A3 a1909709

August 11, 2024

1 Using Machine Learning Tools 2024, Assignment 3

1.1 Sign Language Image Classification using Deep Learning

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D,␣
 ↪Dropout
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.metrics import confusion_matrix
import matplotlib.colors as mcolors

import random
# Set random seeds
np.random.seed(42)
tf.random.set_seed(42)
random.seed(42)
```

2 Loading the data and displaying a sample of each letter

2.1 Load the data

[2]:
```
# Load the data
train_df = pd.read_csv('Assignment3_UMLT_2024/sign_mnist_train.csv')
test_df = pd.read_csv('Assignment3_UMLT_2024/sign_mnist_test.csv')

# Inspect the shape of the data
print('Shape of training data: ', train_df.shape)
print("Unique Label Values:")
print(np.unique(train_df['label'].unique()), '\n')

print('Shape of testing data: ',test_df.shape)
print("Unique Label Values:")
print(np.unique(test_df['label'].unique()))
```

```
Shape of training data:  (27455, 1025)
Unique Label Values:
[ 0  1  2  3  4  5  6  7  8 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]

Shape of testing data:  (7172, 1025)
Unique Label Values:
[ 0  1  2  3  4  5  6  7  8 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

The train data contains 27,455 observations and tes data contains 7,172 observations. Both has 1,025 variables.

```
[3]: # Get the unique labels from the training DataFrame
unique_y = np.unique(train_df['label'].unique())
n = len(unique_y) # Get the total number of unique labels

# Define the label names corresponding to each unique label (excluding 'J' and␣
↪'Z')
label_names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N',␣
↪'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y']

# Create a dictionary mapping each unique label to its corresponding letter
index_to_letter = {idx: letter for idx, letter in zip(unique_y, label_names)}

# Display the total number of unique labels
print("The total number of unique labels is", n)
index_to_letter # Output the mapping of indices to letters
```

```
The total number of unique labels is 24
```

```
[3]: {0: 'A',
 1: 'B',
 2: 'C',
 3: 'D',
 4: 'E',
 5: 'F',
 6: 'G',
 7: 'H',
 8: 'I',
 10: 'K',
 11: 'L',
 12: 'M',
 13: 'N',
 14: 'O',
 15: 'P',
 16: 'Q',
 17: 'R',
 18: 'S',
 19: 'T',
 20: 'U',
 21: 'V',
 22: 'W',
 23: 'X',
 24: 'Y'}
```

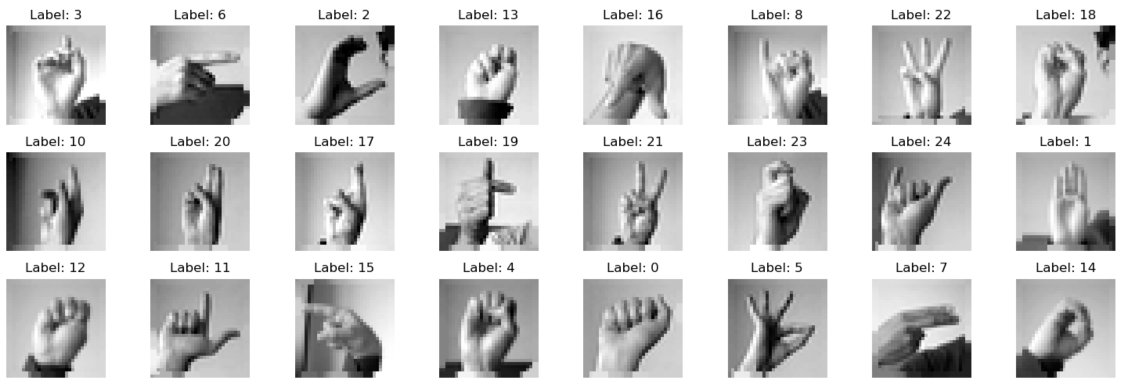
2.2 Display samples

[4]:
```
# Check the number of pixels per image
num_pixels = train_df.shape[1] - 1  # Subtracting 1 for the label column
img_size = int(np.sqrt(num_pixels))
print(f'Each image has {num_pixels} pixels and should be reshaped to␣
 ↪{img_size}x{img_size}.')
```

```
Each image has 1024 pixels and should be reshaped to 32x32.
```

[5]:
```
# Function to display a sample of each letter
def display_samples(data, num_samples=1):
    labels = data['label'].unique()
    fig, axes = plt.subplots(3, 8, figsize=(15, 5))
    axes = axes.flatten()
    for i, label in enumerate(labels):
        sample = data[data['label'] == label].sample(num_samples)
        img = sample.drop('label', axis=1).values.reshape(img_size, img_size)
        axes[i].imshow(img, cmap='gray')
        axes[i].set_title(f'Label: {label}')
        axes[i].axis('off')
    plt.tight_layout()
    plt.show()
```

[6]:
```
# Display samples
display_samples(train_df)
```



[7]:
```
# Function to preprocess the data
def preprocess_data(df):
    # Separate features (X) and labels (y)
    X = df.drop('label', axis=1).values
    y = df['label'].values

    # Normalize pixel values to be between 0 and 1
    X = X / 255.0

    # Reshape the features to match the input shape for CNN (num_samples,␣
 ↪img_size, img_size, 1)
    X = X.reshape(-1, img_size, img_size, 1)

    # One-hot encode the labels
    y = to_categorical(y, num_classes=26)

    return X, y
```

[8]:
```
X_train, y_train = preprocess_data(train_df) # Preprocess the training data
X_test_FULL, y_test_FULL = preprocess_data(test_df) # Preprocess the full test␣
 ↪data

# Split the full test data into validation and test sets
# Here, 40% of the data is used for testing and the remaining 60% for validation
X_val, X_test, y_val, y_test = train_test_split(X_test_FULL, y_test_FULL,␣
 ↪test_size=0.4, random_state=42)
```

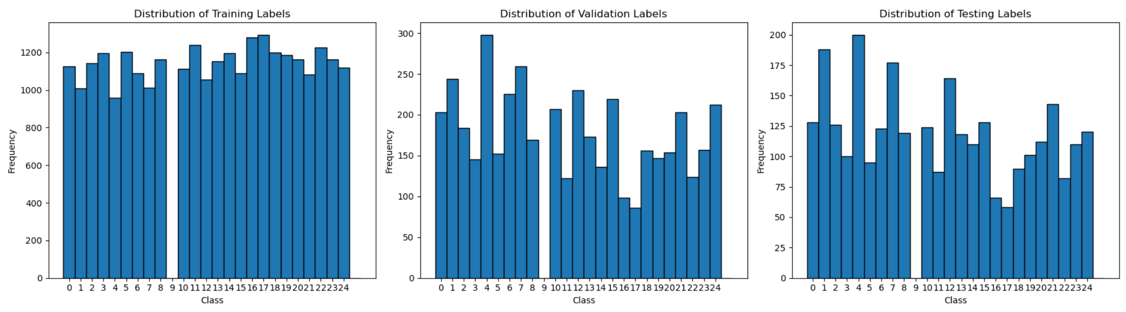
[9]:
```
# Inspect the shape and data types of the processed datasets
print('Shape of X_train: ', X_train.shape, 'dtype: ', X_train.dtype)
print('Shape of y_train: ', y_train.shape, '\n')

print('Shape of X_val: ', X_val.shape, 'dtype: ',X_val.dtype)
print('Shape of y_val: ', y_val.shape, '\n')

print('Shape of X_test: ',X_test.shape, 'dtype: ',X_test.dtype)
print('Shape of y_test: ',y_test.shape)
```

```
Shape of X_train:  (27455, 32, 32, 1) dtype:  float64
Shape of y_train:  (27455, 26)

Shape of X_val:  (4303, 32, 32, 1) dtype:  float64
Shape of y_val:  (4303, 26)

Shape of X_test:  (2869, 32, 32, 1) dtype:  float64
Shape of y_test:  (2869, 26)
```

```
[10]: # Convert one-hot encoded labels back to original labels for histogram plotting
y_train_labels = np.argmax(y_train, axis=1)
y_val_labels = np.argmax(y_val, axis=1)
y_test_labels = np.argmax(y_test, axis=1)
```

```
[11]: # Function to plot histogram of label distribution
def plot_histogram(y, title, ax):
    ax.hist(y, bins=np.arange(27) - 0.5, edgecolor='black')
    ax.set_title(f'Distribution of {title} Labels')
    ax.set_xlabel('Class')
    ax.set_ylabel('Frequency')
    ax.set_xticks(np.arange(25))
```

```
[12]: # Create subplots
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Plot histograms on the subplots
plot_histogram(y_train_labels, 'Training', axes[0])
plot_histogram(y_val_labels, 'Validation', axes[1])
plot_histogram(y_test_labels, 'Testing', axes[2])

# Display the plots
plt.tight_layout()
plt.show()
```



Class 9 and 25 is missing as there are no cases for 9=J or 25=Z because of gesture motions as explained in the description provided in the assignment. The training data is relatively balanced in distribution, while the validation and testing datasets show some imbalance. Specifically, classes “Q” and “R” appear less frequently, whereas class “E” is the most common. Since our focus is on the training data for model development, no adjustments are necessary for the training dataset.

3 Model Optimization

Here I trained and optimised both densely connected (DNN) and CNN style models. I did so on 50 epochs and implemented early stopping to help efficiently find the best model without overfitting or underfitting.

[13]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout, Conv2D,␣
 ↪MaxPooling2D, Input
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras import regularizers
import keras
```

[14]:
```
# Define a range of optimizers and their learning rates to test
learning_rates = [0.01, 0.001, 0.0001]#, 0.00001]
optimizer_setup = {
    'SGD': keras.optimizers.SGD,
    'Adam': keras.optimizers.Adam,
    'RMSprop': keras.optimizers.RMSprop,
    'Nadam': keras.optimizers.Nadam
}

# Store the results for each optimizer, learning rate, and model type
results_dict = {
    'dnn': {},
    'cnn': {}
}
```

[15]:
```
# Define the DNN model architecture
def create_dnn_model(hiddensizes, actfn, optimizer, learningrate, l2_lambda=0.
 ↪0, **kwargs):
    model = Sequential()
    model.add(Input(shape=(img_size, img_size, 1)))  # Specify input shape
    model.add(Flatten())

    # Add hidden layers with L2 regularization
    for n in hiddensizes:
        model.add(Dense(n, activation=actfn,
            kernel_initializer="he_normal",
            kernel_regularizer=regularizers.l2(l2_lambda)
        ))

    # Output layer
    model.add(Dense(26, activation="softmax"))

    # Configure the optimizer with additional parameters
    opt_params = {k: v for k, v in kwargs.items()}
    optimizer_instance = optimizer(learning_rate=learningrate, **opt_params)

    # Compile the model
    model.compile(
        loss="categorical_crossentropy",
        optimizer=optimizer_instance,
        metrics=["accuracy"]
    )

    return model

# Define the CNN model architecture
def create_cnn_model(filtersizes, kernel_sizes, pool_sizes, actfn, optimizer,␣
 ↪learningrate, l2_lambda=0.0, **kwargs):
    model = Sequential()
    model.add(Input(shape=(img_size, img_size, 1)))  # Specify input shape
    model.add(Conv2D(filters=filtersizes[0], kernel_size=kernel_sizes[0],␣
 ↪activation=actfn,
                     padding="same",
                     kernel_initializer="he_normal",␣
 ↪kernel_regularizer=regularizers.l2(l2_lambda)))
    model.add(MaxPooling2D(pool_size=pool_sizes[0]))

    for i in range(1, len(filtersizes)):
        model.add(Conv2D(filters=filtersizes[i], kernel_size=kernel_sizes[i],␣
 ↪activation=actfn,
                         padding="same", kernel_initializer="he_normal",
                         kernel_regularizer=regularizers.l2(l2_lambda)))
        model.add(MaxPooling2D(pool_size=pool_sizes[i]))

    model.add(Flatten())
    model.add(Dense(128, activation=actfn, kernel_regularizer=regularizers.
 ↪l2(l2_lambda)))
    model.add(Dropout(0.5))
    model.add(Dense(26, activation="softmax"))

    # Configure the optimizer with additional parameters
    opt_params = {k: v for k, v in kwargs.items()}
    optimizer_instance = optimizer(learning_rate=learningrate, **opt_params)

    # Compile the model
    model.compile(
        loss="categorical_crossentropy",
        optimizer=optimizer_instance,
        metrics=["accuracy"]
    )

    return model

# Function to evaluate a model with different optimizers and learning rates
def evaluate_model(model_type, **kwargs):
    for optimizer_name, optimizer_class in optimizer_setup.items():
        for lr in learning_rates:
            print(f"Testing {model_type.upper()} model with optimizer:␣
↪{optimizer_name}, learning rate: {lr}")

            # Create and compile a new instance of the model for each␣
↪configuration
            if model_type == 'dnn':
                model = create_dnn_model(
                    hiddensizes=[128, 64],
                    actfn='relu',
                    optimizer=optimizer_class,
                    learningrate=lr,
                    **kwargs
                )
            else:
                model = create_cnn_model(
                    filtersizes=[32, 64],
                    kernel_sizes=[(3, 3), (3, 3)],
                    pool_sizes=[(2, 2), (2, 2)],
                    actfn='relu',
                    optimizer=optimizer_class,
                    learningrate=lr,
                    **kwargs
                )

            optimizer = optimizer_class(learning_rate=lr)
            model.compile(optimizer=optimizer, loss='categorical_crossentropy',␣
↪metrics=['accuracy'])

            # Define early stopping callback
            early_stopping = EarlyStopping(monitor='val_loss', patience=3,␣
↪restore_best_weights=True)

            # Train the model
            history = model.fit(X_train, y_train, epochs=50,␣
↪validation_data=(X_val, y_val), callbacks=[early_stopping], verbose=0)

            # Evaluate the model on the validation set
            val_accuracy = history.history['val_accuracy'][-1]
            num_epochs = len(history.epoch)

            if optimizer_name not in results_dict[model_type]:
                results_dict[model_type][optimizer_name] = []
            results_dict[model_type][optimizer_name].append((lr, val_accuracy,␣
↪num_epochs))
```

```
            print(f"Validation accuracy: {val_accuracy:.4f}, Number of epochs:␣
 ↪{num_epochs}")
```

3.1 Comparison of DNN and CNN Models: Optimizer and Learning Rate Evaluation

In this section, I compare the performance of DNN and CNN models, with a focus on evaluating the effectiveness of different optimizers and learning rates. The analysis considers both the validation accuracy and the number of epochs required for training, providing insights into the efficiency and effectiveness of each model configuration.

[16]:
```
# Evaluate DNN model
evaluate_model('dnn')
```

```
Testing DNN model with optimizer: SGD, learning rate: 0.01
Validation accuracy: 0.7202, Number of epochs: 20
Testing DNN model with optimizer: SGD, learning rate: 0.001
Validation accuracy: 0.6751, Number of epochs: 50
Testing DNN model with optimizer: SGD, learning rate: 0.0001
Validation accuracy: 0.2673, Number of epochs: 50
Testing DNN model with optimizer: Adam, learning rate: 0.01
Validation accuracy: 0.0228, Number of epochs: 21
Testing DNN model with optimizer: Adam, learning rate: 0.001
Validation accuracy: 0.7218, Number of epochs: 8
Testing DNN model with optimizer: Adam, learning rate: 0.0001
Validation accuracy: 0.7104, Number of epochs: 23
Testing DNN model with optimizer: RMSprop, learning rate: 0.01
Validation accuracy: 0.0228, Number of epochs: 4
Testing DNN model with optimizer: RMSprop, learning rate: 0.001
Validation accuracy: 0.6365, Number of epochs: 8
Testing DNN model with optimizer: RMSprop, learning rate: 0.0001
Validation accuracy: 0.7076, Number of epochs: 23
Testing DNN model with optimizer: Nadam, learning rate: 0.01
Validation accuracy: 0.5347, Number of epochs: 5
Testing DNN model with optimizer: Nadam, learning rate: 0.001
Validation accuracy: 0.7016, Number of epochs: 7
Testing DNN model with optimizer: Nadam, learning rate: 0.0001
Validation accuracy: 0.7007, Number of epochs: 20
```

[17]:
```
# Evaluate CNN model
evaluate_model('cnn')
```

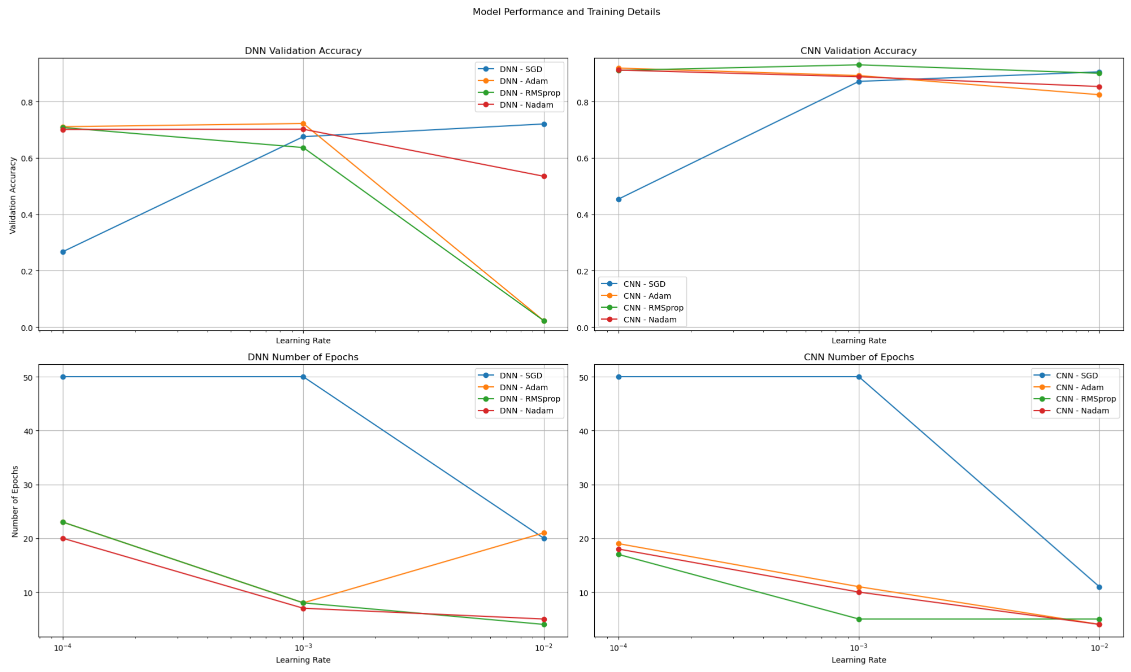
```
Testing CNN model with optimizer: SGD, learning rate: 0.01
Validation accuracy: 0.9050, Number of epochs: 11
Testing CNN model with optimizer: SGD, learning rate: 0.001
Validation accuracy: 0.8710, Number of epochs: 50
Testing CNN model with optimizer: SGD, learning rate: 0.0001
Validation accuracy: 0.4536, Number of epochs: 50
Testing CNN model with optimizer: Adam, learning rate: 0.01
```

```
Validation accuracy: 0.8238, Number of epochs: 4
Testing CNN model with optimizer: Adam, learning rate: 0.001
Validation accuracy: 0.8917, Number of epochs: 11
Testing CNN model with optimizer: Adam, learning rate: 0.0001
Validation accuracy: 0.9182, Number of epochs: 19
Testing CNN model with optimizer: RMSprop, learning rate: 0.01
Validation accuracy: 0.8998, Number of epochs: 5
Testing CNN model with optimizer: RMSprop, learning rate: 0.001
Validation accuracy: 0.9298, Number of epochs: 5
Testing CNN model with optimizer: RMSprop, learning rate: 0.0001
Validation accuracy: 0.9101, Number of epochs: 17
Testing CNN model with optimizer: Nadam, learning rate: 0.01
Validation accuracy: 0.8529, Number of epochs: 4
Testing CNN model with optimizer: Nadam, learning rate: 0.001
Validation accuracy: 0.8875, Number of epochs: 10
Testing CNN model with optimizer: Nadam, learning rate: 0.0001
Validation accuracy: 0.9112, Number of epochs: 18
```

Here I plot the models results to identify the best model obtained during this evaluation.

[18]:
```python
# Create subplots for DNN and CNN results
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(20, 12),␣
 ↪sharex=True)

# Plot DNN validation accuracy
for optimizer_name, values in results_dict['dnn'].items():
    lrs, accuracies, _ = zip(*values)
    ax1.plot(lrs, accuracies, 'o-', label=f"DNN - {optimizer_name}")

ax1.set_xscale('log')
ax1.set_xlabel('Learning Rate')
ax1.set_ylabel('Validation Accuracy')
ax1.set_title('DNN Validation Accuracy')
ax1.legend()
ax1.grid(True)

# Plot CNN validation accuracy
for optimizer_name, values in results_dict['cnn'].items():
    lrs, accuracies, _ = zip(*values)
    ax2.plot(lrs, accuracies, 'o-', label=f"CNN - {optimizer_name}")

ax2.set_xscale('log')
ax2.set_xlabel('Learning Rate')
ax2.set_title('CNN Validation Accuracy')
ax2.legend()
ax2.grid(True)

# Ensure the same y-axis for validation accuracy plots
```

```python
min_val_accuracy = min(ax1.get_ylim()[0], ax2.get_ylim()[0])
max_val_accuracy = max(ax1.get_ylim()[1], ax2.get_ylim()[1])
ax1.set_ylim(min_val_accuracy, max_val_accuracy)
ax2.set_ylim(min_val_accuracy, max_val_accuracy)

# Plot DNN epochs
for optimizer_name, values in results_dict['dnn'].items():
    lrs, _, epochs = zip(*values)
    ax3.plot(lrs, epochs, 'o-', label=f"DNN - {optimizer_name}")

ax3.set_xscale('log')
ax3.set_xlabel('Learning Rate')
ax3.set_ylabel('Number of Epochs')
ax3.set_title('DNN Number of Epochs')
ax3.legend()
ax3.grid(True)

# Plot CNN epochs
for optimizer_name, values in results_dict['cnn'].items():
    lrs, _, epochs = zip(*values)
    ax4.plot(lrs, epochs, 'o-', label=f"CNN - {optimizer_name}")

ax4.set_xscale('log')
ax4.set_xlabel('Learning Rate')
ax4.set_title('CNN Number of Epochs')
ax4.legend()
ax4.grid(True)

# Ensure the same y-axis for number of epochs plots
min_epochs = min(ax3.get_ylim()[0], ax4.get_ylim()[0])
max_epochs = max(ax3.get_ylim()[1], ax4.get_ylim()[1])
ax3.set_ylim(min_epochs, max_epochs)
ax4.set_ylim(min_epochs, max_epochs)

plt.suptitle('Model Performance and Training Details')
plt.tight_layout(rect=[0, 0, 1, 0.96])  # Adjust layout to fit suptitle
plt.show()
```



CNN vs. DNN: CNN models consistently outperformed DNNs, highlighting their superior ability to capture spatial hierarchies in the data.

Optimizer Performance: RMSprop with a learning rate of 0.001 achieved the highest validation accuracy of 0.9298 in 5 epochs. Adam with a learning rate of 0.0001 and RMSprop with 0.0001 also performed well, with accuracies of 0.9182 and 0.9101, respectively.

Learning Rate Impact: A learning rate of 0.001 generally yielded better performance for CNNs, particularly with RMSprop, while lower rates required more epochs for similar results.

Epochs Consideration: RMSprop with a learning rate of 0.001 is the most efficient, reaching high accuracy in fewer epochs.

The CNN with RMSprop and a learning rate of 0.001 is the optimal model, balancing high accuracy and efficiency.

[20]:
```python
# Initialize a dictionary to store the best model information
best_model_info = {
    'model_type': None,
    'optimizer': None,
    'learning_rate': None,
    'accuracy': -float('inf')
}

# Iterate through the results_dict to find the model with the highest␣
 ↪validation accuracy
for model_type, optimizers in results_dict.items():
    for optimizer_name, values in optimizers.items():
        # `values` is a list of tuples: (learning rate, validation accuracy,␣
↪number of epochs, _, _)
        for lr, val_accuracy, _ in values:  # Unpack only learning rate and␣
↪validation accuracy
            # Update the best_model_info if the current model has a higher␣
↪validation accuracy
            if val_accuracy > best_model_info['accuracy']:
                best_model_info.update({
                    'model_type': model_type,
                    'optimizer': optimizer_name,
                    'learning_rate': lr,
                    'accuracy': val_accuracy
                })

# Print the information of the best model
print("Best Model Info:", best_model_info)
```

```
Best Model Info: {'model_type': 'cnn', 'optimizer': 'RMSprop', 'learning_rate':
0.001, 'accuracy': 0.9298164248466492}
```

[21]:
```
best_model_info
```

[21]:
```
{'model_type': 'cnn',
 'optimizer': 'RMSprop',
 'learning_rate': 0.001,
 'accuracy': 0.9298164248466492}
```

Based on the results, I found the optimal validation accuracy was achieved with the CNN model using the RMSprop optimizer and a learning rate of 0.001. This combination provided the best performance in terms of validation accuracy, making it the optimal choice.

3.2 Regularization Values

In this section, I compare different L2 regularization values to evaluate their impact on mitigating potential overfitting.

[22]:
```
l2_lambda_values = [0.0, 0.01, 0.1]

# Function to evaluate a model with different L2 regularization values
def evaluate_model(model_type, learning_rate, optimizer_name, **kwargs):
    # Check if the model_type, learning_rate, and optimizer_name match the best␣
↪model
    if model_type != best_model_info['model_type'] or \
       learning_rate != best_model_info['learning_rate'] or \
       optimizer_name != best_model_info['optimizer']:
        print(f"Warning: The selected model configuration does not match the␣
↪best model configuration.")
```

```
    optimizer_class = optimizer_setup[optimizer_name]

    for l2_lambda in l2_lambda_values:
        print(f"Testing {model_type.upper()} model with optimizer:␣
↪{optimizer_name}, learning rate: {learning_rate}, L2 regularization:␣
↪{l2_lambda}")

        # Create and compile a new instance of the model for each configuration
        if model_type == 'dnn':
            model = create_dnn_model(
                hiddensizes=[128, 64],
                actfn='relu',
                optimizer=optimizer_class,
                learningrate=learning_rate,
                l2_lambda=l2_lambda,
                **kwargs
            )
        else:
            model = create_cnn_model(
                filtersizes=[32, 64],
                kernel_sizes=[(3, 3), (3, 3)],
                pool_sizes=[(2, 2), (2, 2)],
                actfn='relu',
                optimizer=optimizer_class,
                learningrate=learning_rate,
                l2_lambda=l2_lambda,
                **kwargs
            )

        optimizer_instance = optimizer_class(learning_rate=learning_rate)
        model.compile(optimizer=optimizer_instance,␣
↪loss='categorical_crossentropy', metrics=['accuracy'])

        # Define early stopping callback
        early_stopping = EarlyStopping(monitor='val_loss', patience=3,␣
↪restore_best_weights=True)

        # Train the model
        history = model.fit(X_train, y_train, epochs=10,␣
↪validation_data=(X_val, y_val), callbacks=[early_stopping], verbose=0)

        # Evaluate the model on the validation set
        val_accuracy = history.history['val_accuracy'][-1]
        num_epochs = len(history.epoch)

        if optimizer_name not in results_dict[model_type]:
            results_dict[model_type][optimizer_name] = []
        results_dict[model_type][optimizer_name].append((l2_lambda,␣
↪val_accuracy, num_epochs))

        print(f"Validation accuracy: {val_accuracy:.4f}, Number of epochs:␣
↪{num_epochs}")
```

[23]:
```
# Use the best model configuration to evaluate
evaluate_model(
    model_type=best_model_info['model_type'],
    learning_rate=best_model_info['learning_rate'],
    optimizer_name=best_model_info['optimizer']
)
```

```
Testing CNN model with optimizer: RMSprop, learning rate: 0.001, L2
regularization: 0.0
Validation accuracy: 0.9226, Number of epochs: 5
Testing CNN model with optimizer: RMSprop, learning rate: 0.001, L2
regularization: 0.01
Validation accuracy: 0.8699, Number of epochs: 10
Testing CNN model with optimizer: RMSprop, learning rate: 0.001, L2
regularization: 0.1
Validation accuracy: 0.4230, Number of epochs: 10
```

The results from testing the CNN model with different L2 regularization values highlight the impact of regularization on model performance:

L2 Regularization: 0.0 With no L2 regularization, the CNN model achieved the highest validation accuracy of 92.26%. This suggests that the model performed well on the validation set, indicating that the lack of regularization did not lead to overfitting in this scenario.

L2 Regularization: 0.01 Introducing a small amount of L2 regularization (0.01) reduced the validation accuracy to 86.99%. This level of regularization appears effective in mitigating overfitting without drastically impacting performance, though it results in slightly reduced accuracy.

L2 Regularization: 0.1 With a higher L2 regularization value (0.1), the validation accuracy dropped significantly to 42.30%. This indicates that the regularization is too strong, leading to underfitting where the model is overly constrained and unable to learn effectively.

Based on these results, **no regularization** achieved the best performance with a validation accuracy of 92.26%. While **L2 regularization at 0.01** slightly reduced performance to 86.99%, it still maintained relatively high accuracy and helped in mitigating overfitting. **L2 regularization at 0.1** caused a substantial drop in accuracy to 42.30%, indicating over-regularization and underfitting. Therefore, choosing **no regularization** is recommended as it balances high accuracy with minimal risk of overfitting.

[24]:
```
# Manually add the l2_lambda value
best_model_info['l2_lambda'] = 0.0
```

3.3 Filter sizes, kernel sizes, and pool sizes

[25]:
```
# Define a range of filter sizes, kernel sizes, and pool sizes to test
filter_size_configs = [
    ([32, 64], [(3, 3), (3, 3)], [(2, 2), (2, 2)]),
    ([64, 128], [(3, 3), (3, 3)], [(2, 2), (2, 2)]),
    ([32, 64, 128], [(3, 3), (3, 3), (3, 3)], [(2, 2), (2, 2), (2, 2)]),
    ([64, 128, 256], [(3, 3), (3, 3), (3, 3)], [(2, 2), (2, 2), (2, 2)])
]

# Define the parameters based on best_model_info
optimizer_name = best_model_info['optimizer']
learning_rate = best_model_info['learning_rate']
l2_lambda = best_model_info['l2_lambda']
optimizer_class = optimizer_setup[optimizer_name]

# Function to evaluate CNN models with different filter sizes
def evaluate_cnn_filter_sizes_fixed_params():
    for filtersizes, kernel_sizes, pool_sizes in filter_size_configs:
        print(f"Testing CNN model with filters: {filtersizes}, kernel sizes:␣
 ↪{kernel_sizes}, pool sizes: {pool_sizes}, optimizer: {optimizer_name},␣
 ↪learning rate: {learning_rate}, L2 regularization: {l2_lambda}")

        # Create and compile a new instance of the model for each configuration
        model = create_cnn_model(
            filtersizes=filtersizes,
            kernel_sizes=kernel_sizes,
            pool_sizes=pool_sizes,
            actfn='relu',
            optimizer=optimizer_class,
            learningrate=learning_rate,
            l2_lambda=l2_lambda
        )

        # Define early stopping callback
        early_stopping = EarlyStopping(monitor='val_loss', patience=3,␣
 ↪restore_best_weights=True)

        # Train the model
        history = model.fit(X_train, y_train, epochs=50,␣
 ↪validation_data=(X_val, y_val), callbacks=[early_stopping], verbose=0)

        # Evaluate the model on the validation set
        val_accuracy = history.history['val_accuracy'][-1]
        num_epochs = len(history.epoch)

        # Use a tuple instead of a list for the filtersizes key
        key = (tuple(filtersizes), optimizer_class.__name__, learning_rate)

        if key not in results_dict['cnn']:
            results_dict['cnn'][key] = []
        results_dict['cnn'][key].append((val_accuracy, num_epochs))

        print(f"Validation accuracy: {val_accuracy:.4f}, Number of epochs:␣
↪{num_epochs}")
```

[26]:
```
# Run the evaluation for CNN models with fixed parameters
evaluate_cnn_filter_sizes_fixed_params()
```

```
Testing CNN model with filters: [32, 64], kernel sizes: [(3, 3), (3, 3)], pool
sizes: [(2, 2), (2, 2)], optimizer: RMSprop, learning rate: 0.001, L2
regularization: 0.0
Validation accuracy: 0.9312, Number of epochs: 5
Testing CNN model with filters: [64, 128], kernel sizes: [(3, 3), (3, 3)], pool
sizes: [(2, 2), (2, 2)], optimizer: RMSprop, learning rate: 0.001, L2
regularization: 0.0
Validation accuracy: 0.9382, Number of epochs: 5
Testing CNN model with filters: [32, 64, 128], kernel sizes: [(3, 3), (3, 3),
(3, 3)], pool sizes: [(2, 2), (2, 2), (2, 2)], optimizer: RMSprop, learning
rate: 0.001, L2 regularization: 0.0
Validation accuracy: 0.9682, Number of epochs: 8
Testing CNN model with filters: [64, 128, 256], kernel sizes: [(3, 3), (3, 3),
(3, 3)], pool sizes: [(2, 2), (2, 2), (2, 2)], optimizer: RMSprop, learning
rate: 0.001, L2 regularization: 0.0
Validation accuracy: 0.9528, Number of epochs: 5
```

The best model chosen is the model with filters [32, 64, 128], kernel sizes [(3, 3), (3, 3), (3, 3)], and pool sizes [(2, 2), (2, 2), (2, 2)]. This model achieved a validation accuracy of 0.9682, which is the highest among all tested configurations.

The reason this model stands out is its ability to extract more complex features due to the deeper architecture with three convolutional layers. Although it required 8 epochs, slightly more than other models, the significant boost in accuracy justifies the additional training time. The use of RMSprop with a learning rate of 0.001 contributed to fast convergence, making this model not only accurate but also efficient in terms of learning speed.

Therefore, this model offers the best balance between complexity and performance, making it the optimal choice for the task.

[30]:
```
# Filter sizes, kernel sizes, and pool sizes to test
filter_size = [32, 64, 128]
kernel_size = [(3, 3), (3, 3), (3, 3)]
pool_size = [(2, 2), (2, 2), (2, 2)]

# Manually update the dictionary with best model information
best_model_info.update({
    'filter_size': filter_size,
    'kernel_size': kernel_size,
    'pool_size': pool_size
})
```

The best model is achieved using a CNN with the following configuration:

- **Optimizer:** RMSprop
- **Learning Rate:** 0.001
- **L2 Regularization:** 0.0
- **Filter Sizes:** [32, 64, 128]
- **Kernel Sizes:** [(3, 3), (3, 3), (3, 3)]
- **Pool Sizes:** [(2, 2), (2, 2), (2, 2)]

This model attained a validation accuracy of 96.70%, demonstrating optimal performance with these settings.

4 Final Results

The best model was then trained on the full training dataset and evaluated on both the validation and test datasets. To optimize the training process and avoid overfitting or underfitting, early stopping with a maximum of 50 epochs was employed. A patience of 6 epochs was chosen to prevent premature convergence, ensuring the model had sufficient time to fully learn the complexities of the data. This balance was crucial because it ensured that the model not only achieved high overall accuracy but also maintained strong accuracy for each individual letter. By allowing the model to train long enough, the chosen patience setting helped capture additional performance gains without the risk of stopping too early.

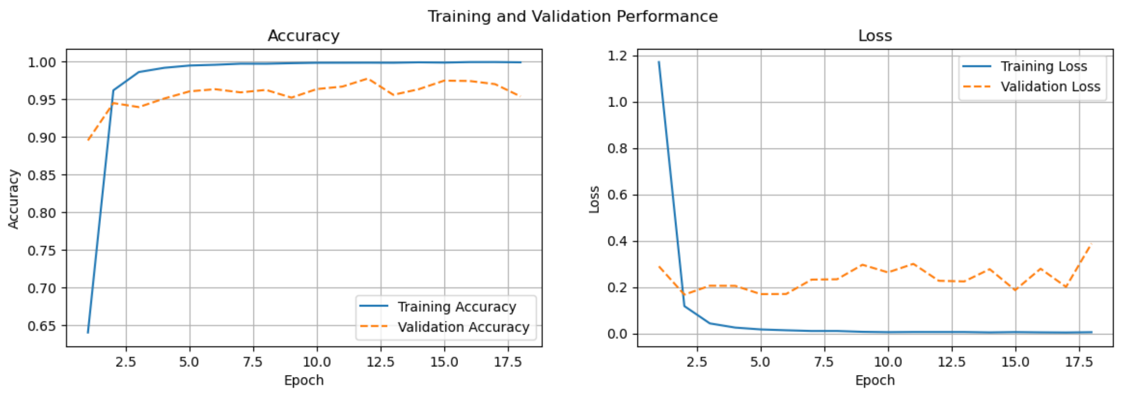
[38]:
```
# Define the parameters based on best_model_info
filtersizes = best_model_info['filter_size']
kernel_sizes = best_model_info['kernel_size']
pool_sizes = best_model_info['pool_size']
activation_function = 'relu'
optimizer_name = best_model_info['optimizer']
learning_rate = best_model_info['learning_rate']
l2_lambda = best_model_info['l2_lambda']

# Get the optimizer from your setup
optimizer = optimizer_setup[optimizer_name]

# Create the model using the defined parameters
model = create_cnn_model(
    filtersizes=filtersizes,
    kernel_sizes=kernel_sizes,
    pool_sizes=pool_sizes,
    actfn=activation_function,
    optimizer=optimizer,
    learningrate=learning_rate,
    l2_lambda=l2_lambda
)

# Compile the model
model.compile(
    optimizer=optimizer(learning_rate=learning_rate),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Define early stopping callback
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=6,
    restore_best_weights=True
)

# Train the model with the best regularizer
history = model.fit(
    X_train,  # Training data
    y_train,  # Training labels
    epochs=50,  # Number of epochs
    validation_data=(X_val, y_val),  # Validation data
    callbacks=[early_stopping],  # List of callbacks
    verbose=1  # Print progress
)
```

```
Epoch 1/50
858/858             36s 39ms/step -
accuracy: 0.3835 - loss: 2.1088 - val_accuracy: 0.8950 - val_loss: 0.2893
Epoch 2/50
858/858             36s 42ms/step -
accuracy: 0.9467 - loss: 0.1636 - val_accuracy: 0.9445 - val_loss: 0.1673
Epoch 3/50
858/858             33s 38ms/step -
accuracy: 0.9837 - loss: 0.0482 - val_accuracy: 0.9391 - val_loss: 0.2057
Epoch 4/50
858/858             34s 40ms/step -
accuracy: 0.9907 - loss: 0.0266 - val_accuracy: 0.9505 - val_loss: 0.2052
Epoch 5/50
858/858             33s 38ms/step -
accuracy: 0.9941 - loss: 0.0185 - val_accuracy: 0.9600 - val_loss: 0.1696
Epoch 6/50
858/858             34s 40ms/step -
accuracy: 0.9944 - loss: 0.0138 - val_accuracy: 0.9628 - val_loss: 0.1701
Epoch 7/50
858/858             34s 40ms/step -
accuracy: 0.9961 - loss: 0.0118 - val_accuracy: 0.9586 - val_loss: 0.2321
```

```
Epoch 8/50
858/858                 33s 39ms/step -
accuracy: 0.9968 - loss: 0.0095 - val_accuracy: 0.9619 - val_loss: 0.2336
Epoch 9/50
858/858                 38s 45ms/step -
accuracy: 0.9971 - loss: 0.0071 - val_accuracy: 0.9517 - val_loss: 0.2962
Epoch 10/50
858/858                 40s 47ms/step -
accuracy: 0.9973 - loss: 0.0063 - val_accuracy: 0.9630 - val_loss: 0.2632
Epoch 11/50
858/858                 37s 43ms/step -
accuracy: 0.9978 - loss: 0.0067 - val_accuracy: 0.9663 - val_loss: 0.3000
Epoch 12/50
858/858                 35s 41ms/step -
accuracy: 0.9984 - loss: 0.0056 - val_accuracy: 0.9770 - val_loss: 0.2270
Epoch 13/50
858/858                 36s 42ms/step -
accuracy: 0.9980 - loss: 0.0057 - val_accuracy: 0.9554 - val_loss: 0.2243
Epoch 14/50
858/858                 36s 42ms/step -
accuracy: 0.9980 - loss: 0.0053 - val_accuracy: 0.9628 - val_loss: 0.2771
Epoch 15/50
858/858                 36s 42ms/step -
accuracy: 0.9978 - loss: 0.0060 - val_accuracy: 0.9742 - val_loss: 0.1867
Epoch 16/50
858/858                 35s 41ms/step -
accuracy: 0.9986 - loss: 0.0063 - val_accuracy: 0.9737 - val_loss: 0.2790
Epoch 17/50
858/858                 36s 42ms/step -
accuracy: 0.9985 - loss: 0.0055 - val_accuracy: 0.9696 - val_loss: 0.2007
Epoch 18/50
858/858                 37s 43ms/step -
accuracy: 0.9985 - loss: 0.0050 - val_accuracy: 0.9533 - val_loss: 0.3874
```

[39]:
```
# Ensure the full range of epochs is considered
print("Number of epochs completed:", len(history.history['accuracy']))
epochs = range(1, len(history.history['accuracy']) + 1)

plt.figure(figsize=(14, 4))

# Plot accuracy
plt.subplot(1, 2, 1)
plt.plot(epochs, history.history['accuracy'], label='Training Accuracy')
plt.plot(epochs, history.history['val_accuracy'], label='Validation Accuracy',␣
 ↪linestyle='--')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(epochs, history.history['loss'], label='Training Loss')
plt.plot(epochs, history.history['val_loss'], label='Validation Loss',␣
 ↪linestyle='--')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.suptitle('Training and Validation Performance')
plt.show()
```

```
Number of epochs completed: 18
```



The CNN model's performance over 18 epochs demonstrates strong learning with some signs of generalization challenges. Training accuracy quickly increased from 38.35% in epoch 1 to 99.85% by epoch 18, with training loss decreasing steadily, reaching 0.0050, indicating effective learning on the training data.

Validation accuracy also improved significantly, starting at 89.50% in epoch 1 and peaking at 97.70% by epoch 12. However, fluctuations in validation accuracy and loss in the later epochs suggest potential instability in generalization, possibly due to overfitting. While the model shows high final accuracy on both training and validation sets, the variability in validation performance indicates that further adjustments, such as enhanced regularization or early stopping, may be necessary to improve generalization.

In summary, despite some fluctuations, the model effectively met performance criteria, validating its suitability for the intended application.

```
[40]: # Evaluate on the validation set
val_loss, val_accuracy = model.evaluate(X_val, y_val, verbose=1)
print(f'Validation Accuracy: {val_accuracy:.4f}')

# Evaluate on the test set
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=1)
print(f'Test Accuracy: {test_accuracy:.4f}')
```

```
135/135              2s 14ms/step -
accuracy: 0.9760 - loss: 0.2367
Validation Accuracy: 0.9770
90/90              1s 12ms/step -
accuracy: 0.9784 - loss: 0.2475
Test Accuracy: 0.9826
```

The model obtained a test accuracy of 98.26%.

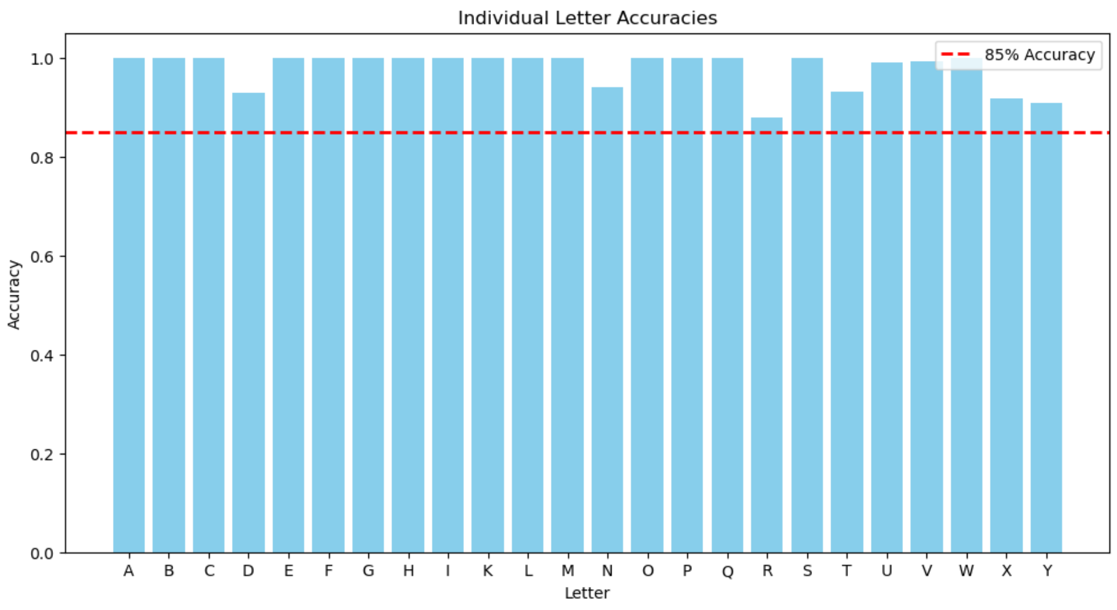
```
[41]: # Get predictions from the model
predictions = model.predict(X_test)
pred_labels = np.argmax(predictions, axis=1)
true_labels = np.argmax(y_test, axis=1)

# Calculate individual accuracies
unique_labels = np.unique(true_labels)
individual_accuracies = {}
for label in unique_labels:
    mask = true_labels == label
    individual_accuracies[label] = accuracy_score(true_labels[mask],␣
  ↪pred_labels[mask])

# Round individual accuracies to 4 decimal places
individual_accuracies_rounded = {label: round(acc, 4) for label, acc in␣
  ↪individual_accuracies.items()}
```

```
90/90              4s 14ms/step
```

```
[42]: # Sort the dictionary by index for consistent ordering
sorted_indices = sorted(individual_accuracies.keys())
accuracies = [individual_accuracies[i] for i in sorted_indices]
letters = [index_to_letter.get(i, 'Unknown') for i in sorted_indices]

# Print individual accuracies nicely with corresponding letters
print(f"{'Letter':<10} {'Label':<10} {'Accuracy':<10}")
print("-" * 30)
for label, acc in sorted(individual_accuracies_rounded.items()):
    letter = index_to_letter.get(label, 'Unknown')
    print(f"{letter:<10} {label:<10} {acc:<10.4f}")
```

```
# Create the bar chart
plt.figure(figsize=(12, 6))
plt.bar(letters, accuracies, color='skyblue')
plt.ylim(0, 1.05)  # Ensure the y-axis covers from 0 to just above 1 for better␣
 ↪visibility

# Add a horizontal line at 85% accuracy
plt.axhline(0.85, color='red', linestyle='--', linewidth=2, label='85%␣
 ↪Accuracy')

# Add labels, title, and legend
plt.xlabel('Letter')
plt.ylabel('Accuracy')
plt.title('Individual Letter Accuracies')
plt.legend()

# Display the plot
plt.show()

# Calculate additional statistics
median_accuracy = np.median(list(individual_accuracies.values()))
max_acc_label = max(individual_accuracies, key=individual_accuracies.get)

# Find the highest accuracy value
max_acc = max(individual_accuracies.values())
# Find all labels with the highest accuracy
max_acc_labels = [label for label, acc in individual_accuracies.items() if acc␣
 ↪== max_acc]

# Format the letters with the highest accuracy
letters_with_max_acc = [index_to_letter.get(label, 'Unknown') for label in␣
 ↪max_acc_labels]
letters_str = ', '.join(letters_with_max_acc)

# Find labels with the lowest accuracy
min_acc_label = min(individual_accuracies, key=individual_accuracies.get)

# Get letter representations
max_acc_label_letter = index_to_letter.get(max_acc_label, 'Unknown')
min_acc_label_letter = index_to_letter.get(min_acc_label, 'Unknown')

print(f'Median Accuracy: {median_accuracy:.4f}')
print(f"\nLetter(s) with Highest Accuracy: {letters_str} (Accuracy: {max_acc:.
 ↪4f})") # Print the letters with the highest accuracy
print(f'Letter with Lowest Accuracy: {min_acc_label_letter} (Accuracy:␣
 ↪{individual_accuracies[min_acc_label]:.4f})\n') # Print the letter with the␣
 ↪lowest accuracy
```

```
Letter     Label     Accuracy
------------------------------
A          0         1.0000
B          1         1.0000
C          2         1.0000
D          3         0.9300
E          4         1.0000
F          5         1.0000
G          6         1.0000
H          7         1.0000
I          8         1.0000
K          10        1.0000
L          11        1.0000
M          12        1.0000
N          13        0.9407
O          14        1.0000
P          15        1.0000
Q          16        1.0000
R          17        0.8793
S          18        1.0000
T          19        0.9307
U          20        0.9911
V          21        0.9930
W          22        1.0000
X          23        0.9182
Y          24        0.9083
```

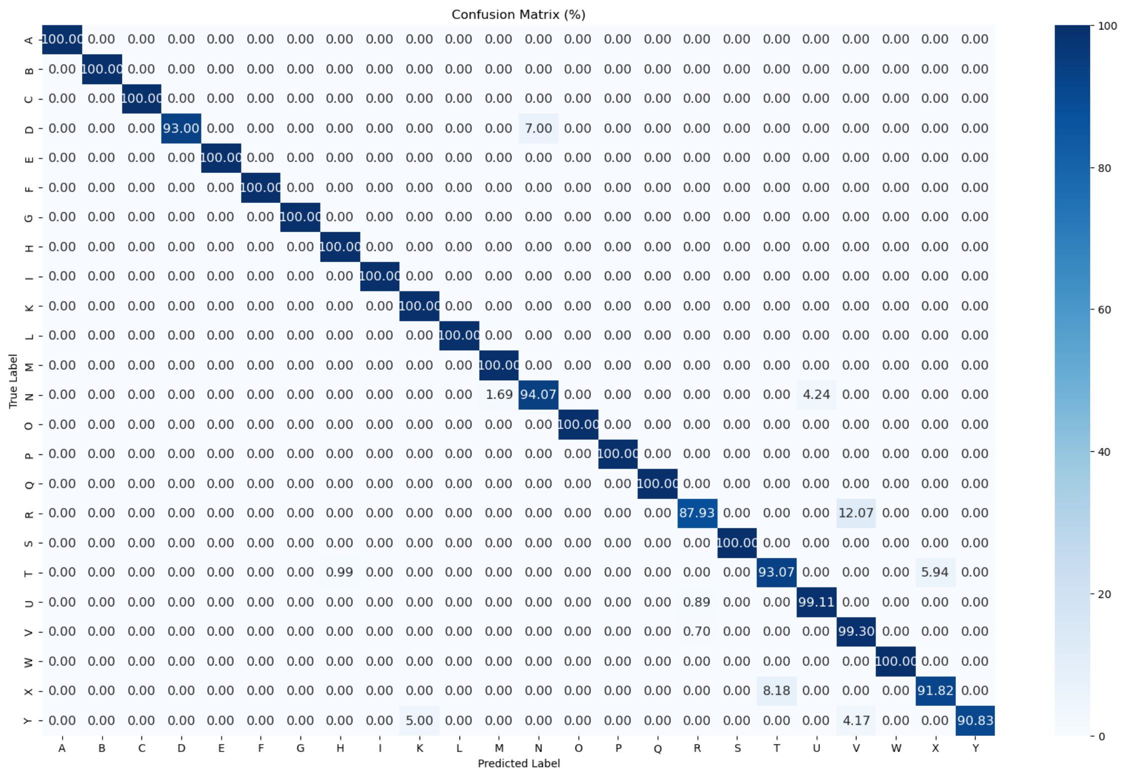


```
Median Accuracy: 1.0000
```

```
Letter(s) with Highest Accuracy: A, B, C, E, F, G, H, I, K, L, M, O, P, Q, S, W
(Accuracy: 1.0000)
Letter with Lowest Accuracy: R (Accuracy: 0.8793)
```

[43]:

```
# Generate confusion matrix with true labels and predicted labels
valid_labels = sorted(set(true_labels) | set(pred_labels))  # Get the union of␣
 ↪true and predicted labels
conf_matrix = confusion_matrix(true_labels, pred_labels, normalize='true',␣
 ↪labels=valid_labels) * 100
valid_index_to_letter = {i: index_to_letter[i] for i in valid_labels}

# Plot confusion matrix
plt.figure(figsize=(20, 12))
sns.heatmap(conf_matrix, annot=True, fmt='.2f', cmap='Blues',
            xticklabels=[valid_index_to_letter[i] for i in valid_labels],
            yticklabels=[valid_index_to_letter[i] for i in valid_labels],
            annot_kws={"size": 12})
plt.title('Confusion Matrix (%)')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

# Calculate errors by zeroing out diagonal values in confusion matrix
errors = conf_matrix.astype(float)
np.fill_diagonal(errors, 0)  # Zero out diagonal for error calculation

# Find the indices of the top 3 most common errors
top_3_errors_indices = np.unravel_index(np.argsort(errors.ravel())[-3:], errors.
 ↪shape)

# Collect top 3 errors and their respective misclassification percentages
top_3_errors = []
for i, j in zip(top_3_errors_indices[0], top_3_errors_indices[1]):
    top_3_errors.append((valid_index_to_letter[valid_labels[i]],␣
 ↪valid_index_to_letter[valid_labels[j]], errors[i, j]))

# Print the top 3 most common errors
print('Top 3 Most Common Errors:')
for error in top_3_errors:
    print(f"Letter '{error[0]}' misclassified as '{error[1]}' with a percentage␣
 ↪of {error[2]:.2f}%")
```



```
Top 3 Most Common Errors:
Letter 'D' misclassified as 'N' with a percentage of 7.00%
Letter 'X' misclassified as 'T' with a percentage of 8.18%
Letter 'R' misclassified as 'V' with a percentage of 12.07%
```

The model achieves an impressive overall accuracy of 98.26% and a median accuracy of 100%, meeting the 94% overall mean accuracy requirement and exceeding 85% accuracy for each individual letter. It performs exceptionally well for many letters. However, it struggles with letters some letters. To address these weaknesses, further optimization and fine-tuning, such as improving regularization techniques or adjusting hyperparameters, are recommended. With targeted improvements, the model has the potential to enhance performance and more effectively achieve the desired accuracy goals.

(81 words)