )
19
20      # Initial projection
21      self.init_conv = nn.Conv2d(3, dim, 7, padding=3)
22
23      # Downsampling
24      self.downs = nn.ModuleList([])
25      self.ups = nn.ModuleList([])
26
27      for ind, dim_mult in enumerate(dim_mults):
28          is_last = ind >= (len(dim_mults) - 1)
29
30          self.downs.append(nn.ModuleList([
31              ResnetBlock(dim, dim * dim_mult, time_emb_dim=time_dim)
32              ,
33              ResnetBlock(dim * dim_mult, dim * dim_mult,
34                  time_emb_dim=time_dim),
35              Downsample(dim * dim_mult) if not is_last else nn.
36                  Identity()
37          ]))
38
39      # Middle
40      mid_dim = dim * dim_mults[-1]
41      self.mid_block1 = ResnetBlock(mid_dim, mid_dim, time_emb_dim=
42          time_dim)
43      self.mid_attn = Residual(PreNorm(mid_dim, Attention(mid_dim)))
44      self.mid_block2 = ResnetBlock(mid_dim, mid_dim, time_emb_dim=
45          time_dim)
46
47      # Upsampling
48      for ind, dim_mult in enumerate(reversed(dim_mults[1:])):
49          is_last = ind >= (len(dim_mults) - 1)
50
51          self.ups.append(nn.ModuleList([
52              Upsample(dim * dim_mult),
53              ResnetBlock(dim * dim_mult * 2, dim * dim_mult,
54                  time_emb_dim=time_dim),
55              ResnetBlock(dim * dim_mult, dim * dim_mult,
56                  time_emb_dim=time_dim),
57          ]))
58
59      self.final_conv = nn.Conv2d(dim, 3, 1)
60
61      def forward(self, x, time, **kwargs):
62          x = self.init_conv(x)
63
64          t = self.time_mlp(time)

```

```

58     h = []
59
60
61     # Downsampling
62     for block1, block2, downsample in self.downs:
63         x = block1(x, t)
64         x = block2(x, t)
65         h.append(x)
66         x = downsample(x)
67
68     # Middle
69     x = self.mid_block1(x, t)
70     x = self.mid_attn(x)
71     x = self.mid_block2(x, t)
72
73     # Upsampling
74     for upsample, block1, block2 in self.ups:
75         x = upsample(x)
76         x = torch.cat((x, h.pop()), dim=1)
77         x = block1(x, t)
78         x = block2(x, t)
79
80     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

```

1 def get_optimizer(model, lr=1e-3, beta1=0.5, beta2=0.999):
2     """
3     Create Adam optimizer with specific parameters for GAN training.
4     The reduced beta1 helps with GAN stability.
5     """
6     return torch.optim.Adam(model.parameters(), lr=lr, betas=(beta1,
7                             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

```
1 def bce_loss(input, target):
2     """
3     Numerically stable implementation of binary cross-entropy loss.
4     This prevents numerical issues with log(0) computations.
5     """
6     neg_abs = -input.abs()
7     loss = input.clamp(min=0) - input * target + (1 + neg_abs.exp()).
8         log()
9     return loss.mean()
10
11 def discriminator_loss(logits_real, logits_fake):
12     """
13     Computes the discriminator loss for GAN training.
14     The discriminator should classify real images as real (1)
15     and fake images as fake (0).
16     """
17     batch_size = logits_real.size(0)
18
19     # Labels for real and fake data
20     true_labels = torch.ones(batch_size).type(dtype)
21     fake_labels = torch.zeros(batch_size).type(dtype)
22
23     # Compute losses
24     real_loss = bce_loss(logits_real.squeeze(), true_labels)
25     fake_loss = bce_loss(logits_fake.squeeze(), fake_labels)
26
27     return real_loss + fake_loss
28
29 def generator_loss(logits_fake):
30     """
31     Computes the generator loss for GAN training.
32     The generator wants the discriminator to classify
33     fake images as real (1).
34     """
35     batch_size = logits_fake.size(0)
36     true_labels = torch.ones(batch_size).type(dtype)
37
38     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

```
1 def evaluate_model(model, data_loader, device):
2     """
3     Comprehensive model evaluation with multiple metrics.
4     This function computes accuracy, precision, recall, and F1-score.
5     """
6     model.eval()
7     correct = 0
8     total = 0
9     all_predictions = []
10    all_labels = []
11
12    with torch.no_grad():
13        for data, target in data_loader:
14            data, target = data.to(device), target.to(device)
15            output = model(data)
16
17            # Accuracy computation
18            _, predicted = torch.max(output.data, 1)
19            total += target.size(0)
20            correct += (predicted == target).sum().item()
21
22            # Store for detailed metrics
23            all_predictions.extend(predicted.cpu().numpy())
24            all_labels.extend(target.cpu().numpy())
25
26    accuracy = 100 * correct / total
27
28    # Compute additional metrics
29    from sklearn.metrics import classification_report, confusion_matrix
30
31    print(f'Accuracy: {accuracy:.2f}%')
32    print('\nClassification Report:')
33    print(classification_report(all_labels, all_predictions))
34
35    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

```
1 class SelfAttention(nn.Module):
2     """
3     Self-attention mechanism for capturing long-range dependencies.
4     This implementation shows how attention can improve model
5     performance.
```

```

5  """
6  def __init__(self, embed_dim, num_heads):
7      super(SelfAttention, self).__init__()
8      self.embed_dim = embed_dim
9      self.num_heads = num_heads
10     self.head_dim = embed_dim // num_heads
11
12     assert self.head_dim * num_heads == embed_dim
13
14     self.query = nn.Linear(embed_dim, embed_dim)
15     self.key = nn.Linear(embed_dim, embed_dim)
16     self.value = nn.Linear(embed_dim, embed_dim)
17     self.out = nn.Linear(embed_dim, embed_dim)
18
19     def forward(self, x):
20         batch_size, seq_len, embed_dim = x.size()
21
22         # Compute Q, K, V
23         Q = self.query(x).view(batch_size, seq_len, self.num_heads,
24                                 self.head_dim)
25         K = self.key(x).view(batch_size, seq_len, self.num_heads, self.
26                               head_dim)
27         V = self.value(x).view(batch_size, seq_len, self.num_heads,
28                                 self.head_dim)
29
30         # Transpose for attention computation
31         Q = Q.transpose(1, 2)
32         K = K.transpose(1, 2)
33         V = V.transpose(1, 2)
34
35         # Scaled dot-product attention
36         attention_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.
37             sqrt(self.head_dim)
38         attention_weights = F.softmax(attention_scores, dim=-1)
39
40         # Apply attention to values
41         attended_values = torch.matmul(attention_weights, V)
42
43         # Concatenate heads and apply final linear transformation
44         attended_values = attended_values.transpose(1, 2).contiguous().
45             view(
46                 batch_size, seq_len, embed_dim)
47
48         output = self.out(attended_values)
49         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

```

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

```

```

2      """
3      Setup transfer learning from pre-trained model.
4      This demonstrates how to leverage pre-trained models for new tasks.
5      """
6      # Load pre-trained model
7      model = torchvision.models.resnet18(pretrained=True)
8
9      # Freeze feature extraction layers if specified
10     if freeze_features:
11         for param in model.parameters():
12             param.requires_grad = False
13
14     # Replace final layer for new task
15     num_features = model.fc.in_features
16     model.fc = nn.Linear(num_features, num_classes)
17
18     return model
19
20 def fine_tune_model(model, train_loader, val_loader, num_epochs=10):
21     """
22     Fine-tune pre-trained model with different learning rates
23     for different parts of the network.
24     """
25     # Different learning rates for different parts
26     feature_params = []
27     classifier_params = []
28
29     for name, param in model.named_parameters():
30         if 'fc' in name:
31             classifier_params.append(param)
32         else:
33             feature_params.append(param)
34
35     optimizer = torch.optim.Adam([
36         {'params': feature_params, 'lr': 1e-4},
37         {'params': classifier_params, 'lr': 1e-3}
38     ])
39
40     # Training loop with validation
41     for epoch in range(num_epochs):
42         train_loss = train_epoch(model, train_loader, optimizer)
43         val_acc = validate_epoch(model, val_loader)
44
45         print(f'Epoch {epoch+1}/{num_epochs}: '
46               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

```
1 def clip_gradients(model, max_norm=1.0):
2     """
3     Clip gradients to prevent exploding gradients.
4     This technique is essential for training stability in deep networks
5     """
6     total_norm = 0
7     for p in model.parameters():
8         if p.grad is not None:
9             param_norm = p.grad.data.norm(2)
10            total_norm += param_norm.item() ** 2
11
12    total_norm = total_norm ** (1. / 2)
13
14    clip_coef = max_norm / (total_norm + 1e-6)
15    if clip_coef < 1:
16        for p in model.parameters():
17            if p.grad is not None:
18                p.grad.data.mul_(clip_coef)
19
20    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

```
1 def gradient_check(f, x, analytic_grad, num_checks=10, h=1e-5):
2     """
3     Gradient checking utility to verify backpropagation implementation.
4     This compares analytical gradients with numerical gradients.
5     """
6     for i in range(num_checks):
7         ix = tuple([random.randrange(m) for m in x.shape])
8
9         oldval = x[ix]
10        x[ix] = oldval + h
11        fxph = f(x)
12        x[ix] = oldval - h
13        fxmh = f(x)
14        x[ix] = oldval
15
16        grad_numerical = (fxph - fxmh) / (2 * h)
17        grad_analytic = analytic_grad[ix]
```

```
18     rel_error = abs(grad_numerical - grad_analytic) / (abs(  
19         grad_numerical) + abs(grad_analytic))  
20  
    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.