```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confu
import matplotlib.pyplot as plt
import os
import zipfile
from PIL import Image
import warnings
warnings.filterwarnings('ignore')

np.random.seed(42)
```

```
def main():
    print("="*80)
    print("DOGS VS CATS - COMPLETE 27 EXPERIMENTS")
    print("3 Activations × 3 Initializations × 3 Optimizers = 27 Combinations")
    print("="*80)

    X_train, y_train, X_test, test_ids = extract_and_load_images(
        max_samples=5000,
        image_size=(64, 64)
    )

    X_train = X_train.astype('float32') / 255.0
    X_test = X_test.astype('float32') / 255.0

    n_train = X_train.shape[0]
    n_test = X_test.shape[0]
    X_train = X_train.reshape(n_train, -1)
    X_test = X_test.reshape(n_test, -1)

    print(f"Flattened - Train: {X_train.shape}, Test: {X_test.shape}")


    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    X_train_split, X_val, y_train_split, y_val = train_test_split(
        X_train, y_train, test_size=0.2, random_state=42, stratify=y_train
    )

    print(f"Split - Train: {X_train_split.shape[0]}, Val: {X_val.shape[0]}, Test: {X_test

    X_train_split = X_train_split.T
    y_train_split = y_train_split.reshape(1, -1)
    X_val = X_val.T
    y_val = y_val.reshape(1, -1)
    X_test_T = X_test.T

    input_size = X_train_split.shape[0]
    layer_sizes = [input_size, 256, 128, 64, 1]

    print(f"\nArchitecture: {layer_sizes}")

    activations = ['relu', 'tanh', 'leaky_relu']
    initializations = ['xavier', 'kaiming', 'random']
    optimizers_configs = [
        ('SGD', SGDOptimizer(learning_rate=0.01, momentum=0.9)),
        ('Adam', AdamOptimizer(learning_rate=0.001)),
        ('RMSprop', RMSpropOptimizer(learning_rate=0.001))
    ]

    all_results = []
```

```
all_results = {}
experiment_num = 0
total_experiments = len(activations) * len(initializations) * len(optimizers_configs)


for activation in activations:
    for initialization in initializations:
        for opt_name, optimizer in optimizers_configs:
            experiment_num += 1
            config_name = f"{activation}_{initialization}_{opt_name}"

            print(f"\n{'='*80}")
            print(f"EXPERIMENT {experiment_num}/{total_experiments}: {config_name}")
            print(f"{'='*80}")


            model = NeuralNetwork(
                layer_sizes,
                dropout_rate=0.3,
                use_batch_norm=True,
                activation=activation,
                initialization=initialization,
                l2_lambda=0.01
            )


            history = train_model_with_early_stopping(
                model, optimizer,
                X_train_split, y_train_split,
                X_val, y_val,
                epochs=100,
                batch_size=64,
                patience=15
            )


            metrics = evaluate_model(model, X_val, y_val)

            print(f"\n✓ Results: Acc={metrics['accuracy']:.4f}, "
                  f"Prec={metrics['precision']:.4f}, "
                  f"Rec={metrics['recall']:.4f}, "
                  f"F1={metrics['f1_score']:.4f}")

            all_results[config_name] = {
                'history': history,
                'metrics': metrics,
                'model': model
            }
```

```
def relu(Z):
    """ReLU activation function"""
    return np.maximum(0, Z)

def relu_derivative(Z):
    """Derivative of ReLU"""
    return (Z > 0).astype(float)

def tanh(Z):
    """Tanh activation function"""
    return np.tanh(Z)

def tanh_derivative(Z):
    """Derivative of tanh"""
    return 1 - np.tanh(Z) ** 2
```

```
def leaky_relu(Z, alpha=0.01):
    """Leaky ReLU activation function"""
    return np.where(Z > 0, Z, alpha * Z)

def leaky_relu_derivative(Z, alpha=0.01):
    """Derivative of Leaky ReLU"""
    return np.where(Z > 0, 1, alpha)

def sigmoid(Z):
    """Sigmoid activation function"""
    Z = np.clip(Z, -500, 500)
    return 1 / (1 + np.exp(-Z))

def sigmoid_derivative(A):
    """Derivative of sigmoid"""
    return A * (1 - A)
```

```
class NeuralNetwork:

    def __init__(self, layer_sizes, dropout_rate=0.3, use_batch_norm=True,
                 activation='relu', initialization='kaiming', l2_lambda=0.01):
        self.layer_sizes = layer_sizes
        self.dropout_rate = dropout_rate
        self.use_batch_norm = use_batch_norm
        self.activation = activation
        self.initialization = initialization
        self.l2_lambda = l2_lambda
        self.parameters = {}
        self.bn_parameters = {}
        self.initialize_parameters()

    def initialize_parameters(self):
        """Initialize weights using specified method"""
        np.random.seed(42)

        for l in range(1, len(self.layer_sizes)):
            if self.initialization == 'xavier':

                self.parameters[f'W{l}'] = np.random.randn(
                    self.layer_sizes[l],
                    self.layer_sizes[l-1]
                ) * np.sqrt(1.0 / self.layer_sizes[l-1])

            elif self.initialization == 'kaiming':

                self.parameters[f'W{l}'] = np.random.randn(
                    self.layer_sizes[l],
                    self.layer_sizes[l-1]
                ) * np.sqrt(2.0 / self.layer_sizes[l-1])

            else:
                self.parameters[f'W{l}'] = np.random.randn(
                    self.layer_sizes[l],
                    self.layer_sizes[l-1]
                ) * 0.01

            self.parameters[f'b{l}'] = np.zeros((self.layer_sizes[l], 1))

            if self.use_batch_norm and l < len(self.layer_sizes) - 1:
                self.bn_parameters[f'gamma{l}'] = np.ones((self.layer_sizes[l], 1))
                self.bn_parameters[f'beta{l}'] = np.zeros((self.layer_sizes[l], 1))
                self.bn_parameters[f'running_mean{l}'] = np.zeros((self.layer_sizes[l],
                self.bn_parameters[f'running_var{l}'] = np.ones((self.layer_sizes[l], 1)

    def apply_activation(self, Z):
```

```
        def apply_activation(self, Z):
            if self.activation == 'relu':
                return relu(Z)
            elif self.activation == 'tanh':
                return tanh(Z)
            elif self.activation == 'leaky_relu':
                return leaky_relu(Z)
            else:
                return relu(Z)

        def apply_activation_derivative(self, Z):
            if self.activation == 'relu':
                return relu_derivative(Z)
            elif self.activation == 'tanh':
                return tanh_derivative(Z)
            elif self.activation == 'leaky_relu':
                return leaky_relu_derivative(Z)
            else:
                return relu_derivative(Z)

        def batch_norm_forward(self, Z, layer, training=True, momentum=0.9):
            if not training:
                Z_norm = (Z - self.bn_parameters[f'running_mean{layer}']) / \
                         np.sqrt(self.bn_parameters[f'running_var{layer}'] + 1e-8)
            else:
                mean = np.mean(Z, axis=1, keepdims=True)
                var = np.var(Z, axis=1, keepdims=True)
                Z_norm = (Z - mean) / np.sqrt(var + 1e-8)

                self.bn_parameters[f'running_mean{layer}'] = \
                    momentum * self.bn_parameters[f'running_mean{layer}'] + (1 - momentum) *
                self.bn_parameters[f'running_var{layer}'] = \
                    momentum * self.bn_parameters[f'running_var{layer}'] + (1 - momentum) *

            Z_out = self.bn_parameters[f'gamma{layer}'] * Z_norm + self.bn_parameters[f'beta
            return Z_out, Z_norm

        def forward_propagation(self, X, training=True):
            cache = {'A0': X}
            L = len(self.layer_sizes) - 1

            for l in range(1, L):
                Z = np.dot(self.parameters[f'W{l}'], cache[f'A{l-1}']) + self.parameters[f'b
                cache[f'Z{l}'] = Z

                if self.use_batch_norm:
                    Z_bn, Z_norm = self.batch_norm_forward(Z, l, training)
                    cache[f'Z_norm{l}'] = Z_norm
                    cache[f'Z_bn{l}'] = Z_bn
                    A = self.apply_activation(Z_bn)
                else:
                    A = self.apply_activation(Z)

                if training and self.dropout_rate > 0:
                    D = (np.random.rand(A.shape[0], A.shape[1]) > self.dropout_rate).astype(
                    A = A * D / (1 - self.dropout_rate)
                    cache[f'D{l}'] = D

                cache[f'A{l}'] = A

            Z = np.dot(self.parameters[f'W{L}'], cache[f'A{L-1}']) + self.parameters[f'b{L}'
            A = sigmoid(Z)
            cache[f'Z{L}'] = Z
            cache[f'A{L}'] = A

            return A, cache

        def compute_cost(self, AL, Y, parameters):
```

```
        m = Y.shape[1]
        cross_entropy_cost = -np.mean(Y * np.log(AL + 1e-8) + (1 - Y) * np.log(1 - AL +

        l2_cost = 0
        L = len(self.layer_sizes) - 1
        for l in range(1, L + 1):
            l2_cost += np.sum(np.square(parameters[f'W{l}']))

        l2_cost = (self.l2_lambda / (2 * m)) * l2_cost
        total_cost = cross_entropy_cost + l2_cost
        return total_cost

    def batch_norm_backward(self, dZ_bn, Z_norm, layer, m):
        dgamma = np.sum(dZ_bn * Z_norm, axis=1, keepdims=True)
        dbeta = np.sum(dZ_bn, axis=1, keepdims=True)
        dZ_norm = dZ_bn * self.bn_parameters[f'gamma{layer}']

        var = self.bn_parameters[f'running_var{layer}']
        std = np.sqrt(var + 1e-8)

        dZ = (1.0 / m) * (1.0 / std) * (
            m * dZ_norm - np.sum(dZ_norm, axis=1, keepdims=True) -
            Z_norm * np.sum(dZ_norm * Z_norm, axis=1, keepdims=True)
        )

        return dZ, dgamma, dbeta

    def backward_propagation(self, cache, Y):
        grads = {}
        L = len(self.layer_sizes) - 1
        m = Y.shape[1]

        dZ = cache[f'A{L}'] - Y
        grads[f'dW{L}'] = np.dot(dZ, cache[f'A{L-1}'].T) / m + \
                          (self.l2_lambda / m) * self.parameters[f'W{L}']
        grads[f'db{L}'] = np.sum(dZ, axis=1, keepdims=True) / m
        dA_prev = np.dot(self.parameters[f'W{L}'].T, dZ)

        for l in reversed(range(1, L)):
            if f'D{l}' in cache:
                dA_prev = dA_prev * cache[f'D{l}'] / (1 - self.dropout_rate)

            if self.use_batch_norm:
                dZ_bn = dA_prev * self.apply_activation_derivative(cache[f'Z_bn{l}'])
                dZ, dgamma, dbeta = self.batch_norm_backward(dZ_bn, cache[f'Z_norm{l}'],
                grads[f'dgamma{l}'] = dgamma
                grads[f'dbeta{l}'] = dbeta
            else:
                dZ = dA_prev * self.apply_activation_derivative(cache[f'Z{l}'])

            grads[f'dW{l}'] = np.dot(dZ, cache[f'A{l-1}'].T) / m + \
                              (self.l2_lambda / m) * self.parameters[f'W{l}']
            grads[f'db{l}'] = np.sum(dZ, axis=1, keepdims=True) / m

            if l > 1:
                dA_prev = np.dot(self.parameters[f'W{l}'].T, dZ)

        return grads
```

```
class SGDOptimizer:
    def __init__(self, learning_rate=0.01, momentum=0.9):
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.velocity = {}
        self.name = "SGD"

    def update(self, parameters, bn_parameters, grads, use_batch_norm):
```

```
        if not self.velocity:
            for key in parameters:
                self.velocity[key] = np.zeros_like(parameters[key])
            if use_batch_norm:
                for key in bn_parameters:
                    if 'gamma' in key or 'beta' in key:
                        self.velocity[key] = np.zeros_like(bn_parameters[key])

        for key in parameters:
            if f'd{key}' in grads:
                self.velocity[key] = (self.momentum * self.velocity[key] -
                                      self.learning_rate * grads[f'd{key}'])
                parameters[key] += self.velocity[key]

        if use_batch_norm:
            for key in bn_parameters:
                if ('gamma' in key or 'beta' in key) and f'd{key}' in grads:
                    self.velocity[key] = (self.momentum * self.velocity[key] -
                                          self.learning_rate * grads[f'd{key}'])
                    bn_parameters[key] += self.velocity[key]

        return parameters, bn_parameters
```

```
class AdamOptimizer:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.learning_rate = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = {}
        self.v = {}
        self.t = 0
        self.name = "Adam"

    def update(self, parameters, bn_parameters, grads, use_batch_norm):
        if not self.m:
            for key in parameters:
                self.m[key] = np.zeros_like(parameters[key])
                self.v[key] = np.zeros_like(parameters[key])
            if use_batch_norm:
                for key in bn_parameters:
                    if 'gamma' in key or 'beta' in key:
                        self.m[key] = np.zeros_like(bn_parameters[key])
                        self.v[key] = np.zeros_like(bn_parameters[key])

        self.t += 1

        for key in parameters:
            if f'd{key}' in grads:
                self.m[key] = self.beta1 * self.m[key] + (1 - self.beta1) * grads[f'd{ke
                self.v[key] = self.beta2 * self.v[key] + (1 - self.beta2) * (grads[f'd{k

                m_corrected = self.m[key] / (1 - self.beta1 ** self.t)
                v_corrected = self.v[key] / (1 - self.beta2 ** self.t)

                parameters[key] -= self.learning_rate * m_corrected / (np.sqrt(v_correct

        if use_batch_norm:
            for key in bn_parameters:
                if ('gamma' in key or 'beta' in key) and f'd{key}' in grads:
                    self.m[key] = self.beta1 * self.m[key] + (1 - self.beta1) * grads[f'
                    self.v[key] = self.beta2 * self.v[key] + (1 - self.beta2) * (grads[f

                    m_corrected = self.m[key] / (1 - self.beta1 ** self.t)
                    v_corrected = self.v[key] / (1 - self.beta2 ** self.t)
```

```
                    bn_parameters[key] -= self.learning_rate * m_corrected / (np.sqrt(v_

        return parameters, bn_parameters
```

```
class RMSpropOptimizer:
    def __init__(self, learning_rate=0.001, decay_rate=0.9, epsilon=1e-8):
        self.learning_rate = learning_rate
        self.decay_rate = decay_rate
        self.epsilon = epsilon
        self.s = {}
        self.name = "RMSprop"

    def update(self, parameters, bn_parameters, grads, use_batch_norm):
        if not self.s:
            for key in parameters:
                self.s[key] = np.zeros_like(parameters[key])
            if use_batch_norm:
                for key in bn_parameters:
                    if 'gamma' in key or 'beta' in key:
                        self.s[key] = np.zeros_like(bn_parameters[key])

        for key in parameters:
            if f'd{key}' in grads:
                self.s[key] = (self.decay_rate * self.s[key] +
                               (1 - self.decay_rate) * (grads[f'd{key}'] ** 2))
                parameters[key] -= (self.learning_rate * grads[f'd{key}'] /
                                    (np.sqrt(self.s[key]) + self.epsilon))

        if use_batch_norm:
            for key in bn_parameters:
                if ('gamma' in key or 'beta' in key) and f'd{key}' in grads:
                    self.s[key] = (self.decay_rate * self.s[key] +
                                   (1 - self.decay_rate) * (grads[f'd{key}'] ** 2))
                    bn_parameters[key] -= (self.learning_rate * grads[f'd{key}'] /
                                           (np.sqrt(self.s[key]) + self.epsilon))

        return parameters, bn_parameters
```

```
def extract_and_load_images(max_samples=5000, image_size=(64, 64)):

    global used_sample_data
    used_sample_data = False

    if not os.path.exists('train.zip'):
        print("\ntrain.zip not found!")
        print("Creating sample data instead...")
        used_sample_data = True
        return create_sample_data(max_samples, image_size)

    print("\nExtracting train.zip...")
    train_dir = 'train_extracted'
    if not os.path.exists(train_dir):
        os.makedirs(train_dir)
        with zipfile.ZipFile('train.zip', 'r') as zip_ref:
            zip_ref.extractall(train_dir)
        print(f"   Extracted to {train_dir}/")
    else:
        print(f"   Already extracted")

    train_files = []
    for root, dirs, files in os.walk(train_dir):
        for file in files:
            if file.endswith(('.jpg', '.jpeg', '.png')):
                train_files.append(os.path.join(root, file))
```

```
print(f"  Found {len(train_files)} training images")

if len(train_files) > max_samples:
    print(f"   Limiting to {max_samples} samples...")
    np.random.seed(42)
    train_files = np.random.choice(train_files, max_samples, replace=False)

print(f"\nLoading {len(train_files)} training images...")
X_train_list = []
y_train_list = []

for idx, img_path in enumerate(train_files):
    try:
        img = Image.open(img_path).convert('RGB')
        img = img.resize(image_size)
        img_array = np.array(img)

        filename = os.path.basename(img_path)
        if filename.startswith('dog'):
            label = 1
        elif filename.startswith('cat'):
            label = 0
        else:
            continue

        X_train_list.append(img_array)
        y_train_list.append(label)

        if (idx + 1) % 1000 == 0:
            print(f"  Loaded {idx + 1}/{len(train_files)} images...")

    except Exception as e:
        continue

X_train = np.array(X_train_list)
y_train = np.array(y_train_list)

print(f"\Training set: {X_train.shape}")
print(f"    Cats: {np.sum(y_train==0)}, Dogs: {np.sum(y_train==1)}")

X_test = None
test_ids = None

if os.path.exists('test.zip'):
    print("\nExtracting test.zip...")
    test_dir = 'test_extracted'
    if not os.path.exists(test_dir):
        os.makedirs(test_dir)
        with zipfile.ZipFile('test.zip', 'r') as zip_ref:
            zip_ref.extractall(test_dir)
        print(f"Extracted")
    else:
        print(f"Already extracted")

    test_files = []
    for root, dirs, files in os.walk(test_dir):
        for file in files:
            if file.endswith(('.jpg', '.jpeg', '.png')):
                test_files.append(os.path.join(root, file))

    print(f" Found {len(test_files)} test images")

    if len(test_files) > 1000:
        print(f" Limiting to 1000 test samples...")
        test_files = sorted(test_files)[:1000]

    print(f"\nLoading {len(test_files)} test images...")
```

```
            X_test_list = []
            test_ids = []

            for idx, img_path in enumerate(test_files):
                try:
                    img = Image.open(img_path).convert('RGB')
                    img = img.resize(image_size)
                    img_array = np.array(img)

                    filename = os.path.basename(img_path)
                    img_id = int(filename.split('.')[0])

                    X_test_list.append(img_array)
                    test_ids.append(img_id)

                    if (idx + 1) % 200 == 0:
                        print(f"    Loaded {idx + 1}/{len(test_files)} images...")

                except Exception as e:
                    continue

            X_test = np.array(X_test_list)
            test_ids = np.array(test_ids)
            print(f"\n    Test set: {X_test.shape}")
        else:
            print("\ntest.zip not found")
            X_test = X_train[:100]
            test_ids = np.arange(1, 101)


        return X_train, y_train, X_test, test_ids


    def create_sample_data(max_samples=5000, image_size=(64, 64)):
        print("\nCreating sample data...")
        np.random.seed(42)

        X_train = np.random.randint(0, 256, (max_samples, *image_size, 3), dtype=np.uint8)
        y_train = np.random.randint(0, 2, max_samples)
        X_test = np.random.randint(0, 256, (500, *image_size, 3), dtype=np.uint8)
        test_ids = np.arange(1, 501)

        print(f"Sample data: Train {X_train.shape}, Test {X_test.shape}")

        return X_train, y_train, X_test, test_ids
```

```
    def create_mini_batches(X, Y, batch_size):
        m = X.shape[1]
        mini_batches = []

        permutation = list(np.random.permutation(m))
        shuffled_X = X[:, permutation]
        shuffled_Y = Y[:, permutation].reshape((1, m))

        num_complete_batches = m // batch_size

        for k in range(num_complete_batches):
            mini_batch_X = shuffled_X[:, k * batch_size:(k + 1) * batch_size]
            mini_batch_Y = shuffled_Y[:, k * batch_size:(k + 1) * batch_size]
            mini_batches.append((mini_batch_X, mini_batch_Y))

        if m % batch_size != 0:
            mini_batch_X = shuffled_X[:, num_complete_batches * batch_size:]
            mini_batch_Y = shuffled_Y[:, num_complete_batches * batch_size:]
            mini_batches.append((mini_batch_X, mini_batch_Y))
```

```
    return mini_batches


def train_model_with_early_stopping(model, optimizer, X_train, y_train, X_val, y_val,
                                    epochs=100, batch_size=64, patience=15):

    print(f"\nTraining: {optimizer.name}, Act: {model.activation}, Init: {model.initiali
    print(f"Epochs: {epochs}, Batch: {batch_size}, Patience: {patience}")
    print("-" * 70)

    history = {
        'train_cost': [],
        'val_cost': [],
        'train_acc': [],
        'val_acc': []
    }

    best_val_cost = float('inf')
    patience_counter = 0
    best_parameters = None
    best_bn_parameters = None

    for epoch in range(epochs):
        epoch_cost = 0
        mini_batches = create_mini_batches(X_train, y_train, batch_size)

        for mini_batch_X, mini_batch_Y in mini_batches:
            AL, cache = model.forward_propagation(mini_batch_X, training=True)
            cost = model.compute_cost(AL, mini_batch_Y, model.parameters)
            epoch_cost += cost

            grads = model.backward_propagation(cache, mini_batch_Y)
            model.parameters, model.bn_parameters = optimizer.update(
                model.parameters, model.bn_parameters, grads, model.use_batch_norm
            )

epoch_cost /= len(mini_batches)
history['train_cost'].append(epoch_cost)

AL_val, _ = model.forward_propagation(X_val, training=False)
val_cost = model.compute_cost(AL_val, y_val, model.parameters)
history['val_cost'].append(val_cost)


train_pred = (model.forward_propagation(X_train, training=False)[0] > 0.5).astyp
train_acc = np.mean(train_pred == y_train)
history['train_acc'].append(train_acc)

val_pred = (AL_val > 0.5).astype(int)
val_acc = np.mean(val_pred == y_val)
history['val_acc'].append(val_acc)


if val_cost < best_val_cost:
    best_val_cost = val_cost
    patience_counter = 0
    best_parameters = {k: v.copy() for k, v in model.parameters.items()}
    best_bn_parameters = {k: v.copy() for k, v in model.bn_parameters.items()}
else:
    patience_counter += 1

if (epoch + 1) % 10 == 0 or epoch == 0:
    print(f"Ep {epoch+1:3d} | TrL: {epoch_cost:.4f} TrA: {train_acc:.4f} | "
          f"VaL: {val_cost:.4f} VaA: {val_acc:.4f} | Pat: {patience_counter}/{pa

if patience_counter >= patience:
    print(f"\nEarly stop at epoch {epoch+1}, Best: {best_val_cost:.4f}")
```

```
                model.parameters = best_parameters
                model.bn_parameters = best_bn_parameters
                break

        print("-" * 70)

        return history
```

```
def evaluate_model(model, X_test, y_test):
    AL, _ = model.forward_propagation(X_test, training=False)
    y_pred = (AL > 0.5).astype(int).flatten()
    y_true = y_test.flatten()

    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, zero_division=0)
    recall = recall_score(y_true, y_pred, zero_division=0)
    f1 = f1_score(y_true, y_pred, zero_division=0)
    cm = confusion_matrix(y_true, y_pred)

    return {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1_score': f1,
        'confusion_matrix': cm,
        'predictions': y_pred
    }
```

```
def predict_test_set(model, X_test):
    AL, _ = model.forward_propagation(X_test, training=False)
    y_pred = (AL > 0.5).astype(int).flatten()
    return y_pred
```

```
def plot_comprehensive_results(all_results):

    fig = plt.figure(figsize=(24, 20))
    configs = list(all_results.keys())

    for idx, config in enumerate(configs[:8]):
        if idx >= 8:
            break
        history = all_results[config]['history']


        ax = plt.subplot(5, 4, idx + 1)
        ax.plot(history['train_cost'], label='Train', alpha=0.7, linewidth=1)
        ax.plot(history['val_cost'], label='Val', alpha=0.7, linewidth=1)
        ax.set_title(f'{config[:20]}\nLoss', fontsize=8, fontweight='bold')
        ax.set_xlabel('Epoch', fontsize=7)
        ax.set_ylabel('Loss', fontsize=7)
        ax.legend(fontsize=6)
        ax.grid(True, alpha=0.3)
        ax.tick_params(labelsize=6)


    ax9 = plt.subplot(5, 4, 9)
    metrics = ['accuracy', 'precision', 'recall', 'f1_score']


    top_configs = sorted(all_results.items(),
                         key=lambda x: x[1]['metrics']['f1_score'],
```

```
                      reverse=True)[:10]
top_config_names = [c[0] for c in top_configs]

x = np.arange(len(top_config_names))
width = 0.2

for i, metric in enumerate(metrics):
    values = [all_results[cfg]['metrics'][metric] for cfg in top_config_names]
    ax9.bar(x + i*width, values, width, label=metric.replace('_', ' ').title())

ax9.set_xlabel('Configuration', fontsize=8)
ax9.set_ylabel('Score', fontsize=8)
ax9.set_title('Top 10 Configs - Metrics', fontsize=10, fontweight='bold')
ax9.set_xticks(x + width * 1.5)
ax9.set_xticklabels([c[:8] for c in top_config_names], rotation=45, ha='right', font
ax9.legend(fontsize=7)
ax9.grid(True, alpha=0.3, axis='y')
ax9.set_ylim([0, 1.1])
ax9.tick_params(labelsize=6)

for idx, (config_name, config_data) in enumerate(top_configs[:10]):
    ax = plt.subplot(5, 4, 10 + idx)
    cm = config_data['metrics']['confusion_matrix']

    im = ax.imshow(cm, cmap='Blues')
    ax.set_title(f'{config_name[:15]}', fontsize=7, fontweight='bold')
    ax.set_xlabel('Pred', fontsize=6)
    ax.set_ylabel('True', fontsize=6)
    ax.set_xticks([0, 1])
    ax.set_yticks([0, 1])
    ax.set_xticklabels(['Cat', 'Dog'], fontsize=6)
    ax.set_yticklabels(['Cat', 'Dog'], fontsize=6)

    for i in range(2):
        for j in range(2):
            ax.text(j, i, str(cm[i, j]), ha='center', va='center',
                    color='white' if cm[i, j] > cm.max()/2 else 'black',
                    fontsize=10, fontweight='bold')

plt.suptitle('Dogs vs Cats', fontsize=16, fontweight='bold')
plt.tight_layout()
plt.savefig('all_27_experiments_results.png', dpi=300, bbox_inches='tight')
print("\nall_27_experiments_results.png")
```

```
def main():
    print("DOGS VS CATS")

    # Load data
    X_train, y_train, X_test, test_ids = extract_and_load_images(
        max_samples=5000,
        image_size=(64, 64)
    )

    print("\nprocessing")
    X_train = X_train.astype('float32') / 255.0
    X_test = X_test.astype('float32') / 255.0

    n_train = X_train.shape[0]
    n_test = X_test.shape[0]
    X_train = X_train.reshape(n_train, -1)
    X_test = X_test.reshape(n_test, -1)

    print(f"Flattened - Train: {X_train.shape}, Test: {X_test.shape}")

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)
```

```
 _                            _

# Train-val split
X_train_split, X_val, y_train_split, y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42, stratify=y_train
)

print(f"Split - Train: {X_train_split.shape[0]}, Val: {X_val.shape[0]}, Test: {X_tes


X_train_split = X_train_split.T
y_train_split = y_train_split.reshape(1, -1)
X_val = X_val.T
y_val = y_val.reshape(1, -1)
X_test_T = X_test.T

input_size = X_train_split.shape[0]
layer_sizes = [input_size, 256, 128, 64, 1]

print(f"\nArchitecture: {layer_sizes}")

activations = ['relu', 'tanh', 'leaky_relu']
initializations = ['xavier', 'kaiming', 'random']
optimizers_configs = [
    ('SGD', SGDOptimizer(learning_rate=0.01, momentum=0.9)),
    ('Adam', AdamOptimizer(learning_rate=0.001)),
    ('RMSprop', RMSpropOptimizer(learning_rate=0.001))
]


print(f"\nRunning Experiment")
print(f"\n{'-'*60}")


all_results = {}
experiment_num = 0
total_experiments = len(activations) * len(initializations) * len(optimizers_configs

# Test ALL combinations
for activation in activations:
    for initialization in initializations:
        for opt_name, optimizer in optimizers_configs:
            experiment_num += 1
            config_name = f"{activation}_{initialization}_{opt_name}"


            print(f"EXPERIMENT {experiment_num}/{total_experiments}: {config_name}")
            print(f"{'-'*80}")

            model = NeuralNetwork(
                layer_sizes,
                dropout_rate=0.3,
                use_batch_norm=True,
                activation=activation,
                initialization=initialization,
                l2_lambda=0.01
            )

            history = train_model_with_early_stopping(
                model, optimizer,
                X_train_split, y_train_split,
                X_val, y_val,
                epochs=100,
                batch_size=64,
                patience=15
            )

            # Evaluate
```

```
            metrics = evaluate_model(model, X_val, y_val)

            print(f"\n✓ Results: Acc={metrics['accuracy']:.4f}, "
                  f"Prec={metrics['precision']:.4f}, "
                  f"Rec={metrics['recall']:.4f}, "
                  f"F1={metrics['f1_score']:.4f}")

            all_results[config_name] = {
                'history': history,
                'metrics': metrics,
                'model': model
            }
print(f"{'Configuration':<30} {'Acc':<8} {'Prec':<8} {'Rec':<8} {'F1':<8}")
print("-"*80)

sorted_results = sorted(all_results.items(),
                        key=lambda x: x[1]['metrics']['f1_score'],
                        reverse=True)

for config_name, config_data in sorted_results:
    m = config_data['metrics']
    print(f"{config_name:<30} {m['accuracy']:<8.4f} {m['precision']:<8.4f} "
          f"{m['recall']:<8.4f} {m['f1_score']:<8.4f}")

best_config_name, best_config_data = sorted_results[0]
best_model = best_config_data['model']
best_metrics = best_config_data['metrics']

print(f"Best Configuration: {best_config_name}")

print(f"Activation:     {best_config_name.split('_')[0]}")
print(f"Initialization: {best_config_name.split('_')[1]}")
print(f"Optimizer:      {best_config_name.split('_')[2]}")
print(f"\nPerformance:")
print(f"  Accuracy:  {best_metrics['accuracy']:.4f} ({best_metrics['accuracy']*100:.
print(f"  Precision: {best_metrics['precision']:.4f}")
print(f"  Recall:    {best_metrics['recall']:.4f}")
print(f"  F1-Score:  {best_metrics['f1_score']:.4f}")


print("\nVisualization")
plot_comprehensive_results(all_results)

results_data = []
for config_name, config_data in sorted_results:
    m = config_data['metrics']
    act, init, opt = config_name.split('_')
    results_data.append({
        'Rank': len(results_data) + 1,
        'Configuration': config_name,
        'Activation': act,
        'Initialization': init,
        'Optimizer': opt,
        'Accuracy': f"{m['accuracy']:.4f}",
        'Precision': f"{m['precision']:.4f}",
        'Recall': f"{m['recall']:.4f}",
        'F1-Score': f"{m['f1_score']:.4f}"
    })

results_df = pd.DataFrame(results_data)
results_df.to_csv('all_27_experiments_results.csv', index=False)
print("all_27_experiments_results.csv")

print("\nMaking Prediction")
test_preds = predict_test_set(best_model, X_test_T)

submission = pd.DataFrame({
    'id': test_ids
```

```
        'id': test_ids,
        'label': test_preds
    })
    submission.to_csv('best_submission.csv', index=False)

    print(f"  best_submission.csv")
    print(f"  Total predictions: {len(test_preds)}")
    print(f"  Predicted Cats (0): {np.sum(test_preds==0)}")
    print(f"  Predicted Dogs (1): {np.sum(test_preds==1)}")

    for i, (config_name, config_data) in enumerate(sorted_results[:3], 1):
        print(f"  {i}. {config_name}: F1={config_data['metrics']['f1_score']:.4f}")
```

```
Start coding or generate with AI.
```

```
if __name__ == "__main__":
    main()
```

```
DOGS VS CATS

Extracting train.zip...
   Already extracted
  Found 25000 training images
   Limiting to 5000 samples...

Loading 5000 training images...
  Loaded 1000/5000 images...
  Loaded 2000/5000 images...
  Loaded 3000/5000 images...
  Loaded 4000/5000 images...
  Loaded 5000/5000 images...
\Training set: (5000, 64, 64, 3)
    Cats: 2521, Dogs: 2479

Extracting test.zip...
Already extracted
 Found 12500 test images
 Limiting to 1000 test samples...

Loading 1000 test images...
    Loaded 200/1000 images...
    Loaded 400/1000 images...
    Loaded 600/1000 images...
    Loaded 800/1000 images...
    Loaded 1000/1000 images...

    Test set: (1000, 64, 64, 3)

processing
Flattened - Train: (5000, 12288), Test: (1000, 12288)
Split - Train: 4000, Val: 1000, Test: 1000

Architecture: [12288, 256, 128, 64, 1]

Running Experiment

------------------------------------------------------------
EXPERIMENT 1/27: relu_xavier_SGD
------------------------------------------------------------------------------

Training: SGD, Act: relu, Init: xavier
Epochs: 100, Batch: 64, Patience: 15
--------------------------------------------------------------------------
Ep   1 | TrL: 0.7315 TrA: 0.6412 | VaL: 0.6677 VaA: 0.5840 | Pat: 0/15
Ep  10 | TrL: 0.5570 TrA: 0.8167 | VaL: 0.6932 VaA: 0.6080 | Pat: 5/15
```