Deep Learning

June 15, 2025

1 Gesture Recognition using Deep Learning

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:
```
import os
import glob
import random
import itertools
import seaborn as sns

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
import random
import tensorflow
import tensorflow as tf
import shap


from tensorflow.keras.callbacks import Callback
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Conv1D, MaxPooling1D, BatchNormalization,
    Flatten, Dense, Dropout,GaussianNoise, SpatialDropout1D, Input
)

from tensorflow.keras.optimizers.schedules import CosineDecay
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau,␣
 ↪LearningRateScheduler
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.regularizers import l2
```

```
from sklearn.metrics import confusion_matrix, classification_report

from collections import defaultdict, Counter


np.random.seed(42)
```

1.1 Preprocessing

Same procedure as Data Exploratory Code

[ ]:
```
root_dir = "/content/drive/MyDrive/IA_EBM/PROJECT/EMG_data_for_gestures-master"
all_txt = glob.glob(os.path.join(root_dir, "**", "*.txt"), recursive=True)

dfs = []
for f in tqdm(all_txt, desc="Loading .txt files"):
    dfs.append(pd.read_csv(f, sep="\t"))
combined_df = pd.concat(dfs, ignore_index=True)


df = combined_df[(combined_df['class'] != 0) & (combined_df['class'] != 7)].
 ↪reset_index(drop=True)

df['segment'] = (df['class'] != df['class'].shift()).cumsum()
gesture_datasets = {}
for (g_class, seg), group in tqdm(df.groupby(['class', 'segment']),␣
 ↪desc="Segmenting gestures"):
    key = f"gesture_{int(g_class)}_{seg}"
    gesture_datasets[key] = group.drop(columns=['segment']).
 ↪reset_index(drop=True)


classes = sorted({int(k.split('_')[1]) for k in gesture_datasets})
train_segs, val_segs, test_segs = [], [], []
for cls in tqdm(classes, desc="Splitting segments"):
    seg_keys = [k for k in gesture_datasets if int(k.split('_')[1]) == cls]
    random.shuffle(seg_keys)
    n = len(seg_keys)
    n_train = int(0.6 * n)
    n_val   = int(0.2 * n)
    train_segs += seg_keys[:n_train]
    val_segs   += seg_keys[n_train:n_train+n_val]
    test_segs  += seg_keys[n_train+n_val:]
print(f"Segments → train: {len(train_segs)}, val: {len(val_segs)}, test:␣
 ↪{len(test_segs)}")
```

```
def extract_deterministic_windows(signal, window_size=400, step=100):   #FROM␣
 ↪THE SECOND HYPERPARAMETER TUNNING
    return [signal[:, i:i+window_size]
            for i in range(0, signal.shape[1] - window_size + 1, step)]

train_w, val_w, test_w = {}, {}, {}
WINDOW_SIZE = 400

for key in tqdm(gesture_datasets, desc="Extracting windows"):
    gdf = gesture_datasets[key]
    cols = [c for c in gdf.columns if c.startswith("channel")]
    sig  = gdf[cols].to_numpy().T   # (channels, samples)

    # choose step based on split
    if key in train_segs:
        step = 100                # 75% overlap for training
        target = train_w
    else:
        step = WINDOW_SIZE        # no overlap for val/test
        target = val_w if key in val_segs else test_w

    # slide window
    for i in range(0, sig.shape[1] - WINDOW_SIZE + 1, step):
        w = sig[:, i : i + WINDOW_SIZE]
        target[f"{key}_win_{i//step}"] = w


print("Train windows:", len(train_w),
      "Val windows:",   len(val_w),
      "Test windows:",  len(test_w))



def extract_label(k):
    return int(k.split('_')[1])

def prepare(wdict):
    X = np.stack([w.T for w in wdict.values()], axis=0)  # (n_samples, window,␣
 ↪channels)
    y = np.array([extract_label(k) for k in wdict.keys()])
    return X, y

X_train, y_train = prepare(train_w)
X_val,   y_val   = prepare(val_w)
X_test,  y_test  = prepare(test_w)
```

```
mean = X_train.mean(axis=(0,1), keepdims=True)
std  = X_train.std(axis=(0,1), keepdims=True) + 1e-8
for arr in tqdm((X_train, X_val, X_test), desc="Normalizing datasets"):
    arr[:] = (arr - mean) / std



classes     = np.unique(y_train)
num_classes = len(classes)
offset      = classes.min()
y_train_enc = to_categorical(y_train - offset, num_classes)
y_val_enc   = to_categorical(y_val   - offset, num_classes)
y_test_enc  = to_categorical(y_test  - offset, num_classes)
```

```
Loading .txt files: 100%|      | 73/73 [00:06<00:00, 11.86it/s]
Segmenting gestures:  98%|      | 864/883 [00:00<00:00, 1231.00it/s]
Splitting segments: 100%|      | 6/6 [00:00<00:00, 1707.43it/s]

Segments → train: 516, val: 168, test: 180

Extracting windows: 100%|      | 864/864 [00:00<00:00, 1764.30it/s]

Train windows: 7135 Val windows: 638 Test windows: 692

Normalizing datasets: 100%|      | 3/3 [00:00<00:00, 10.77it/s]
```

1.2 Inference Mode Evaluation

During training, layers like Dropout and BatchNormalization behave differently than at inference time:

- **Dropout** randomly deactivates units.
- **BatchNormalization** uses batch statistics rather than the running averages used during deployment.

To get a realistic view of how our model will perform in production, we evaluate it in **inference mode** (using `model.evaluate()`).

This ensures: - Consistent and stable accuracy readings. - No misleading performance due to training-phase randomness. - Accurate tracking of true generalization to validation data.

[ ]:
```
class EvalTrainVal(Callback):
    def __init__(self, train_data, val_data):
        super().__init__()
        self.train_data = train_data
        self.val_data   = val_data
        self.eval_metrics = {'train_acc': [], 'val_acc': []}

    def on_epoch_end(self, epoch, logs=None):
        # run inference-mode eval on both
```

```
        _, tr_acc = self.model.evaluate(*self.train_data, verbose=0)
        _, va_acc = self.model.evaluate(*self.val_data,   verbose=0)
        self.eval_metrics['train_acc'].append(tr_acc)
        self.eval_metrics['val_acc'].append(va_acc)
        print(f" - eval_train_acc: {tr_acc:.4f}, eval_val_acc: {va_acc:.4f}")
```

[ ]:
```
timesteps, n_channels = X_train.shape[1], X_train.shape[2]
num_classes = y_train_enc.shape[1]
```

1.3 Build and Compile the 1D-CNN Model

After building this CNN model, we really struggled with overfitting.

To improve generalization and reduce overfitting, beside reducing the general complexity of the model, we apply several regularization techniques throughout the network:

- **L2 Regularization (Weight Decay):**
  Applied to all convolutional and dense layers to penalize large weights and encourage simpler, more robust models.
- **Dropout & SpatialDropout1D:**
  We use `Dropout` after dense layers and `SpatialDropout1D` in convolutional layers. This randomly deactivates parts of the network during training, reducing reliance on specific neurons and preventing co-adaptation.
- **Batch Normalization:**
  Added after convolutional layers to stabilize training and provide regularization by reducing internal covariate shift.
- **EarlyStopping:**
  Training stops automatically if validation loss doesn't improve for a set number of epochs, preventing the model from overfitting to the training data in later stages.
- **ReduceLROnPlateau:**
  If the validation loss plateaus, the learning rate is reduced to allow the optimizer to make finer adjustments. This can help escape flat regions or local minima without overfitting.
- **Segment-wise Data Splitting:**
  Instead of randomly splitting windows, we split at the gesture segment level. This ensures that no part of a single gesture appears in both training and validation/test sets, eliminating data leakage.

[ ]:
```
def build_model(filters, kernel_size, dropout_rate, weight_decay, dense_units,␣
↪lr):
    """
    Build and compile a 1D-CNN with the given hyperparameters.
    """
    model = Sequential([
        Input(shape=(timesteps, n_channels)),

        Conv1D(filters, kernel_size,
```

```
                activation='relu', kernel_regularizer=l2(weight_decay)),
        BatchNormalization(),
        SpatialDropout1D(dropout_rate),

        Conv1D(filters, kernel_size,
                activation='relu', kernel_regularizer=l2(weight_decay)),
        BatchNormalization(),
        MaxPooling1D(2),
        Dropout(dropout_rate),

        Conv1D(filters * 2, kernel_size - 2,
                activation='relu', kernel_regularizer=l2(weight_decay)),
        MaxPooling1D(2),
        Dropout(dropout_rate),

        Flatten(),
        Dense(dense_units,
              activation='relu', kernel_regularizer=l2(weight_decay)),
        Dropout(dropout_rate),

        Dense(num_classes, activation='softmax')
    ])

    optimizer = Adam(learning_rate=lr)
    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

1.4 Hyperparameter Search Loop

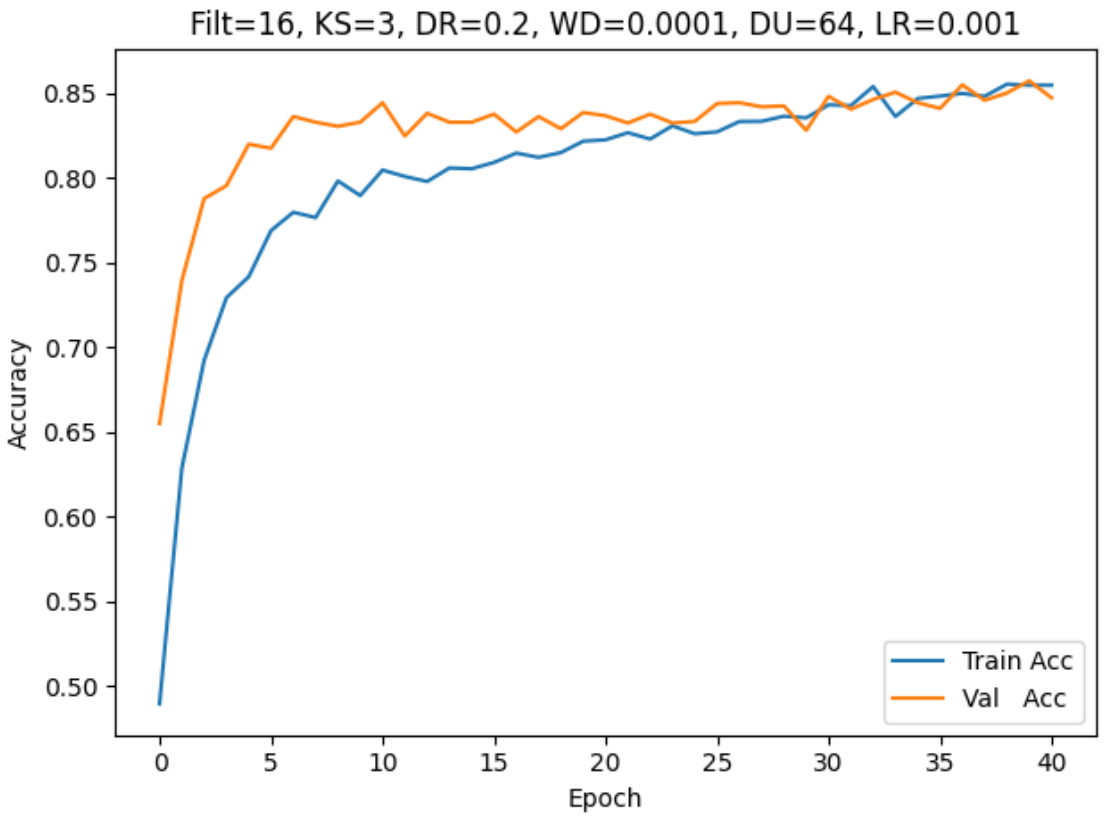
- Iterates over filter sizes, kernel sizes, dropout rates, weight decay, dense units, and learning rates.

For the chose of parameters, we've not selected that with the highest performance in numerical terms but the one it had a most adequate accuracy curve.

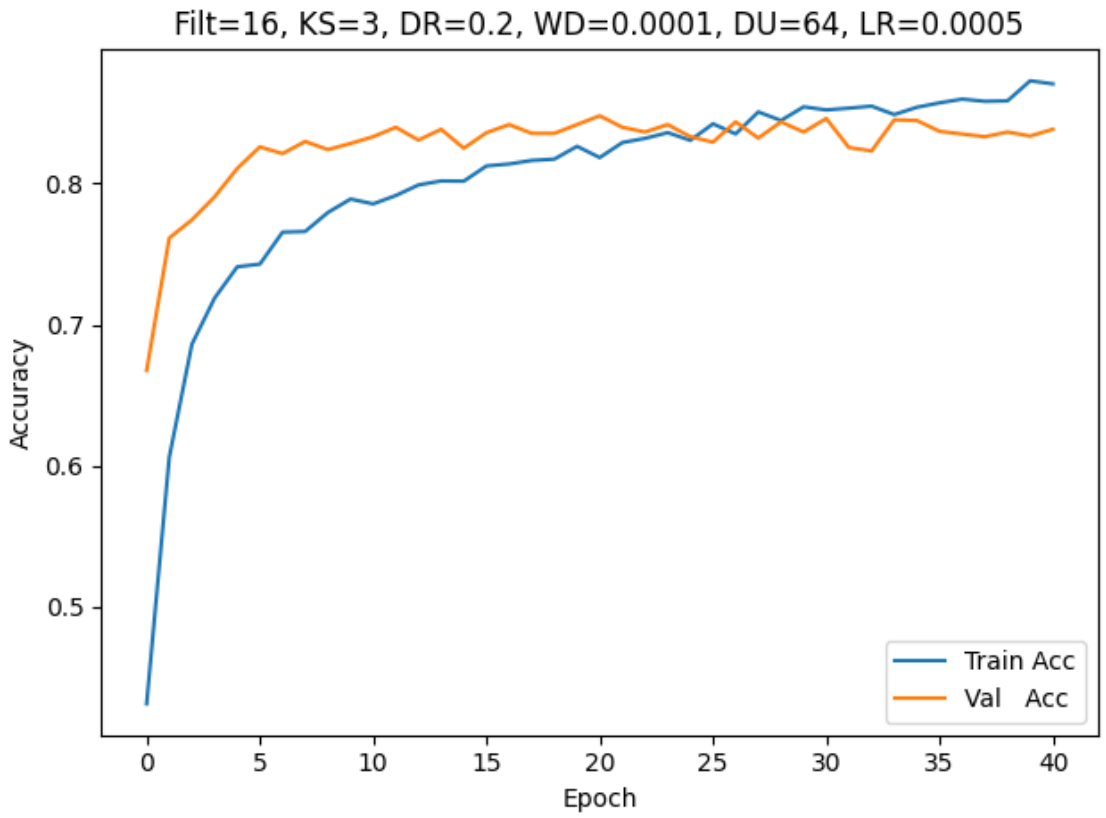
[ ]:
```
# Hyperparameter grid
grid = {
    'filters':      [16, 32],
    'kernel_size':  [3, 5],
    'dropout_rate': [0.2, 0.4],
    'weight_decay': [1e-4, 1e-3],
    'dense_units':  [64, 128],
    'lr':           [1e-3, 5e-4]
}
```

```
# Prepare all combinations
keys, values = zip(*grid.items())
combinations = [dict(zip(keys, v)) for v in itertools.product(*values)]

# EarlyStopping callback
cb_es = EarlyStopping(
    monitor='val_loss', patience=10, restore_best_weights=True, verbose=0
)

# Loop over hyperparameter combinations
for params in tqdm(combinations, desc="Hyperparam search"):

    f   = params['filters']
    ks  = params['kernel_size']
    dr  = params['dropout_rate']
    wd  = params['weight_decay']
    du  = params['dense_units']
    lr  = params['lr']

    # Build & train
    model = build_model(f, ks, dr, wd, du, lr)
    history = model.fit(
        X_train, y_train_enc,
        validation_data=(X_val, y_val_enc),
        epochs=50,
        batch_size=32,
        callbacks=[cb_es],
        verbose=0
    )

    # Plot train vs val accuracy
    plt.figure()
    plt.plot(history.history['accuracy'], label='Train Acc')
    plt.plot(history.history['val_accuracy'], label='Val   Acc')
    plt.title(f"Filt={f}, KS={ks}, DR={dr}, WD={wd}, DU={du}, LR={lr}")
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.tight_layout()
    plt.show()

    best_val = max(history.history['val_accuracy'])
    print(f"--> Best val_acc: {best_val:.4f} for config: {params}\n")
```

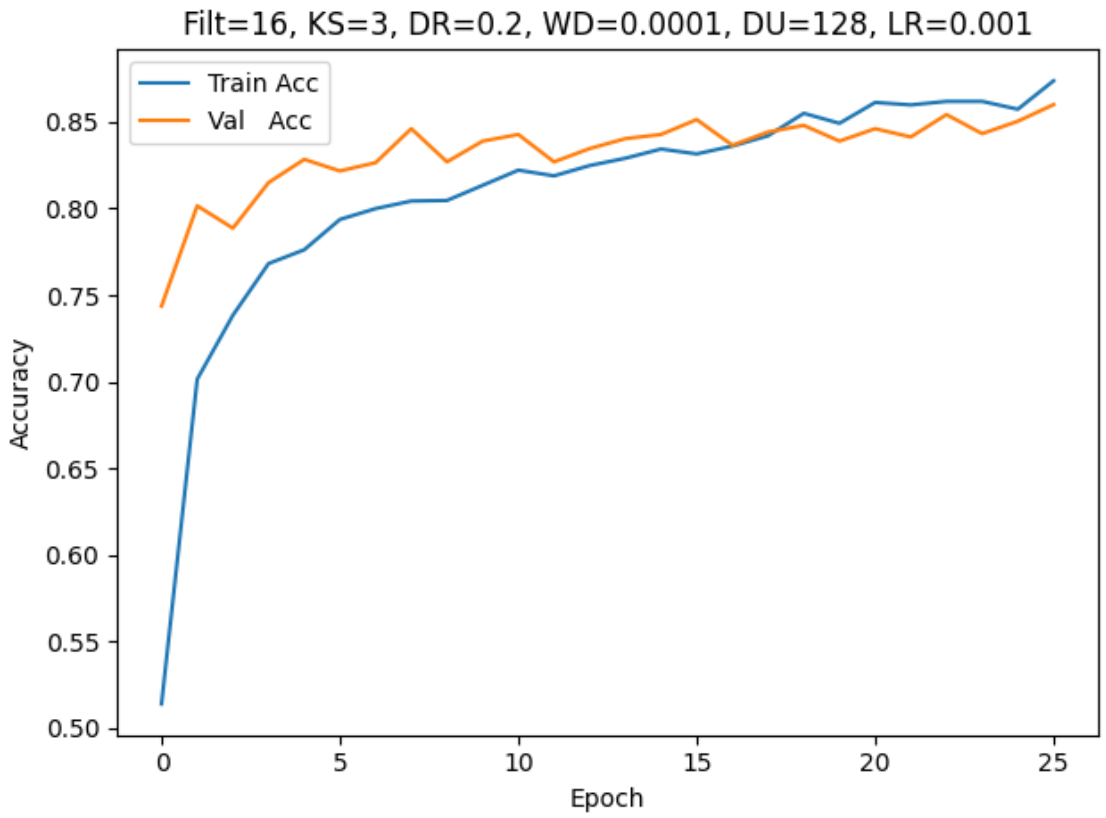
```
Hyperparam search:   0%|          | 0/64 [00:00<?, ?it/s]
```



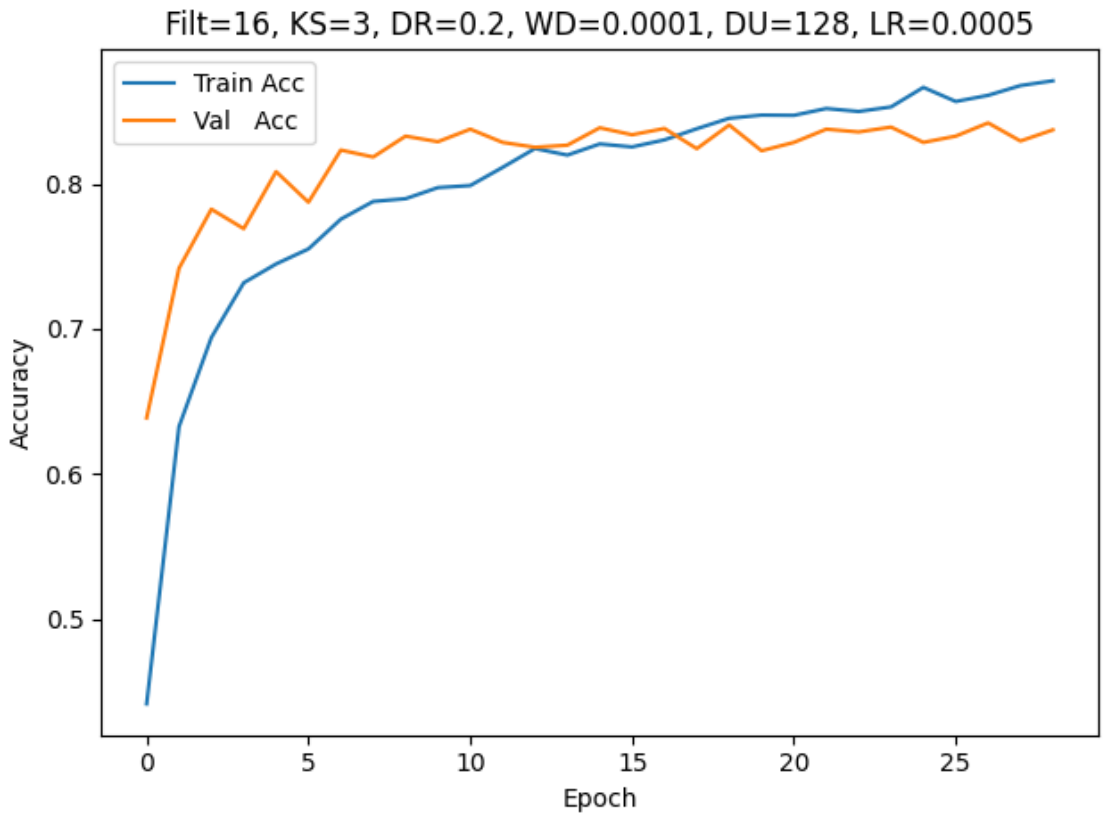

```
--> Best val_acc: 0.8572 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



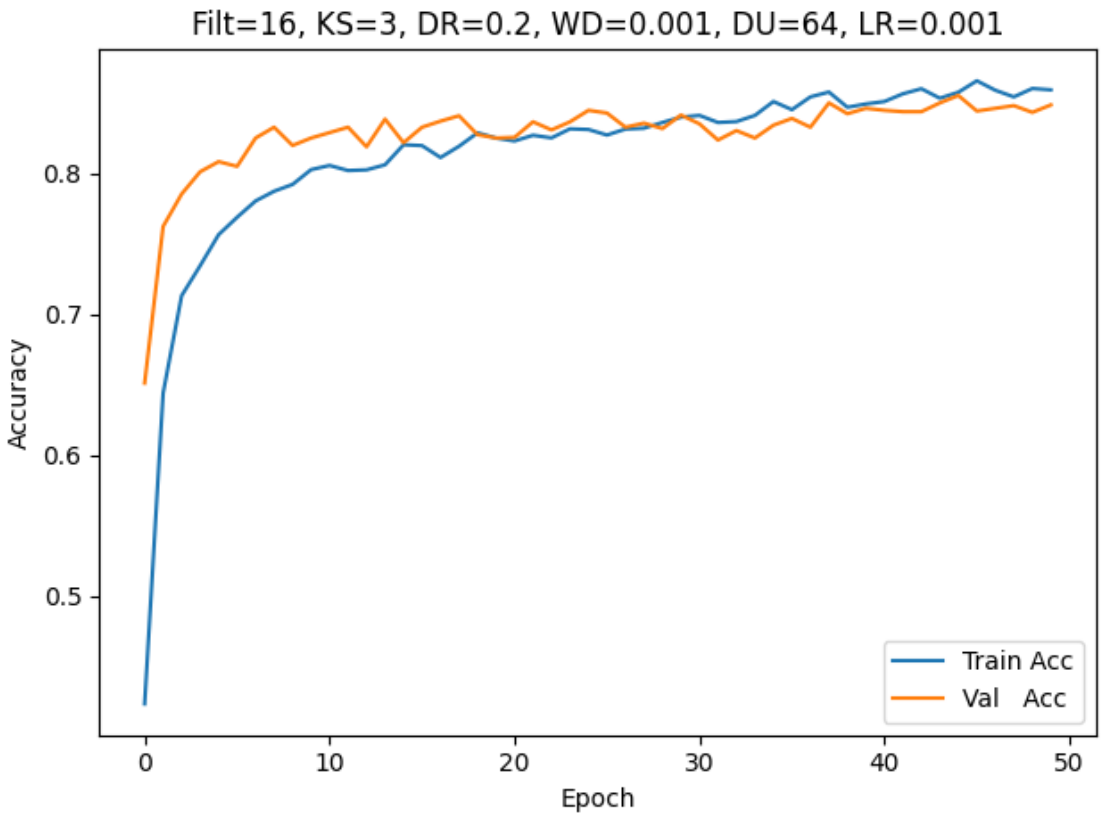
--> Best val_acc: 0.8481 for config: {'filters': 16, 'kernel_size': 3, 'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}



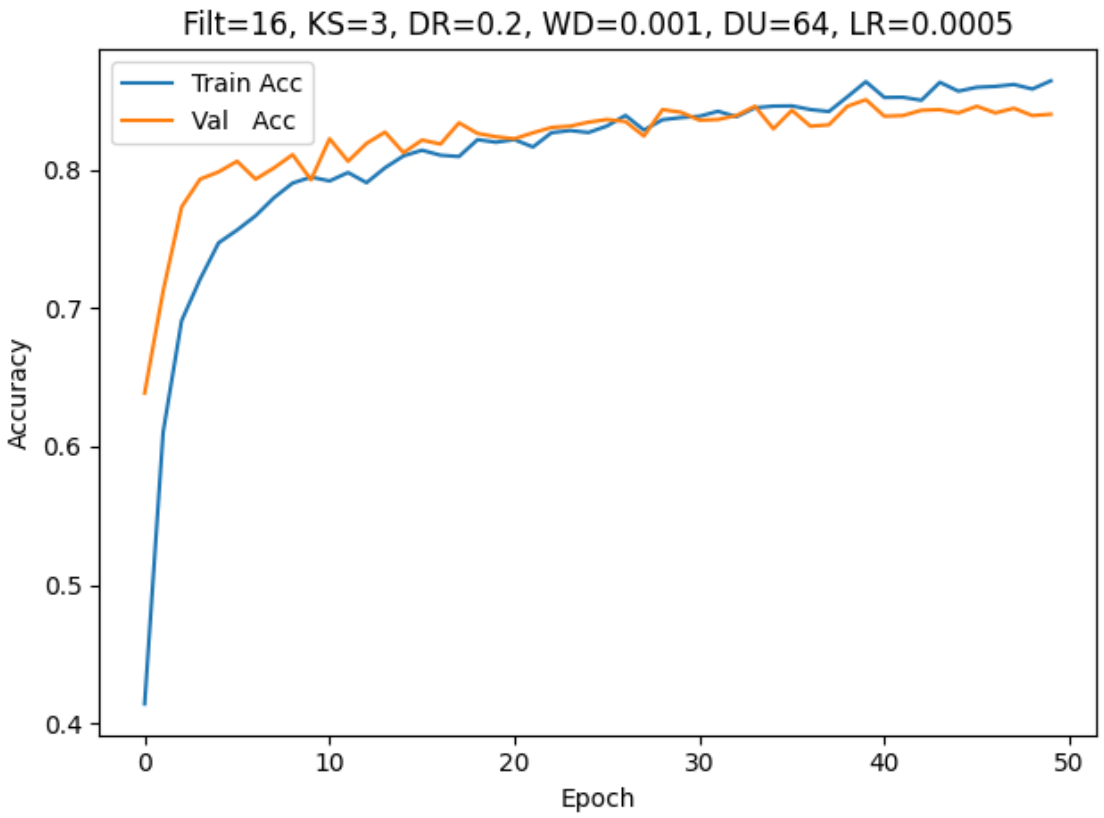

```
--> Best val_acc: 0.8601 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}
```



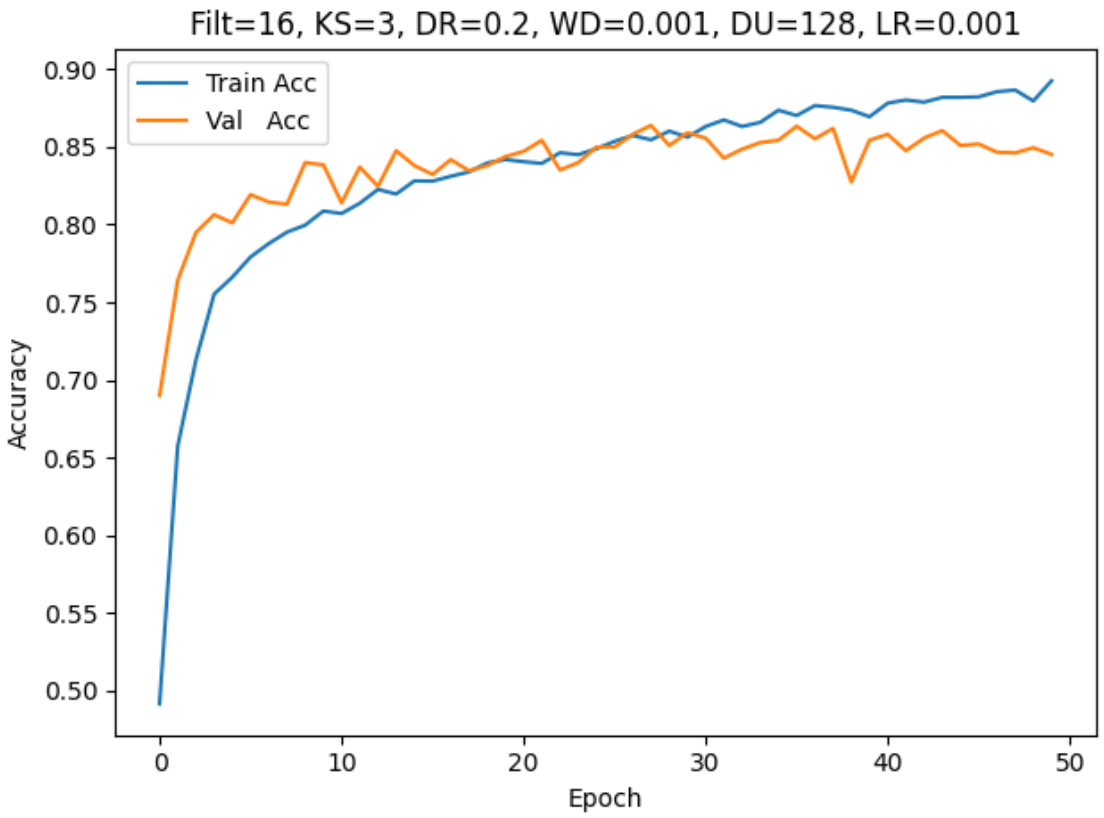

```
--> Best val_acc: 0.8419 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}
```



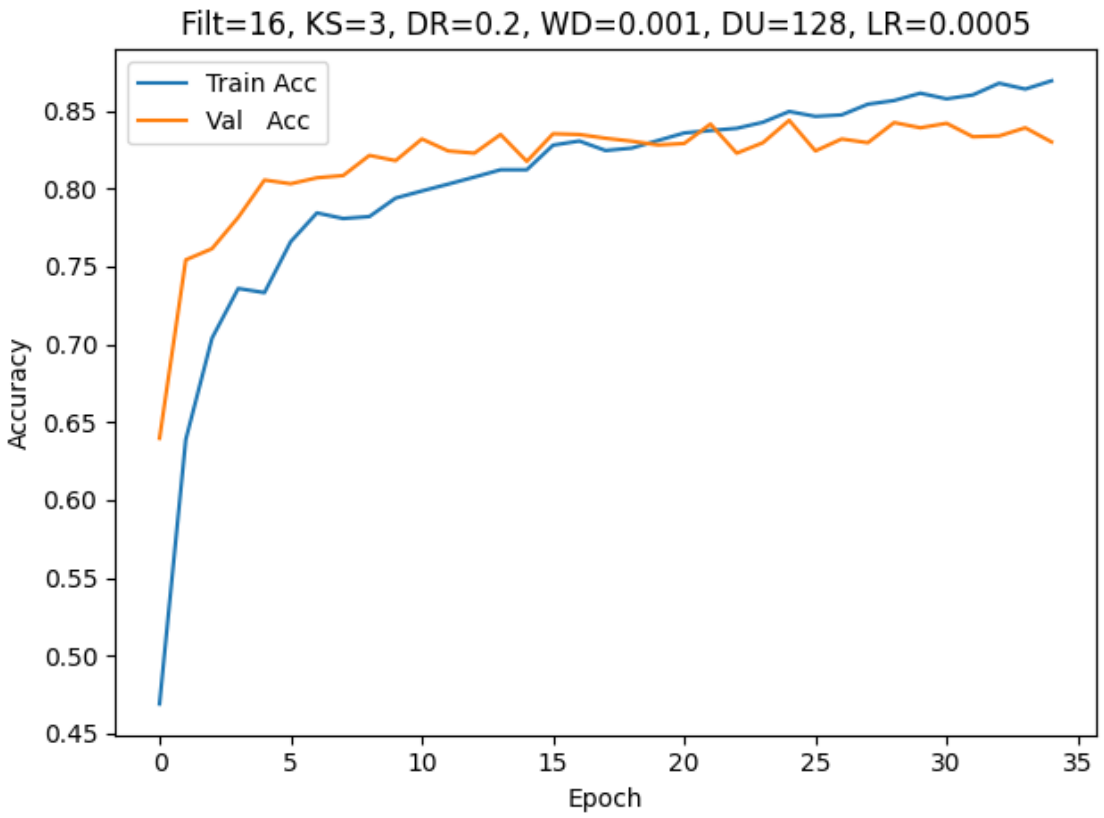

```
--> Best val_acc: 0.8553 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}
```



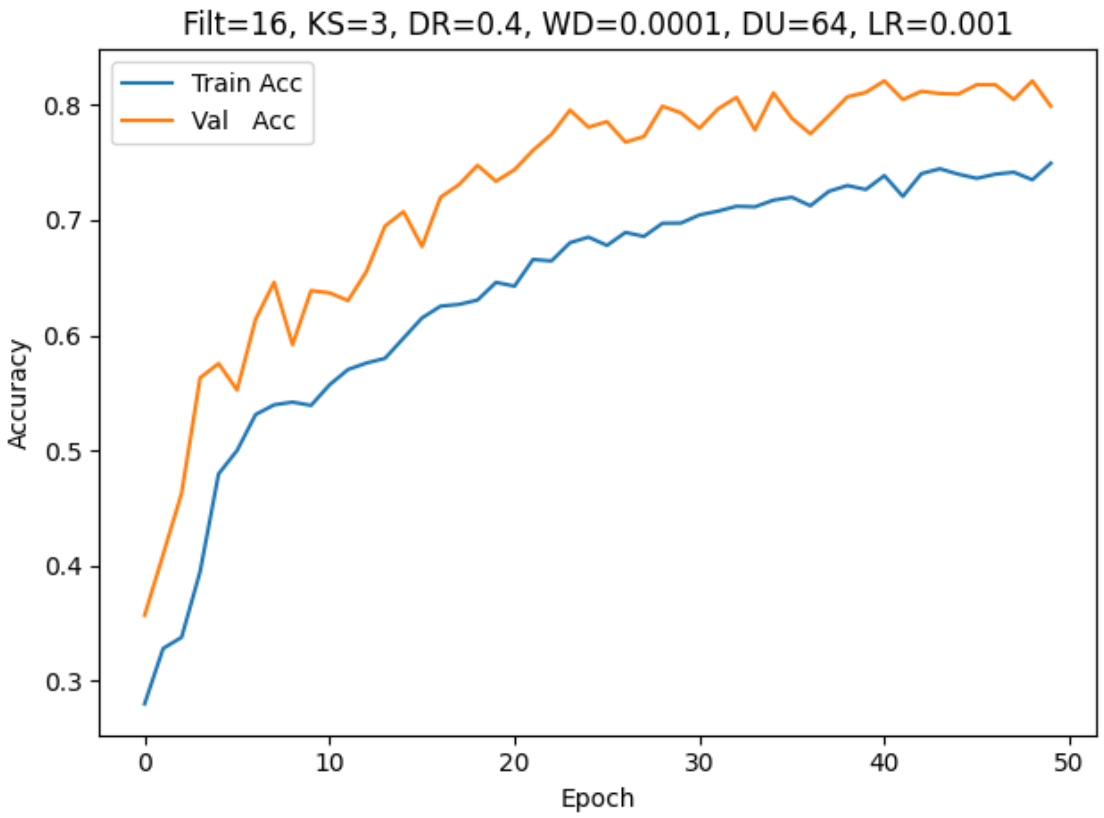
--> Best val_acc: 0.8510 for config: {'filters': 16, 'kernel_size': 3, 'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}



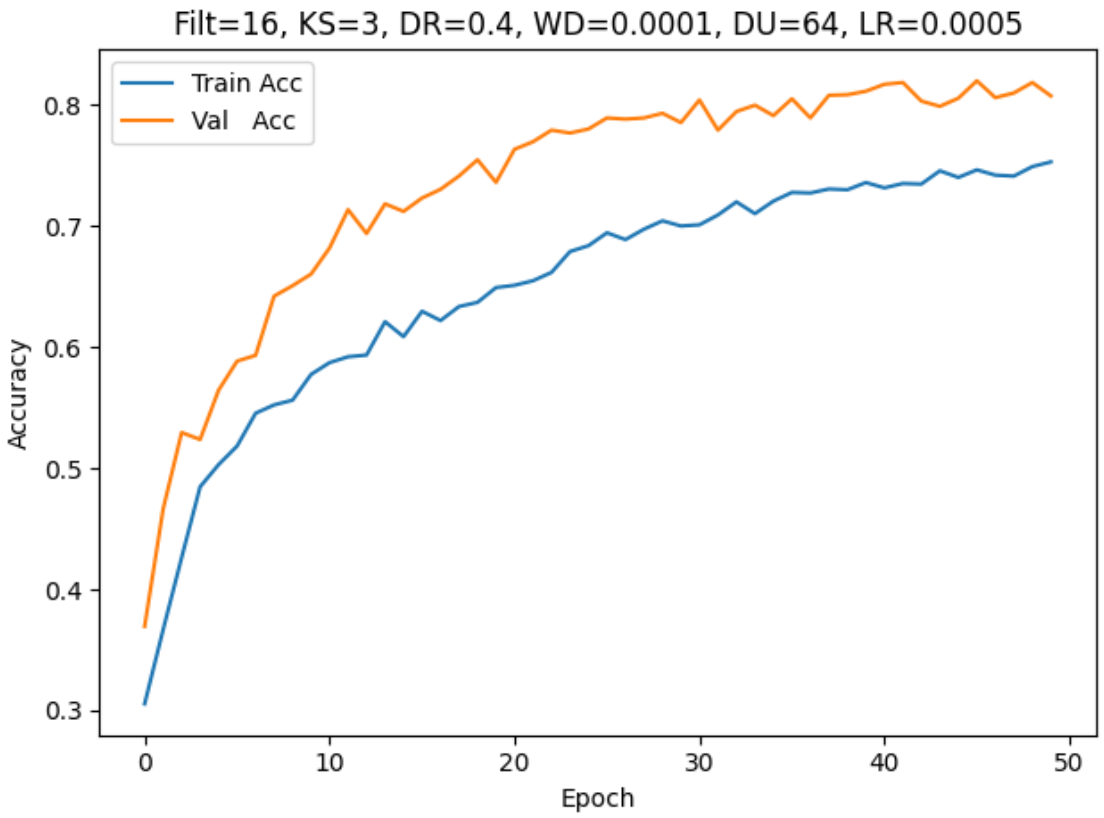

```
--> Best val_acc: 0.8639 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}
```



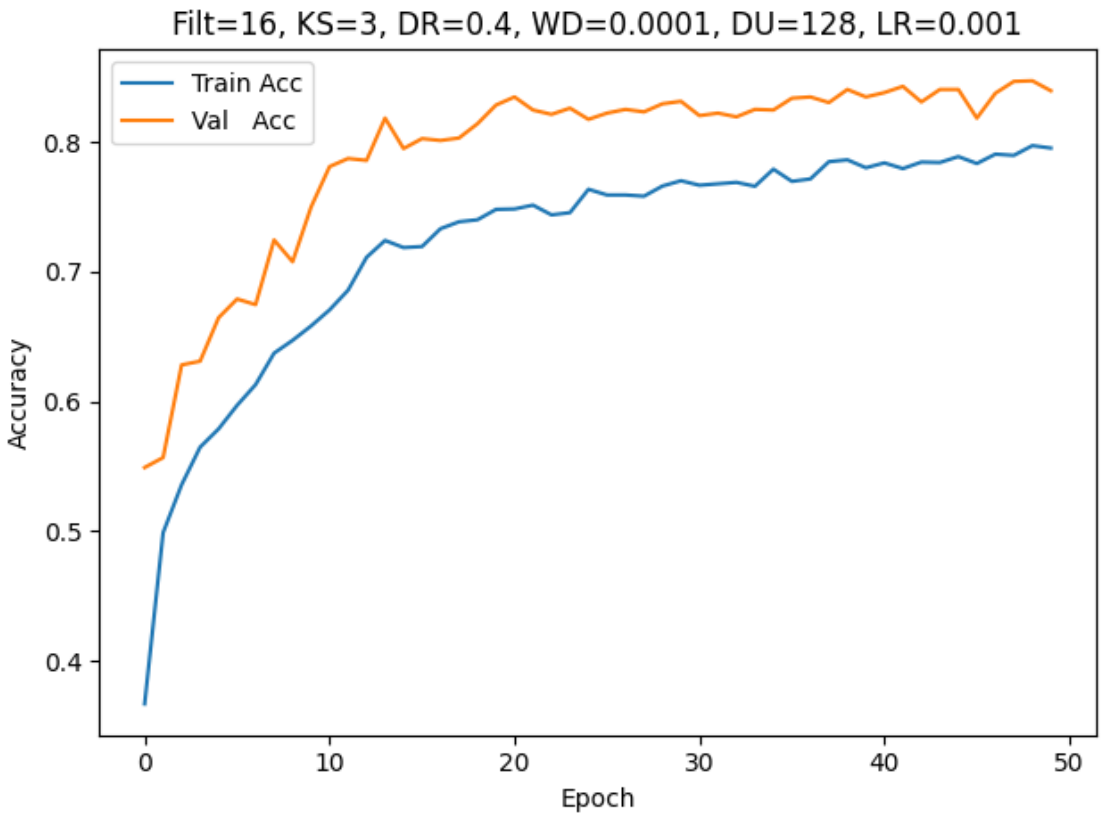

```
--> Best val_acc: 0.8438 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}
```



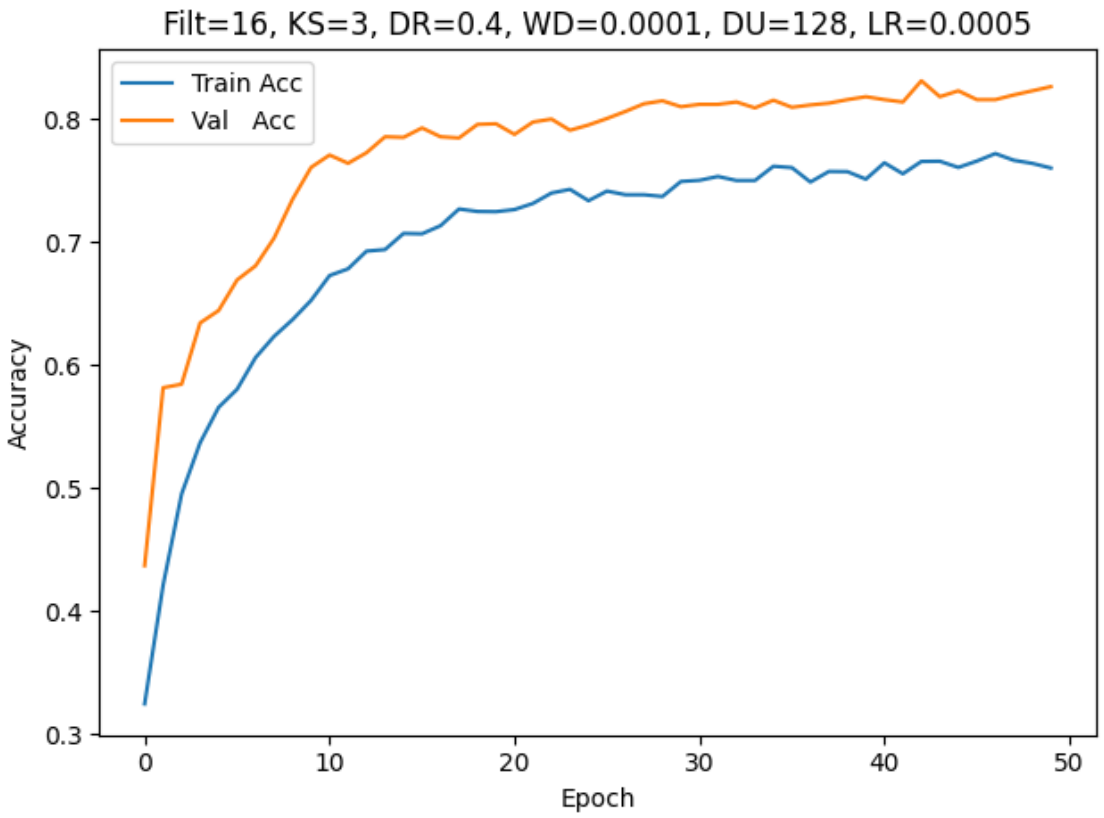

```
--> Best val_acc: 0.8208 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



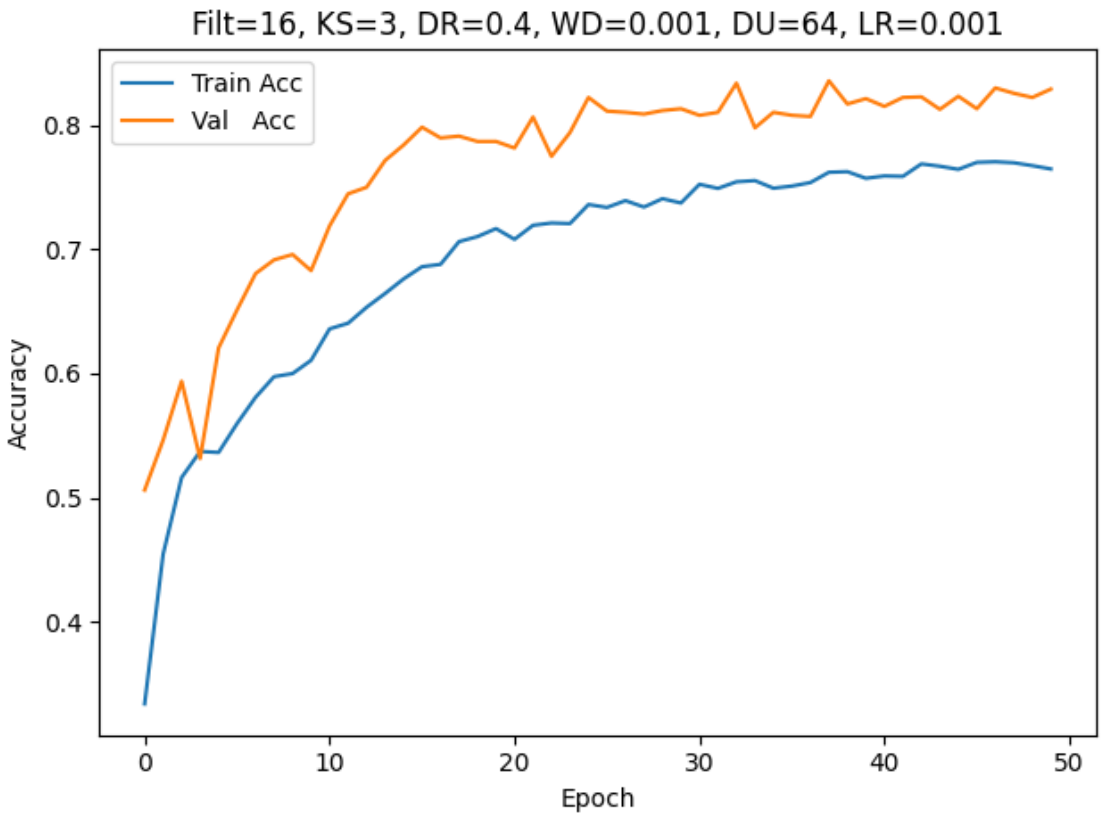

```
--> Best val_acc: 0.8198 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}
```



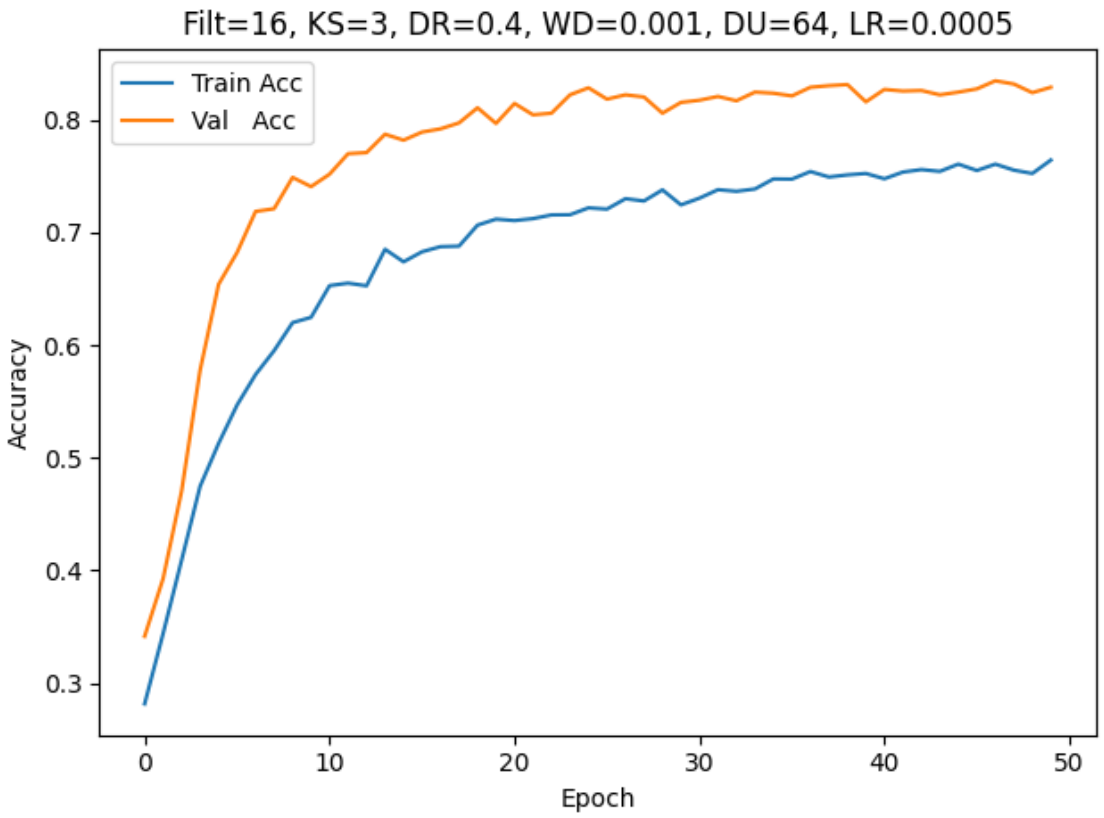

```
--> Best val_acc: 0.8471 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}
```



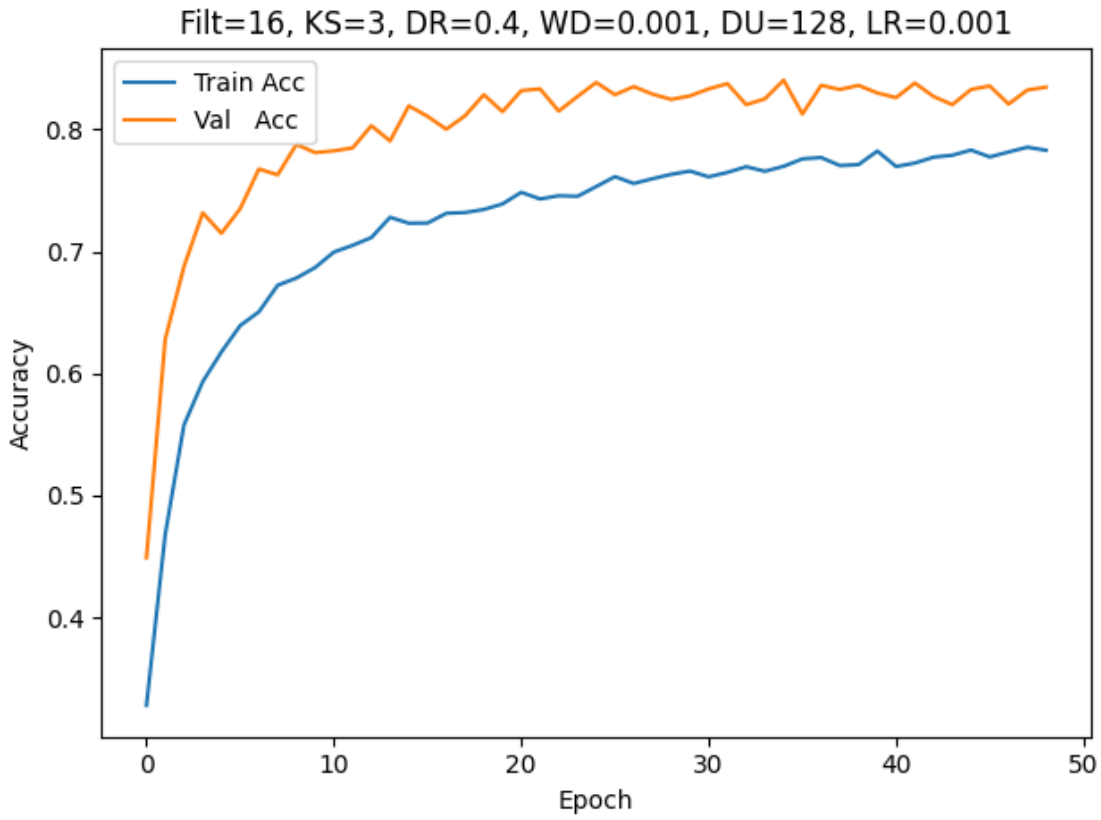
--> Best val_acc: 0.8313 for config: {'filters': 16, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}



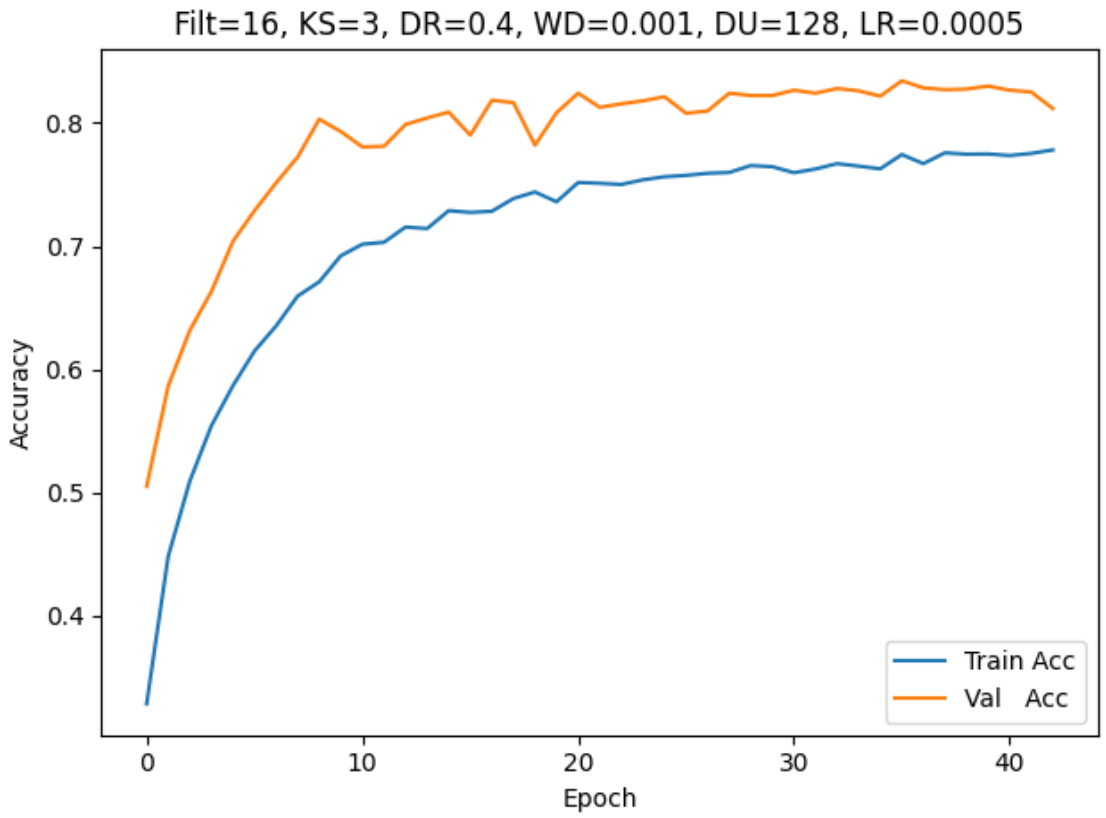

```
--> Best val_acc: 0.8356 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}
```



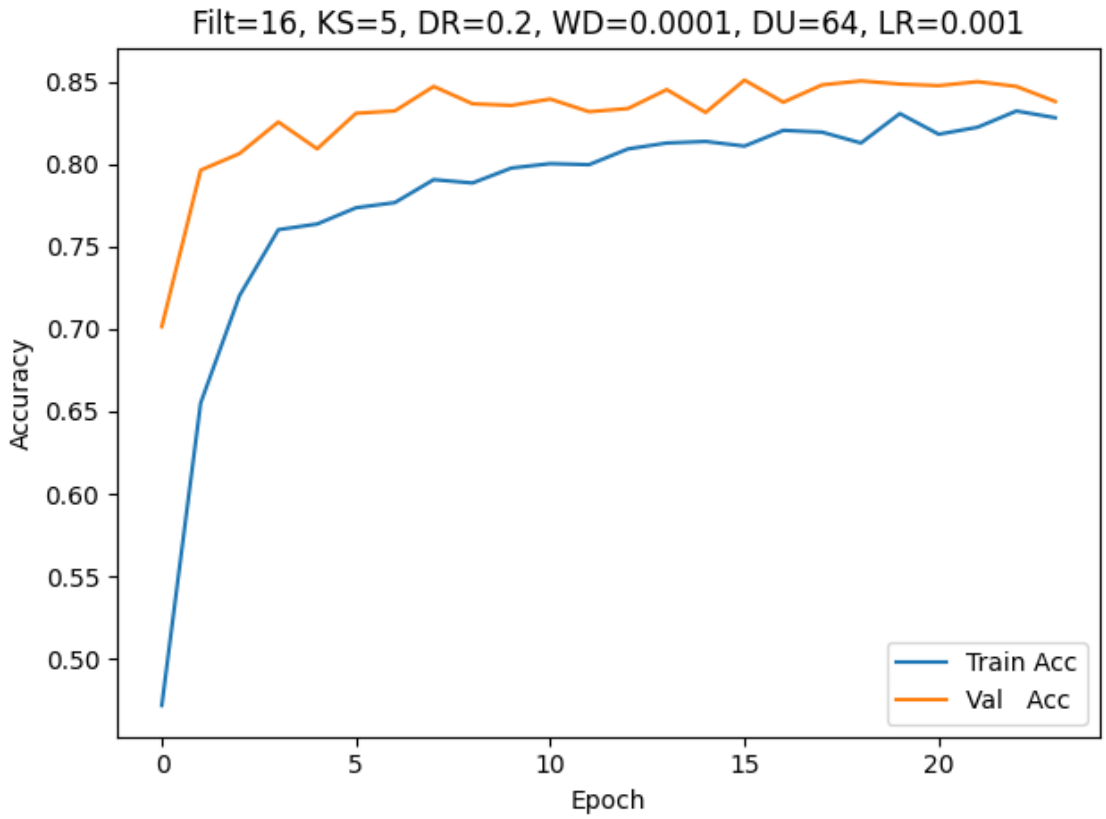
--> Best val_acc: 0.8347 for config: {'filters': 16, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}



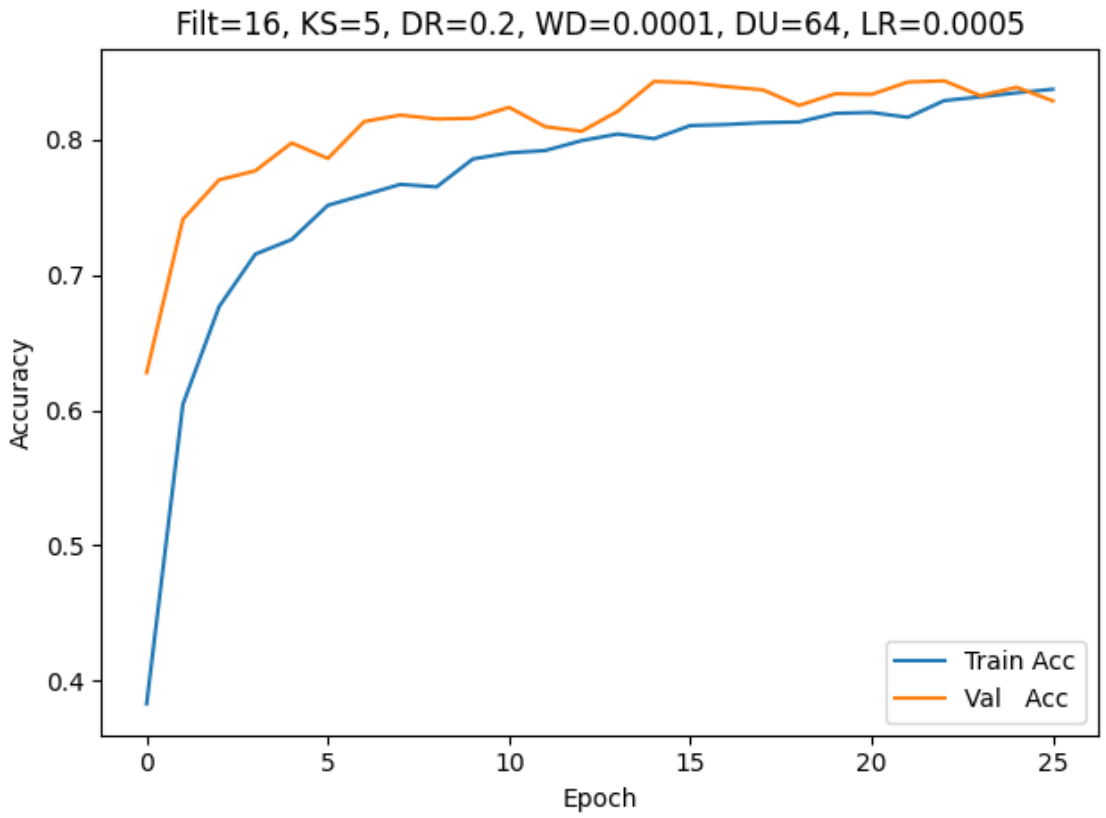

```
--> Best val_acc: 0.8404 for config: {'filters': 16, 'kernel_size': 3,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}
```



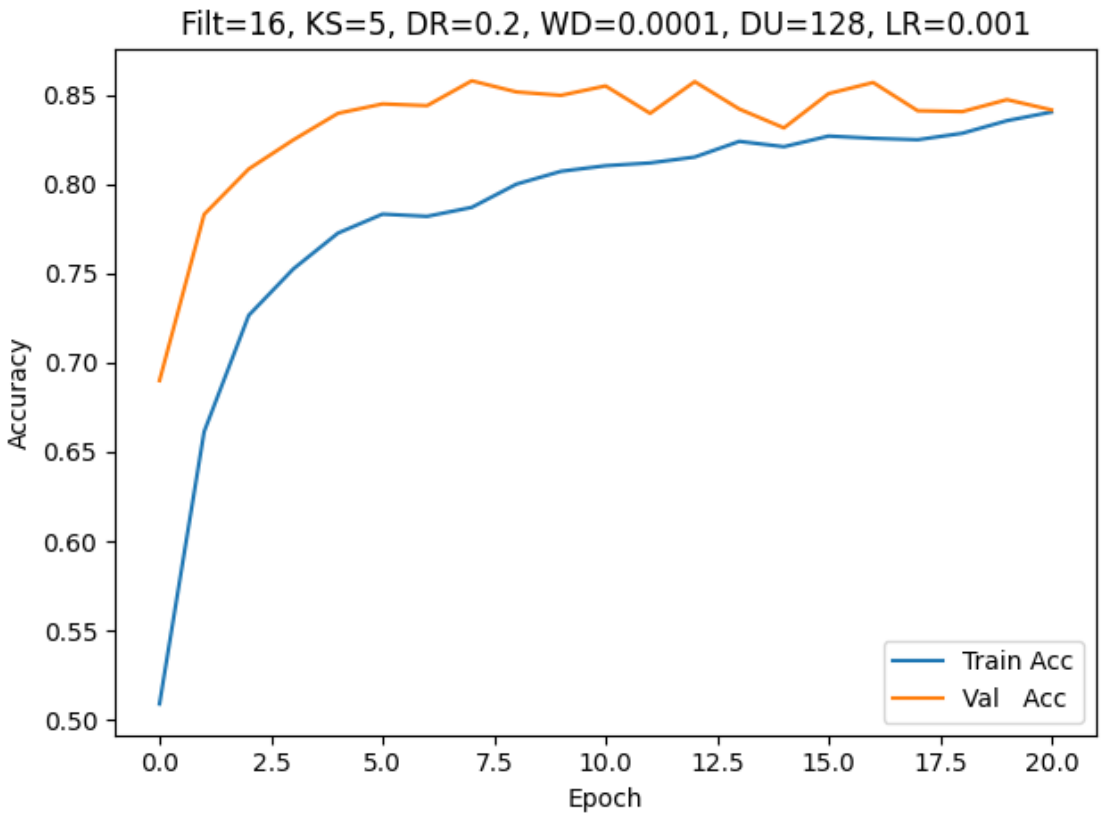
--> Best val_acc: 0.8342 for config: {'filters': 16, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}



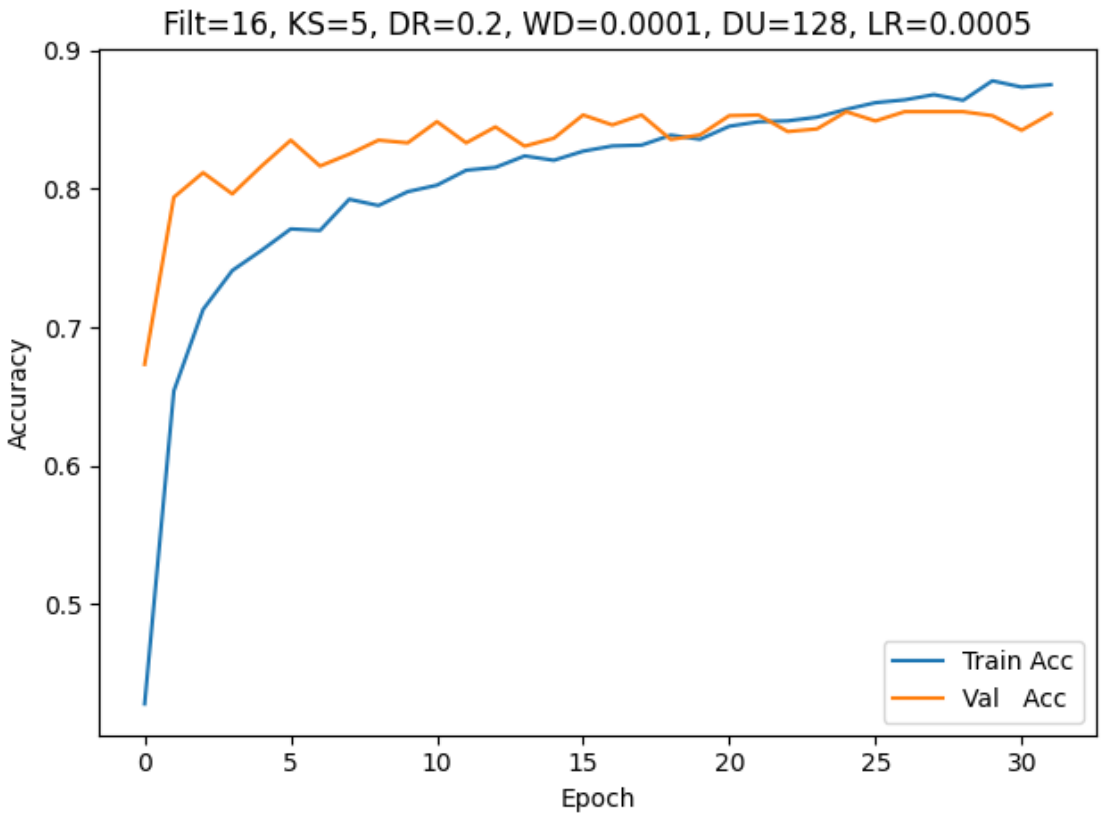

```
--> Best val_acc: 0.8510 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



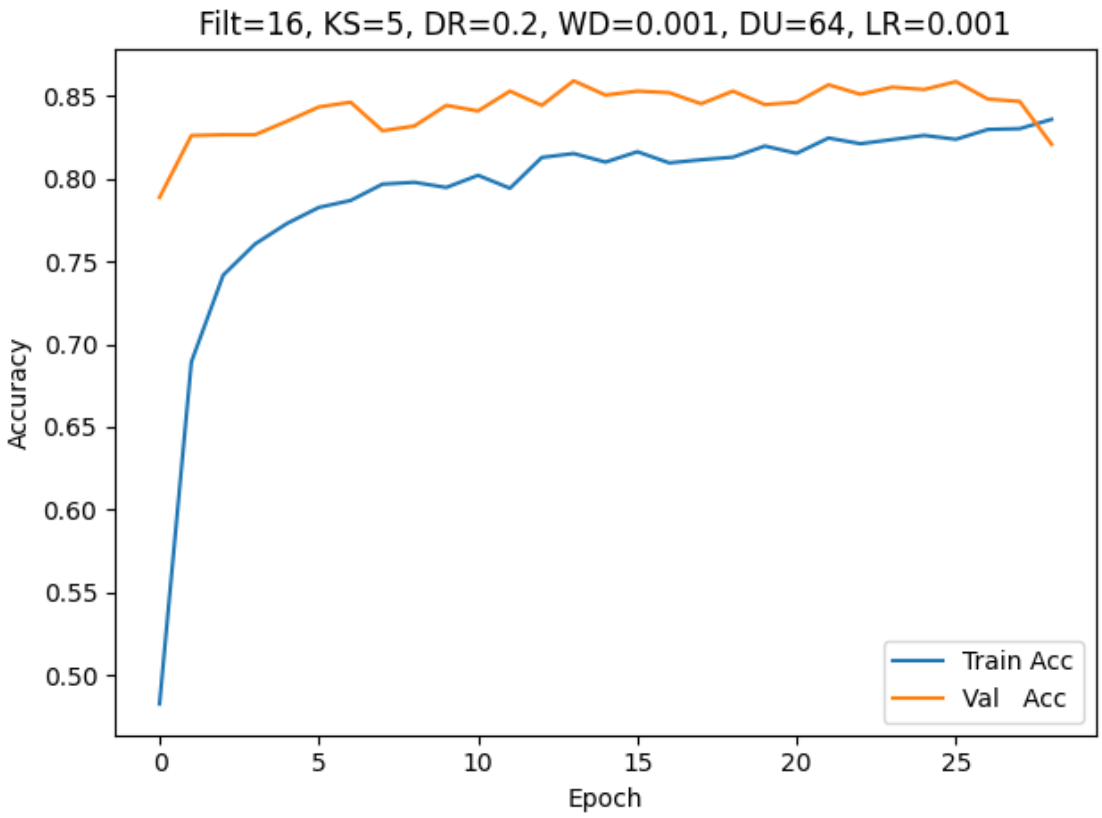
--> Best val_acc: 0.8438 for config: {'filters': 16, 'kernel_size': 5, 'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}



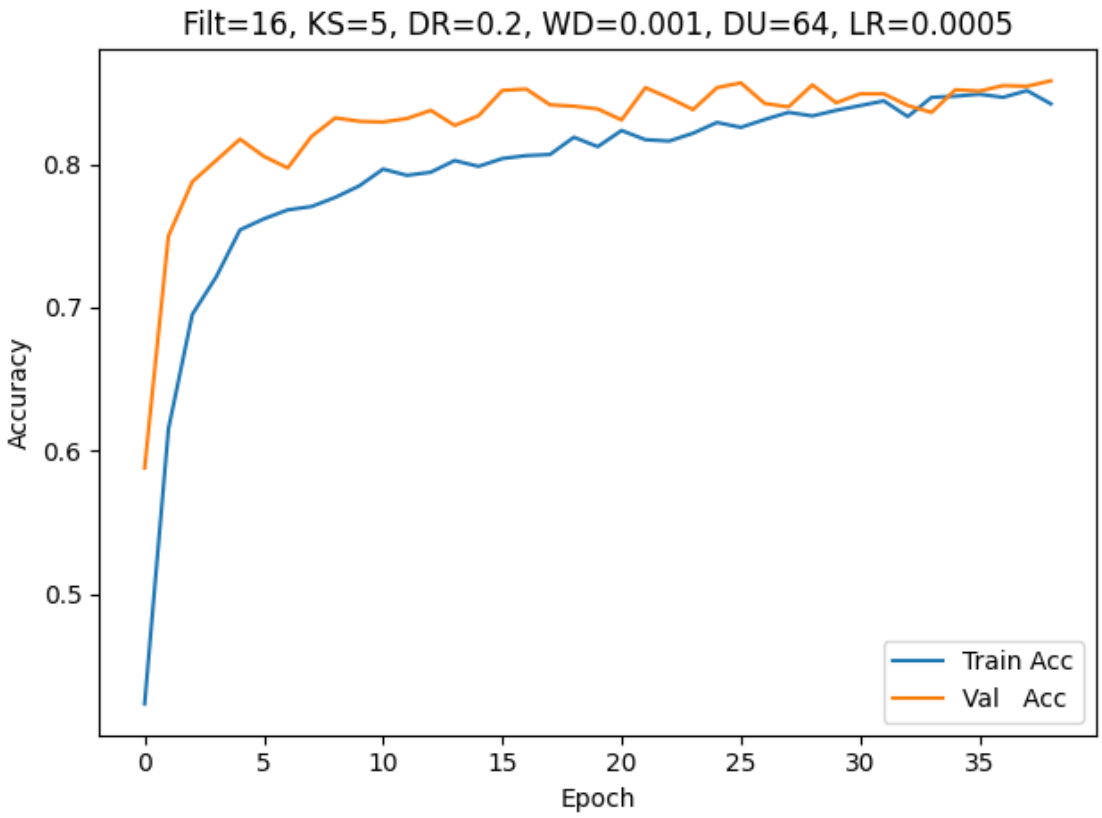

```
--> Best val_acc: 0.8577 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}
```



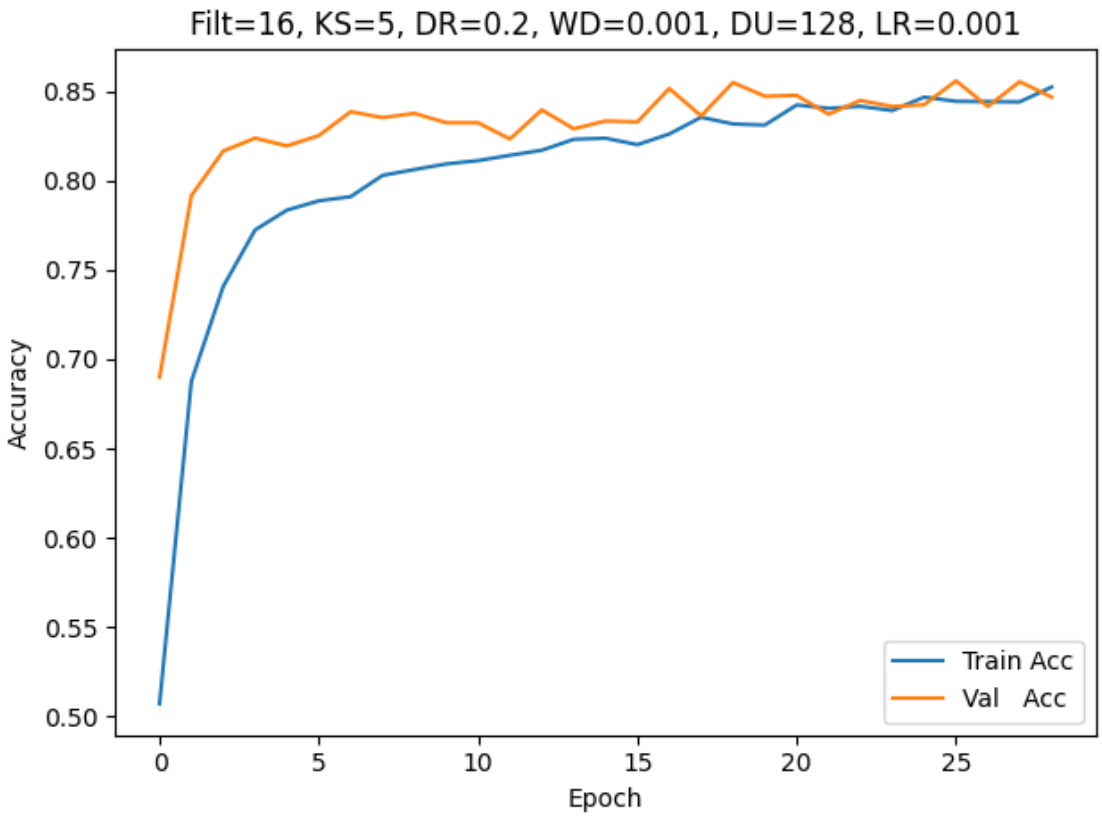

```
--> Best val_acc: 0.8558 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}
```



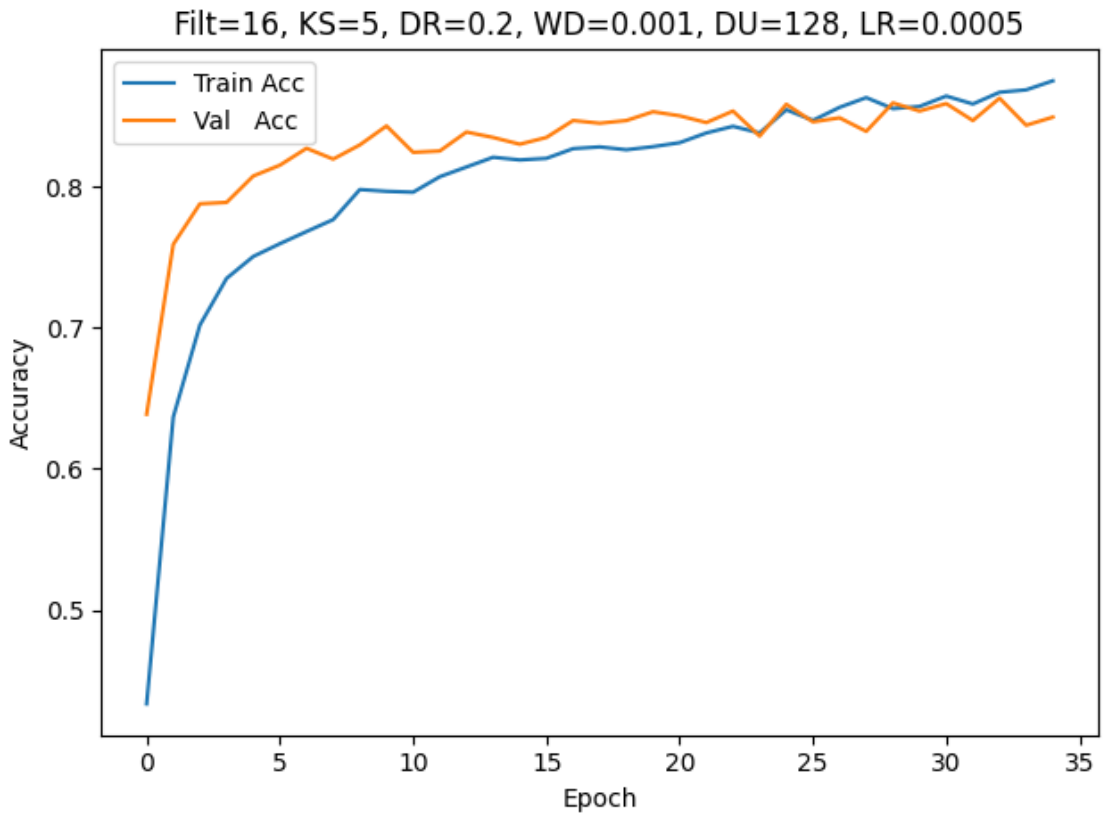
--> Best val_acc: 0.8591 for config: {'filters': 16, 'kernel_size': 5, 'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}



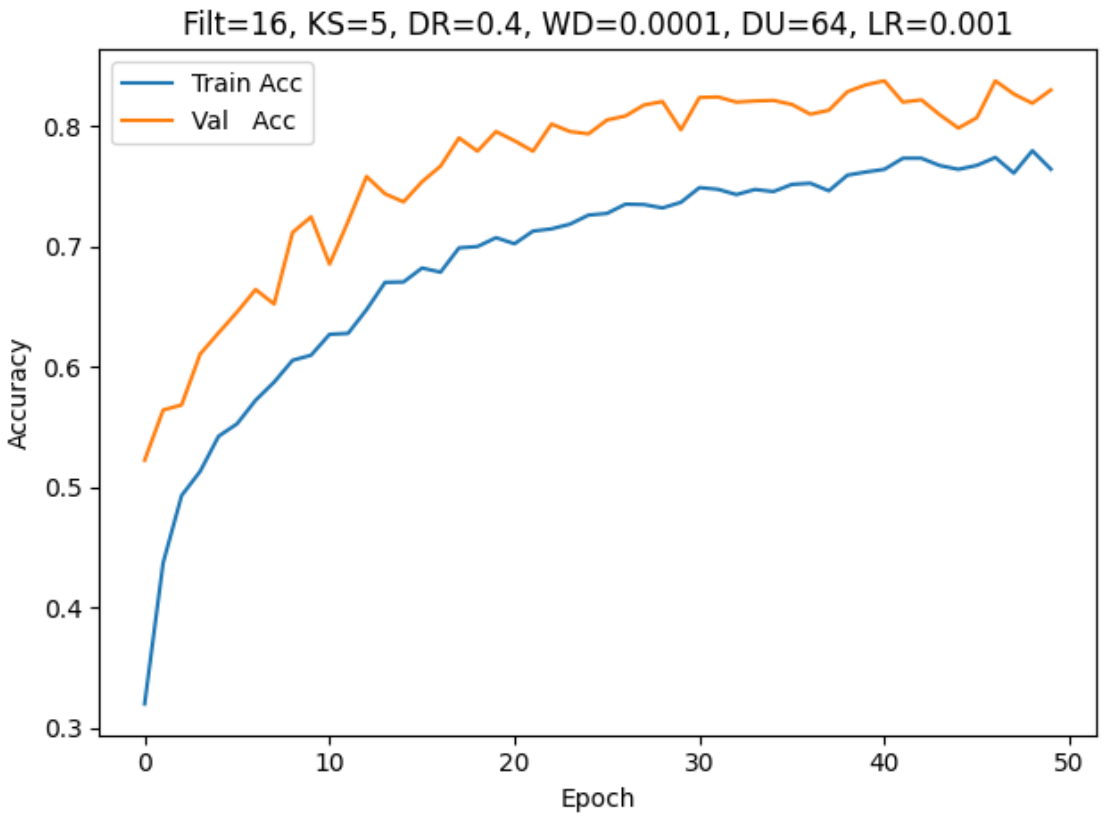

```
--> Best val_acc: 0.8582 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}
```



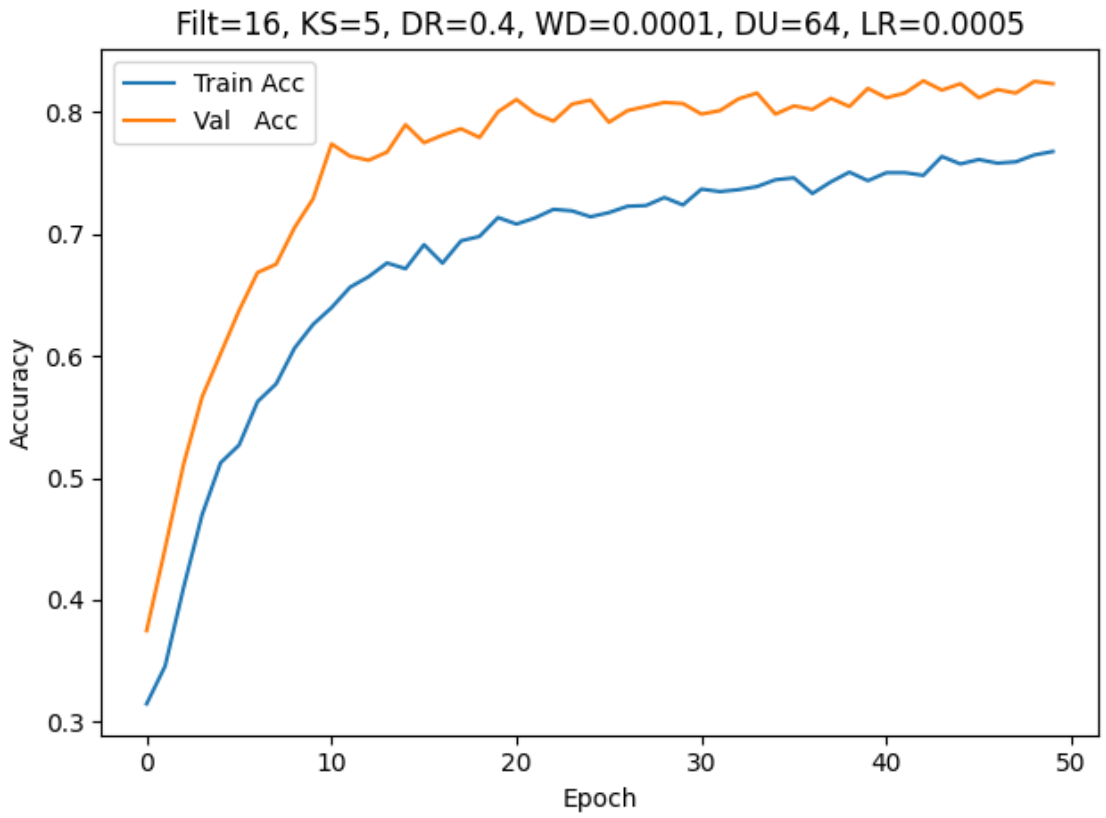
--> Best val_acc: 0.8558 for config: {'filters': 16, 'kernel_size': 5, 'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}



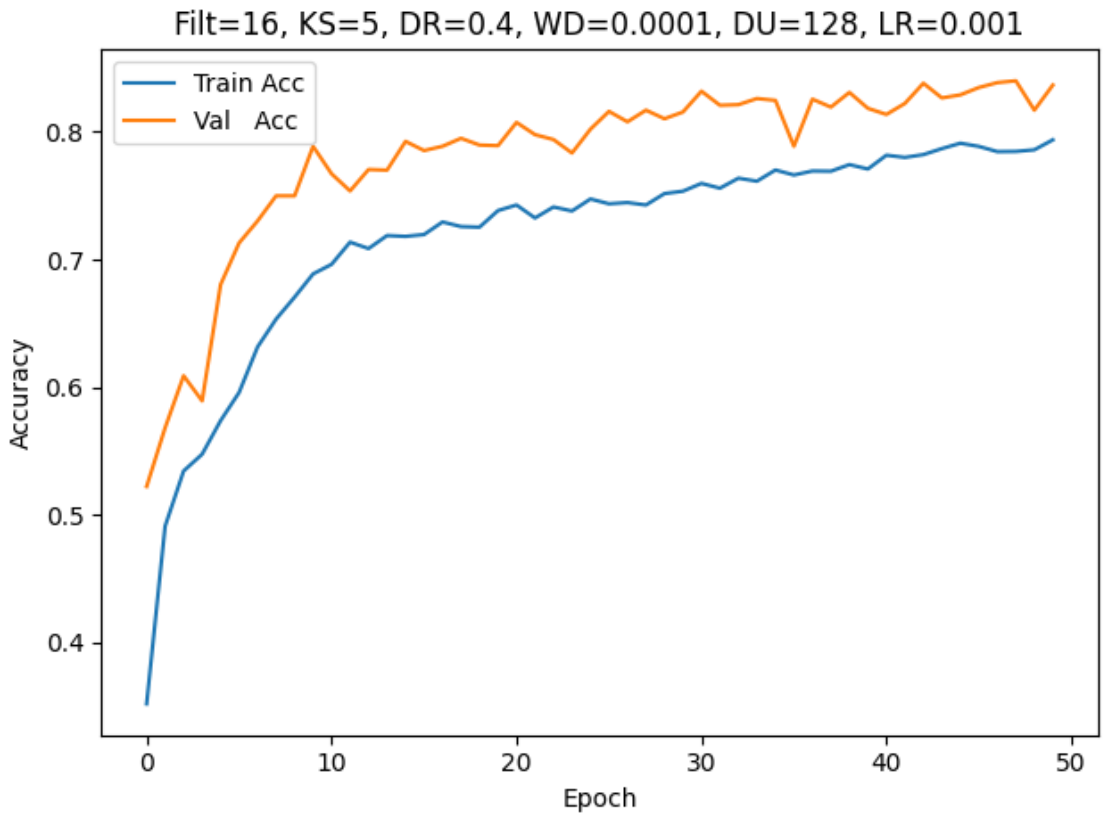

```
--> Best val_acc: 0.8625 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}
```



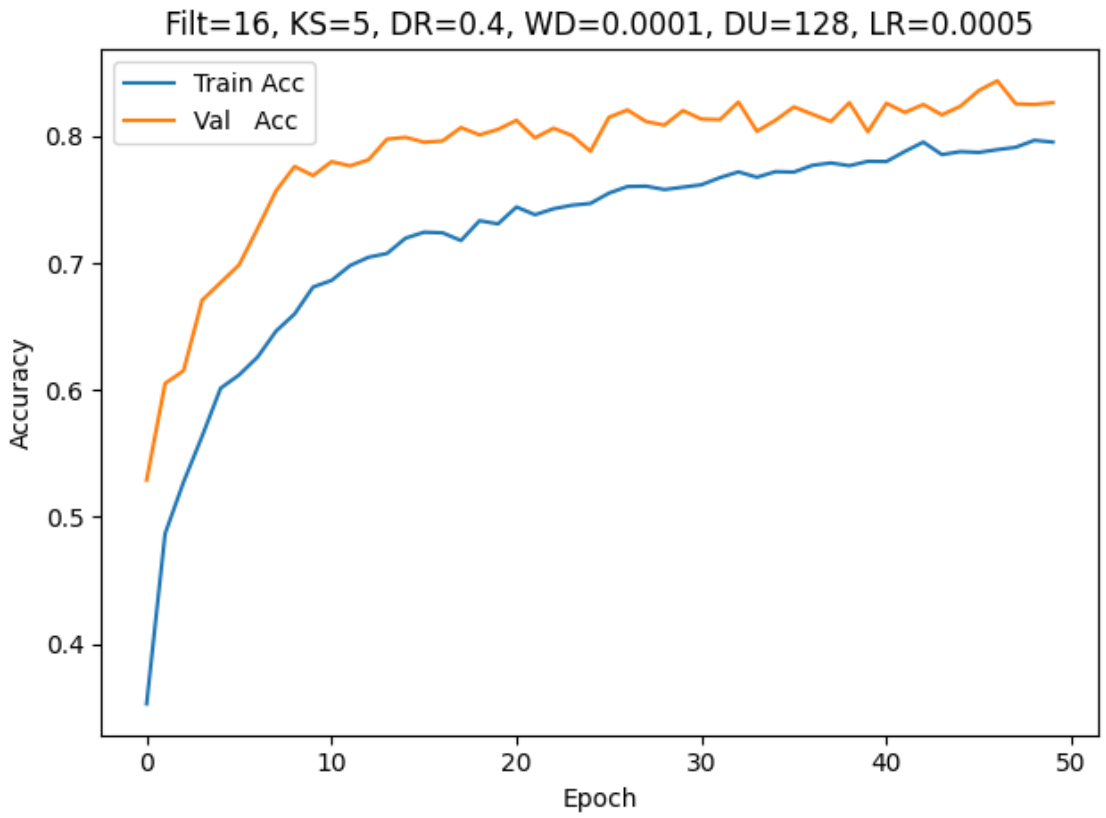

```
--> Best val_acc: 0.8380 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



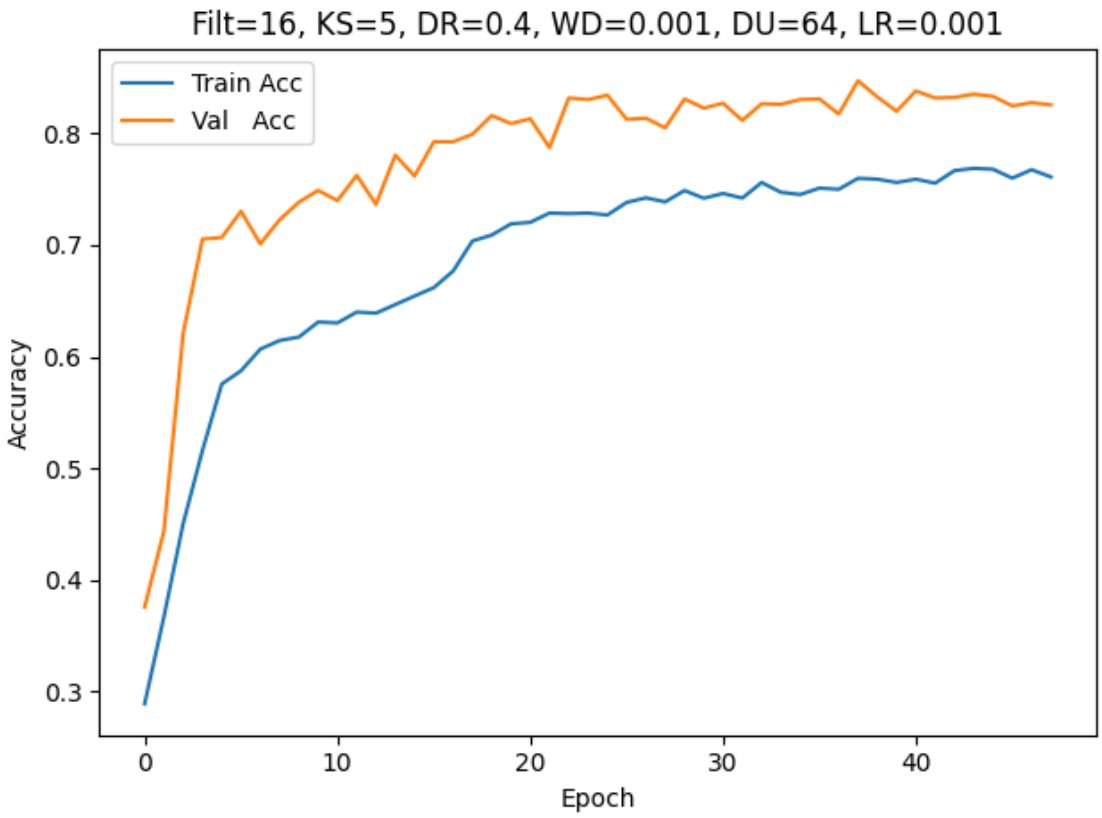
--> Best val_acc: 0.8256 for config: {'filters': 16, 'kernel_size': 5, 'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}



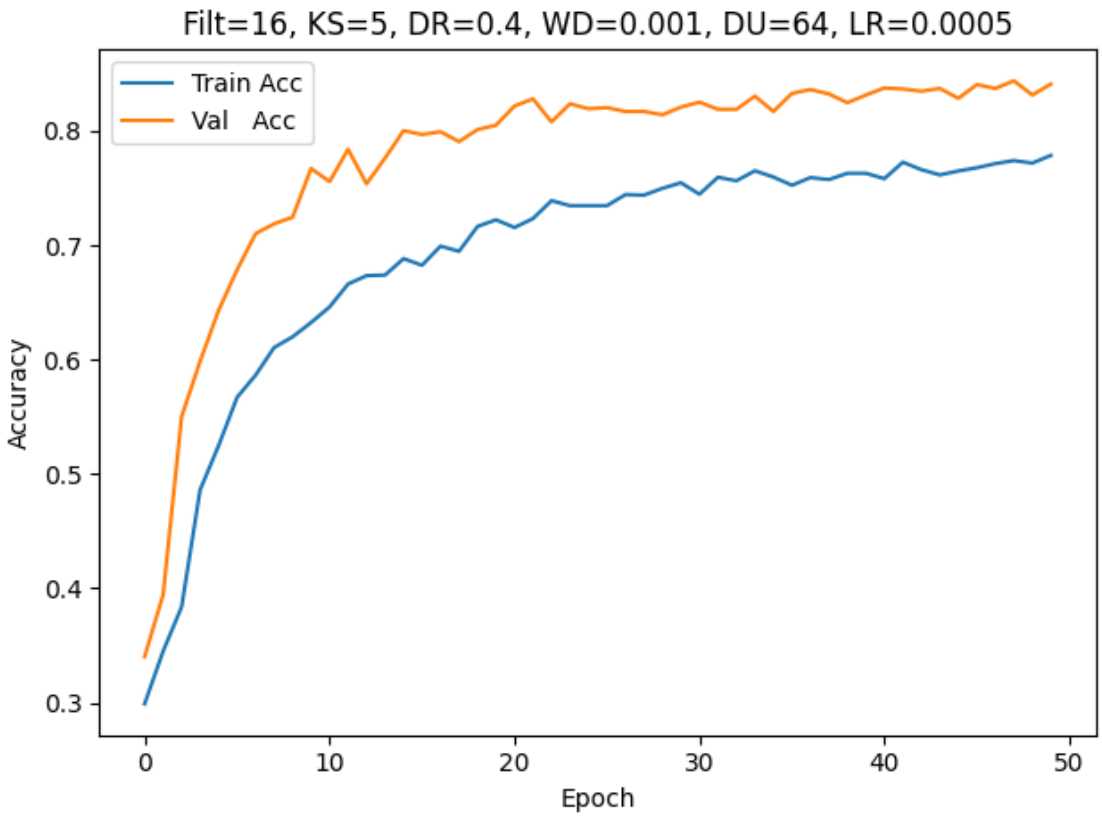

```
--> Best val_acc: 0.8400 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}
```



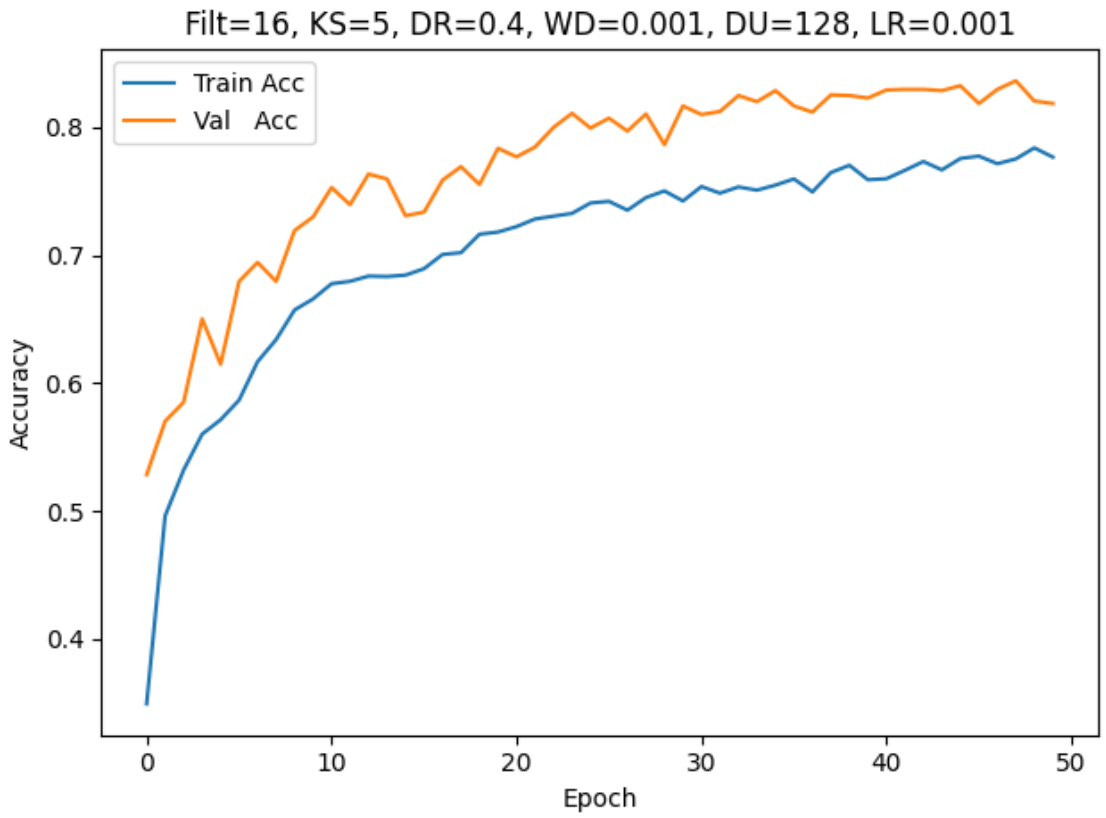

```
--> Best val_acc: 0.8438 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}
```



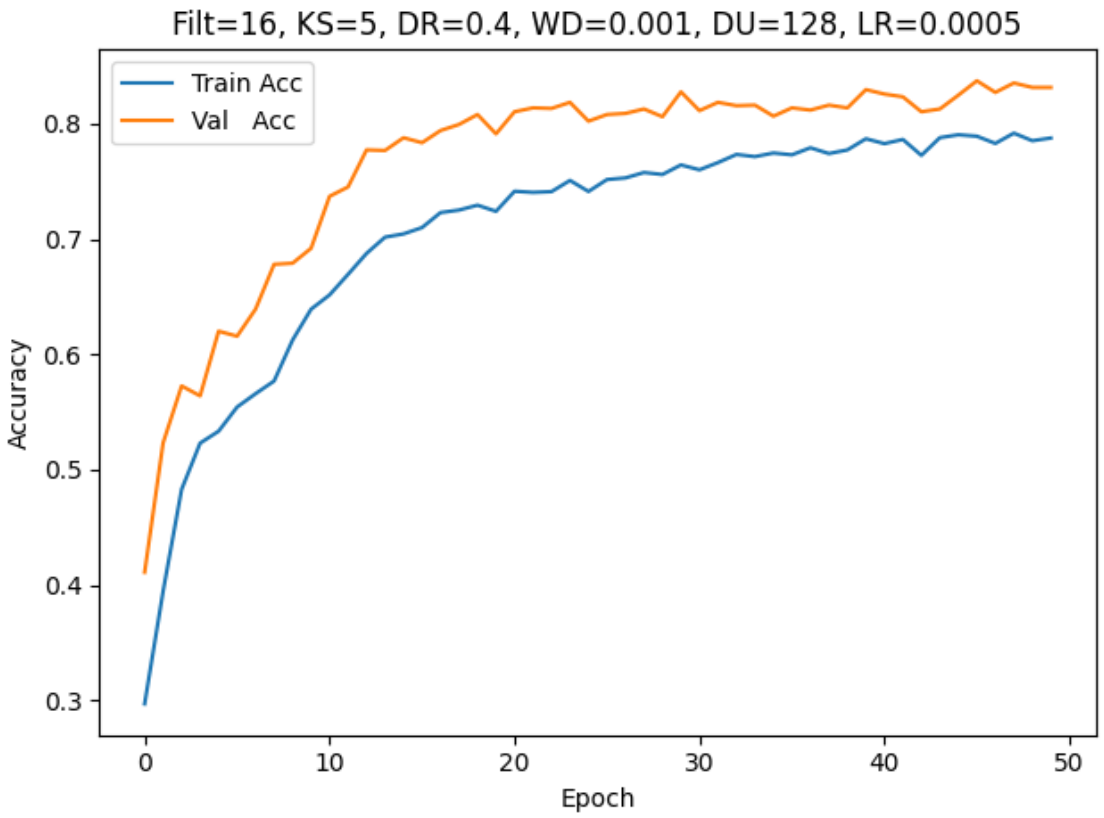

```
--> Best val_acc: 0.8471 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}
```



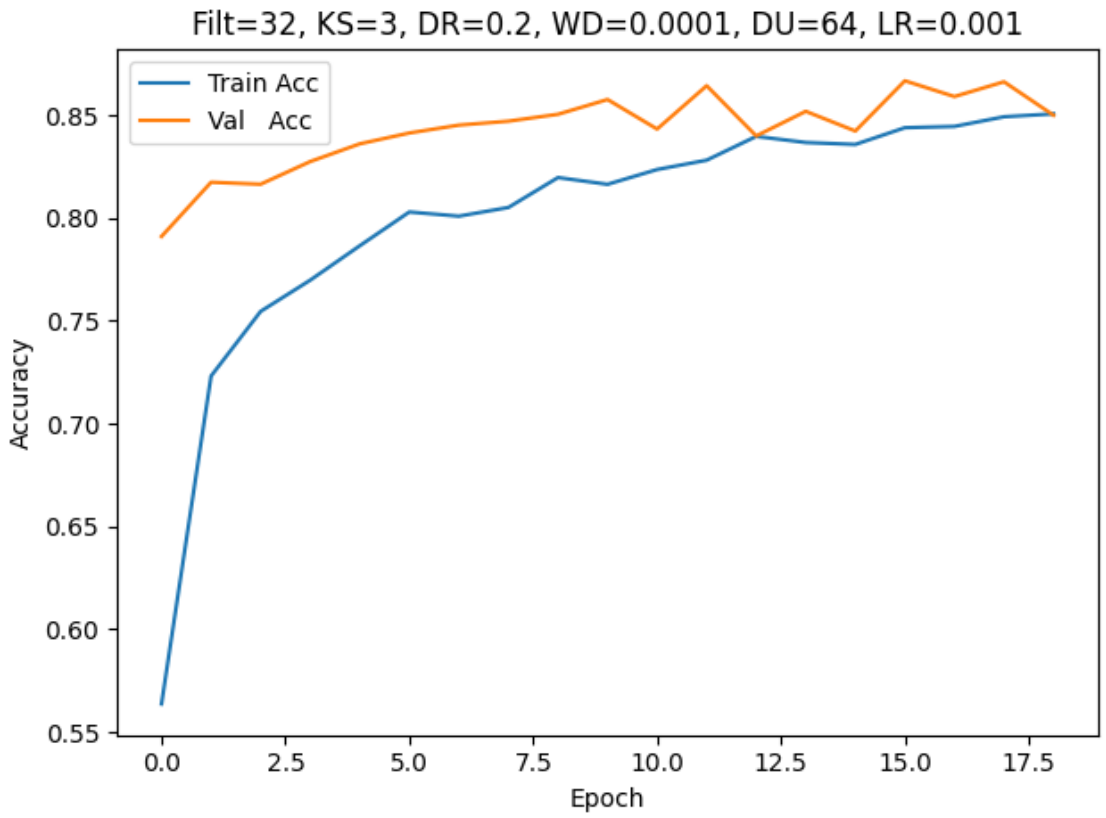

```
--> Best val_acc: 0.8438 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}
```



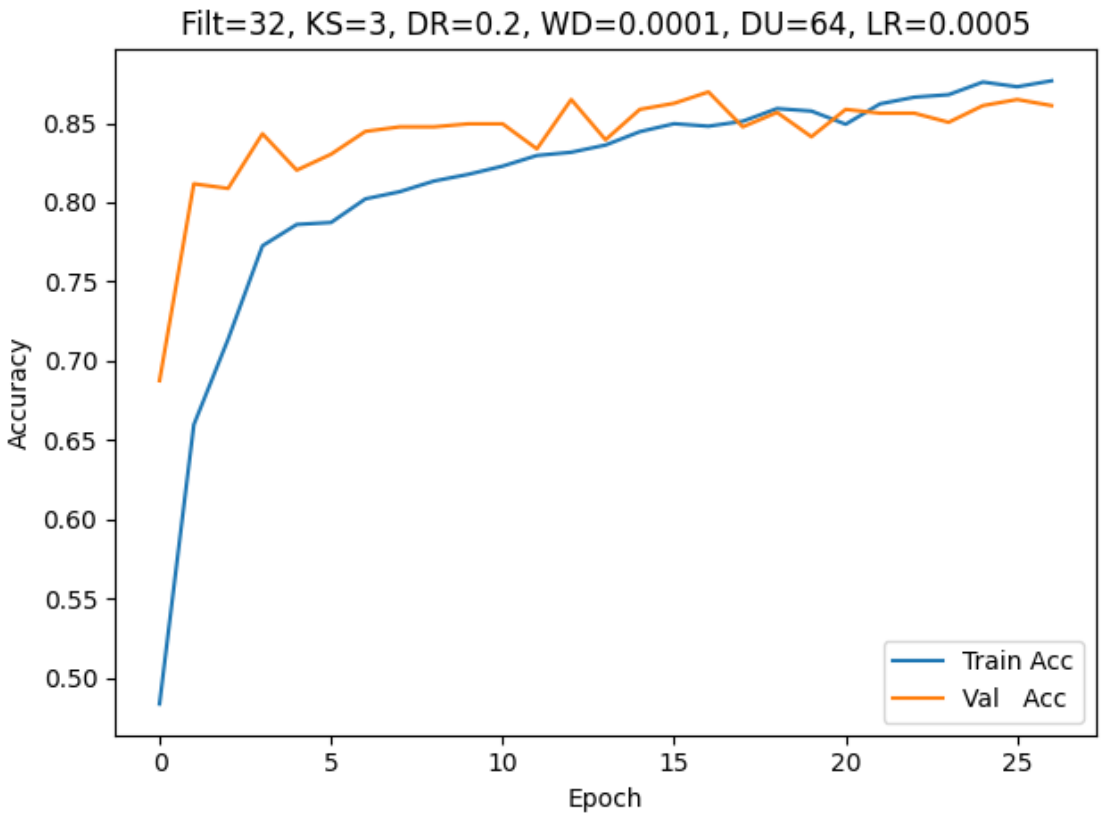
--> Best val_acc: 0.8361 for config: {'filters': 16, 'kernel_size': 5, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}



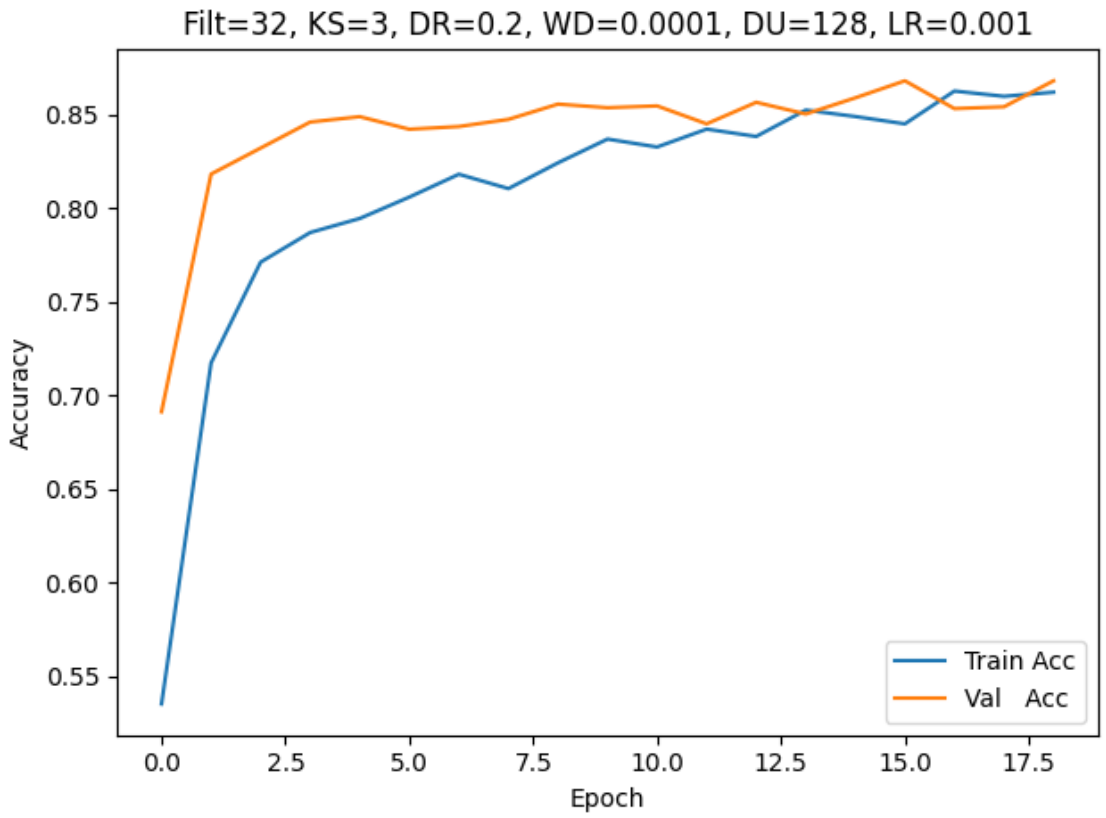

```
--> Best val_acc: 0.8371 for config: {'filters': 16, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}
```



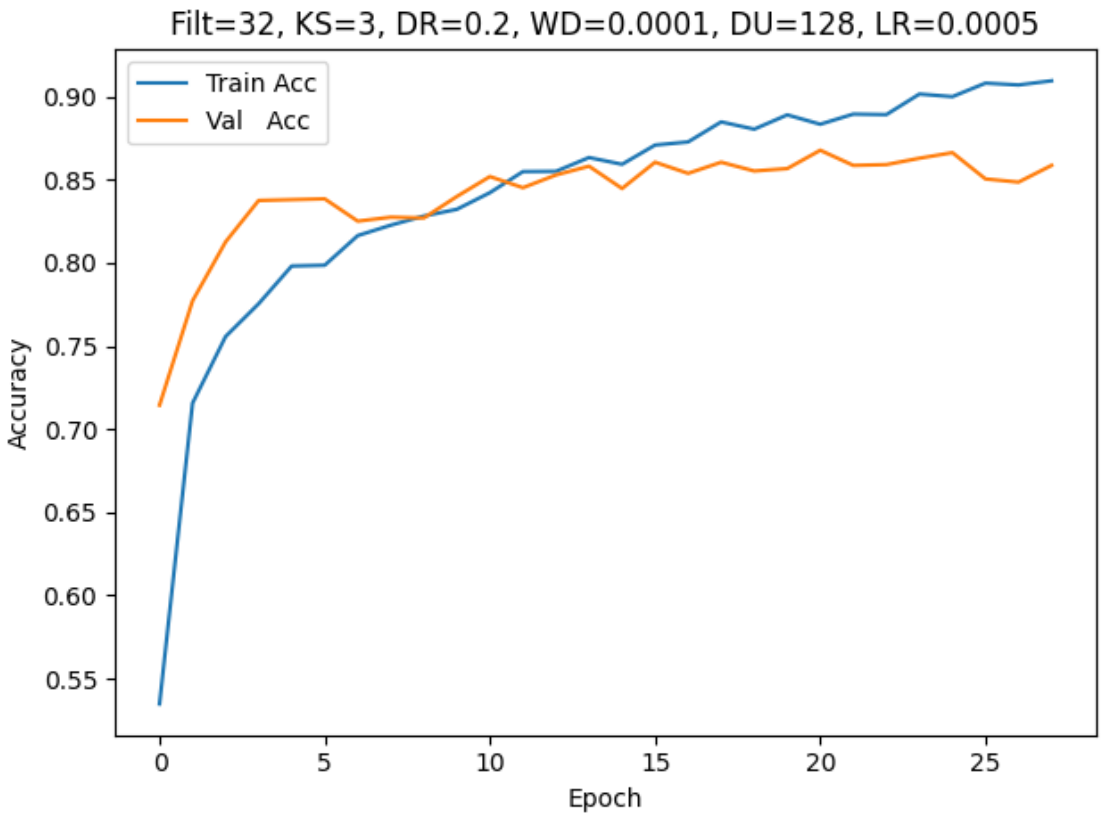

```
--> Best val_acc: 0.8668 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



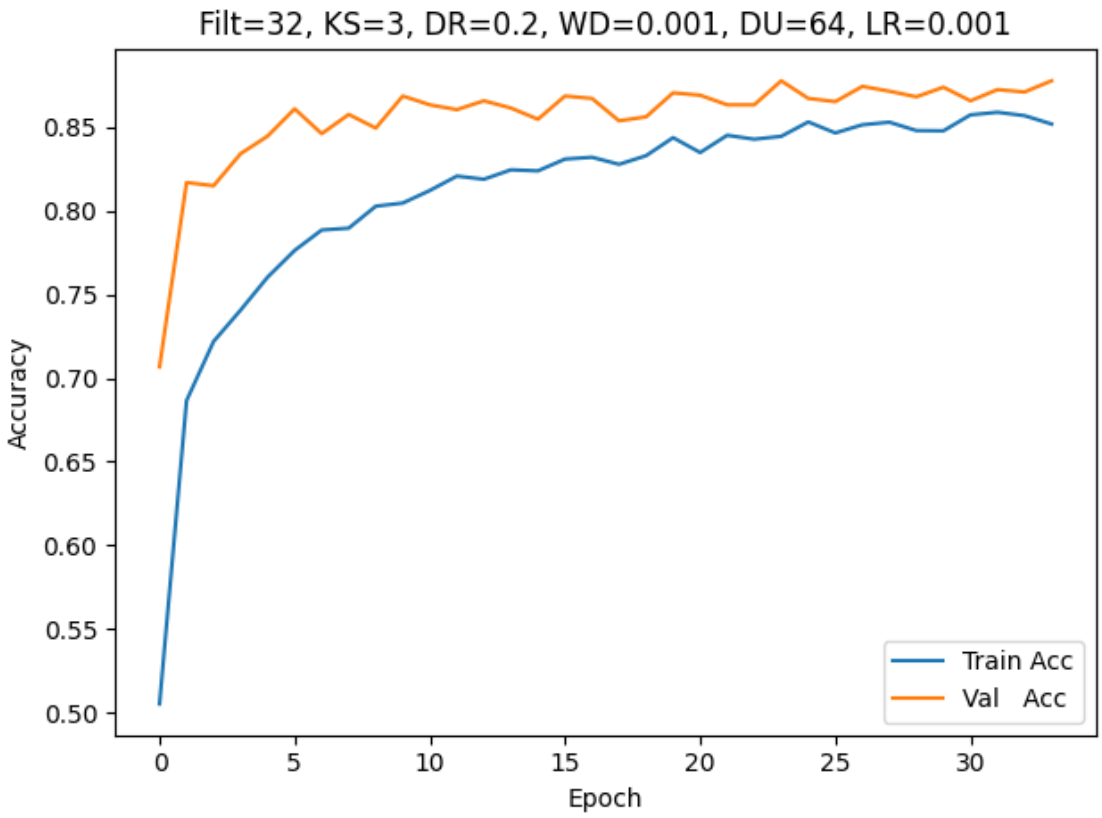
--> Best val_acc: 0.8697 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}



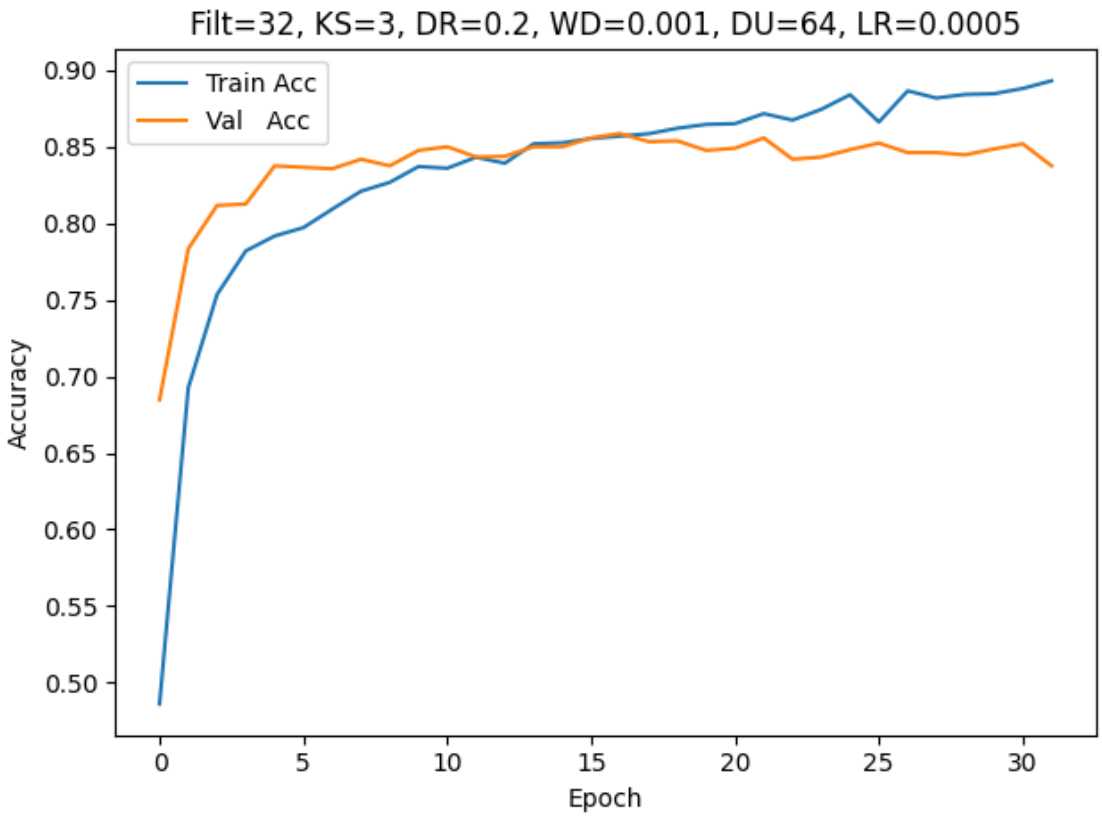

```
--> Best val_acc: 0.8682 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}
```



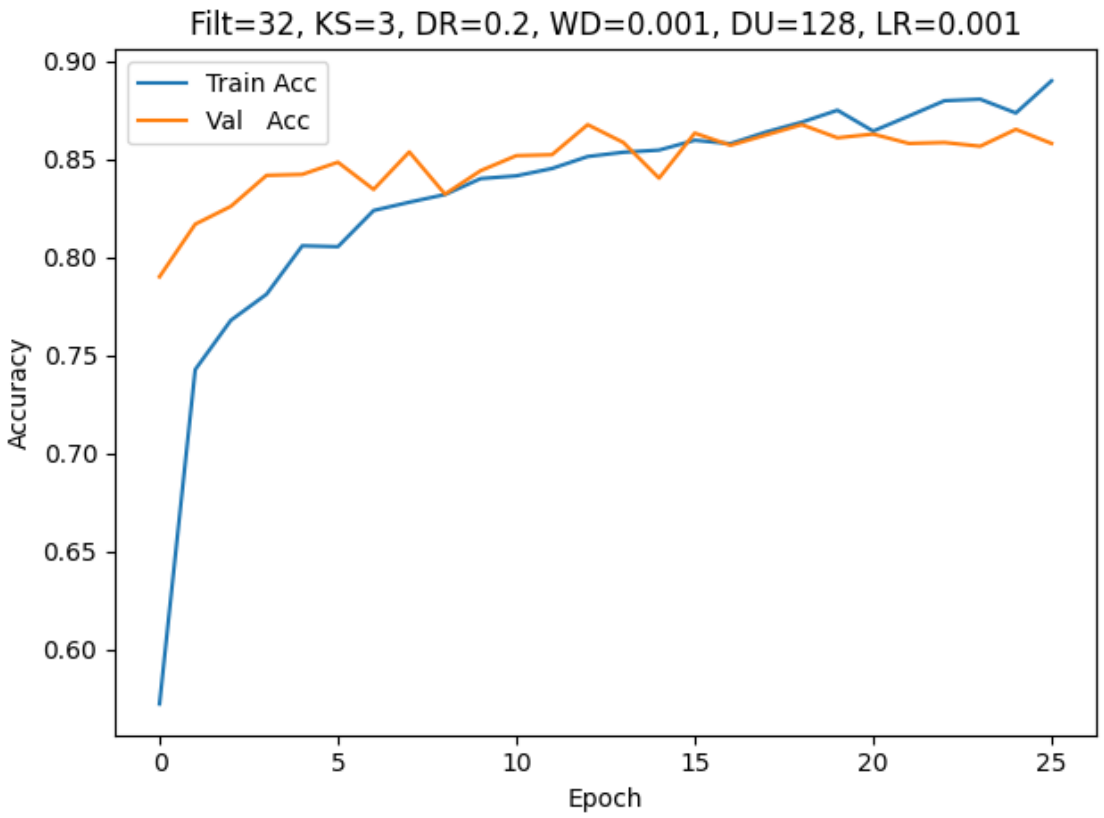
--> Best val_acc: 0.8678 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}



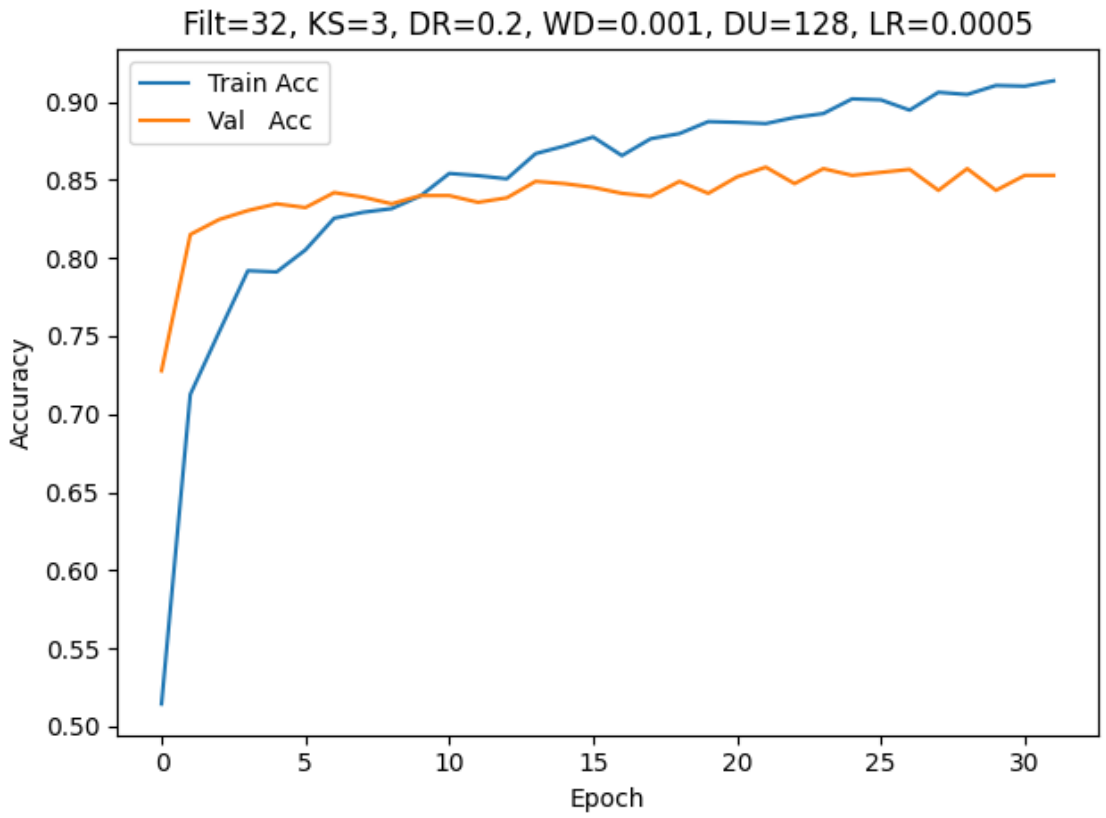

```
--> Best val_acc: 0.8778 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}
```



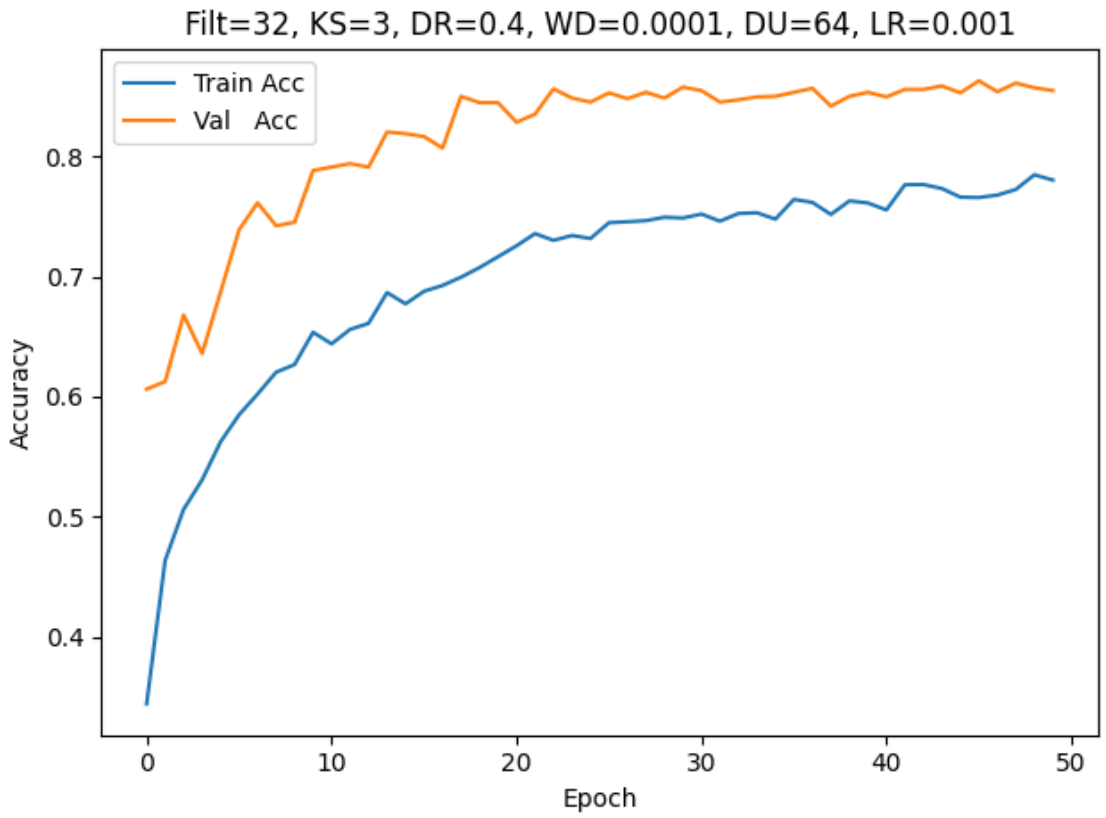

```
--> Best val_acc: 0.8586 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}
```



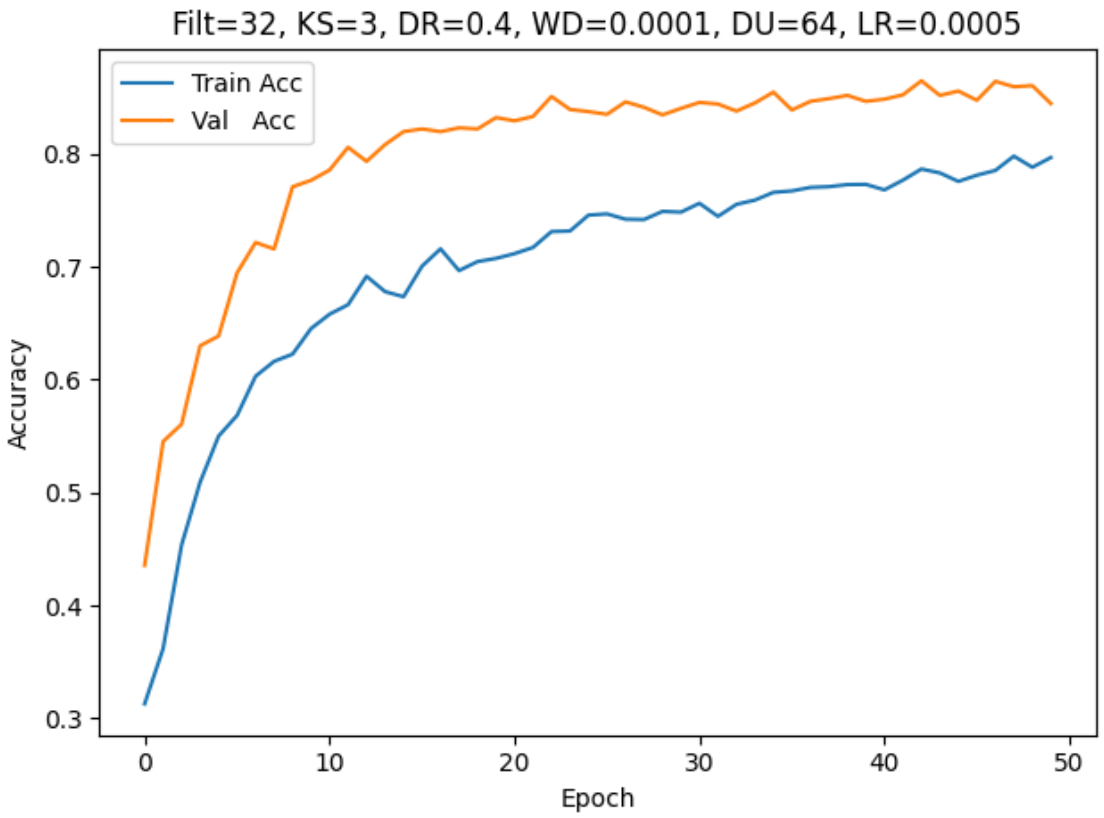

```
--> Best val_acc: 0.8678 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}
```



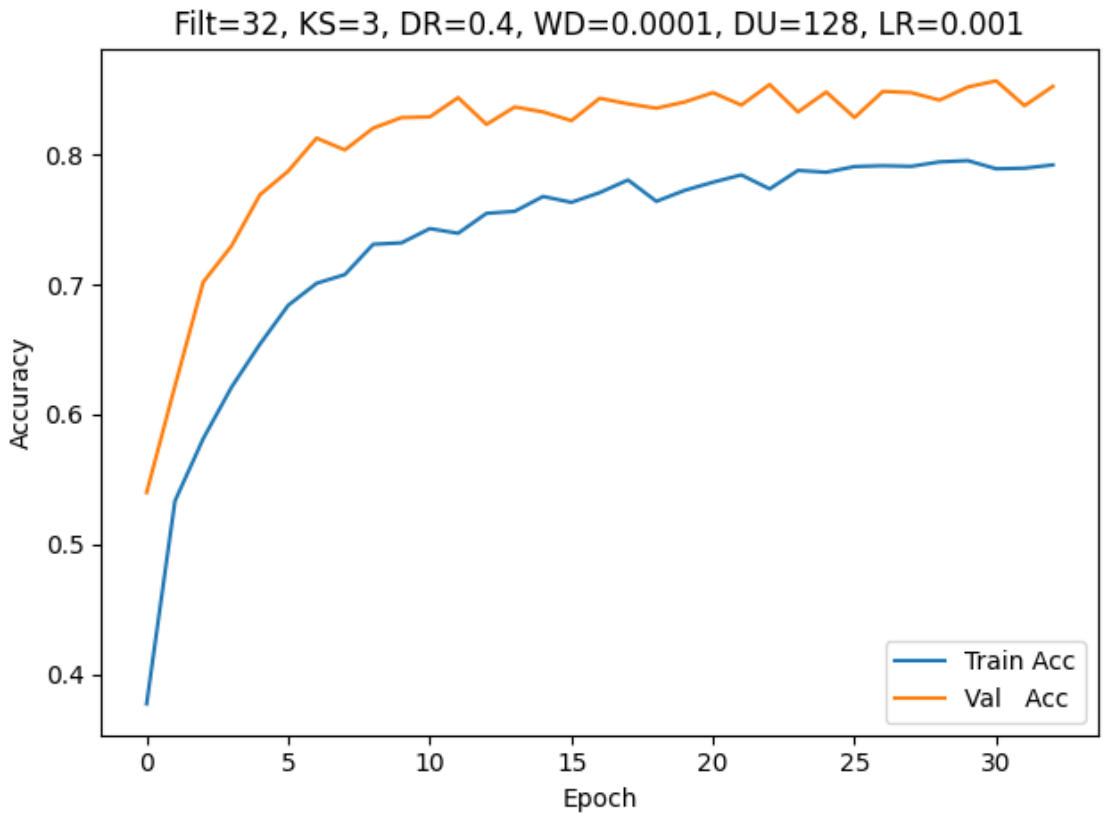

```
--> Best val_acc: 0.8582 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}
```



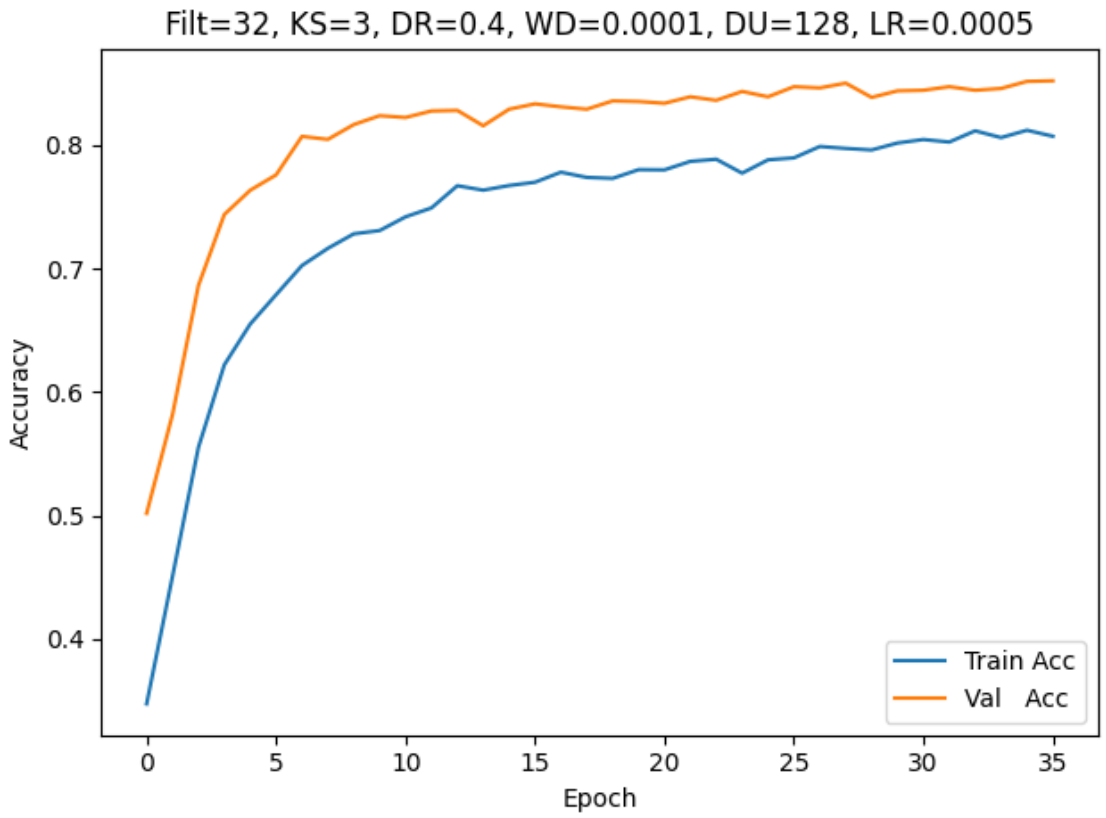

```
--> Best val_acc: 0.8630 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



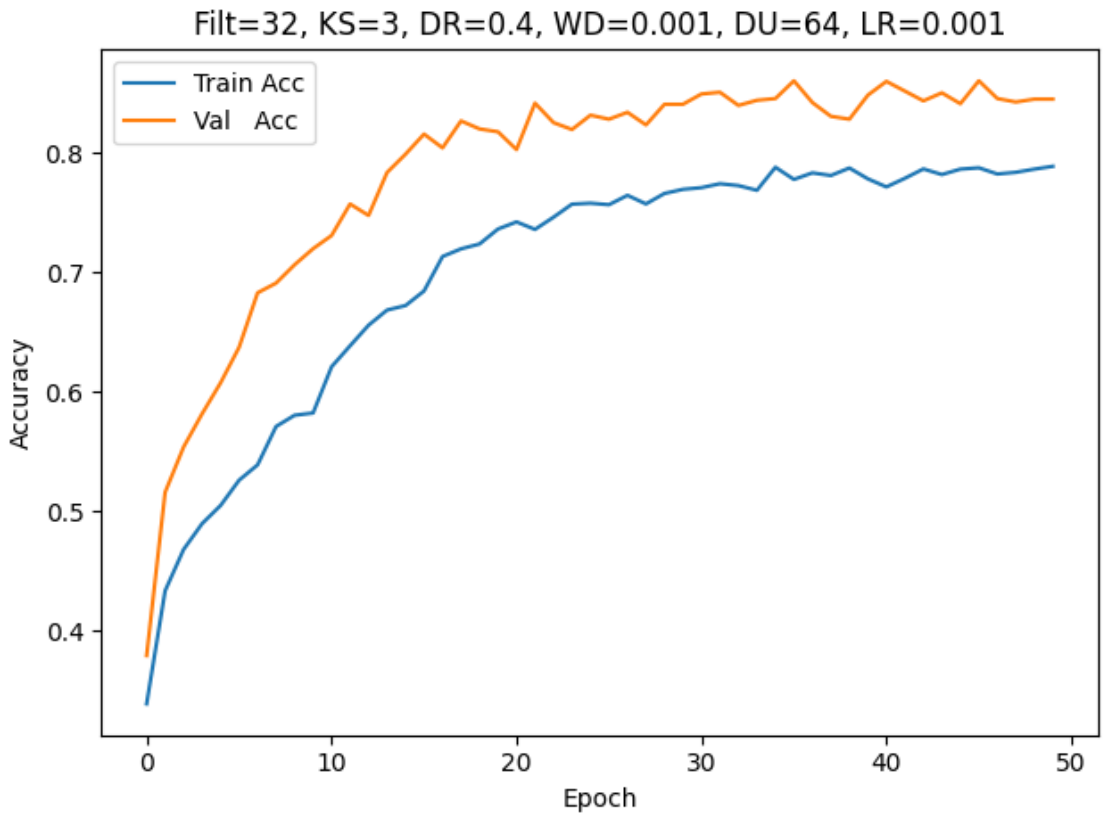
--> Best val_acc: 0.8649 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}



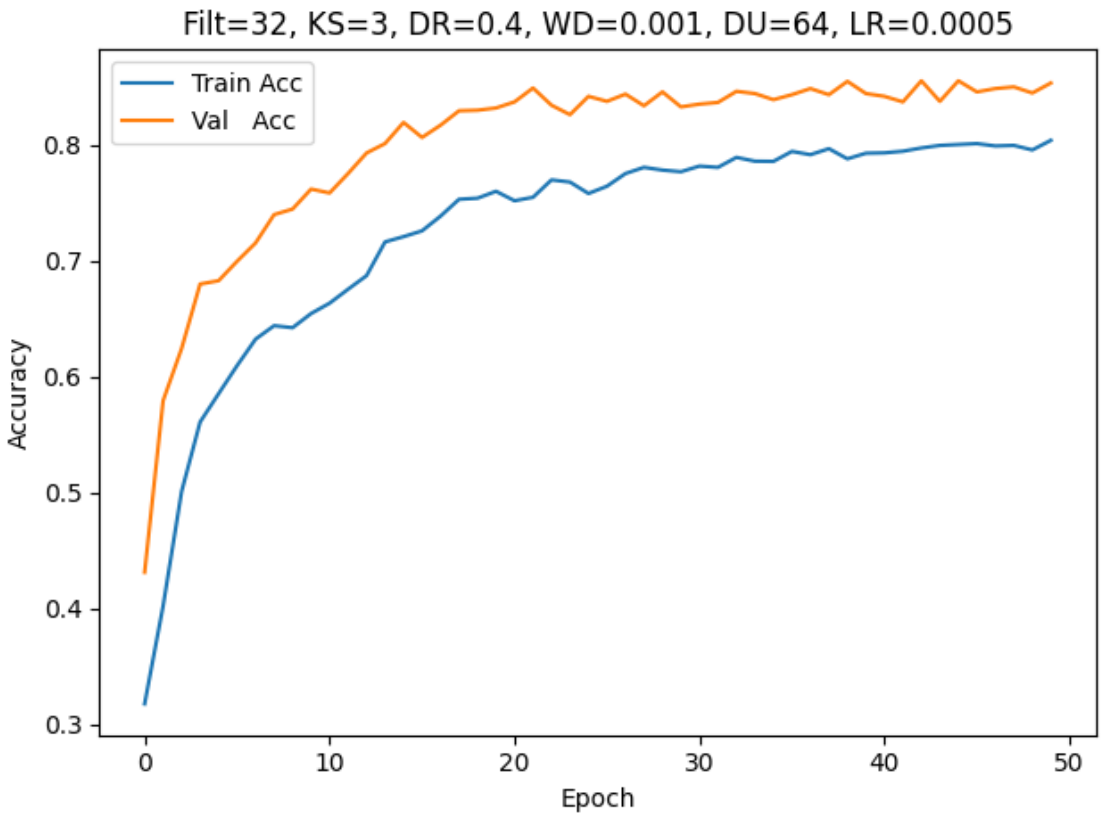
--> Best val_acc: 0.8567 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}



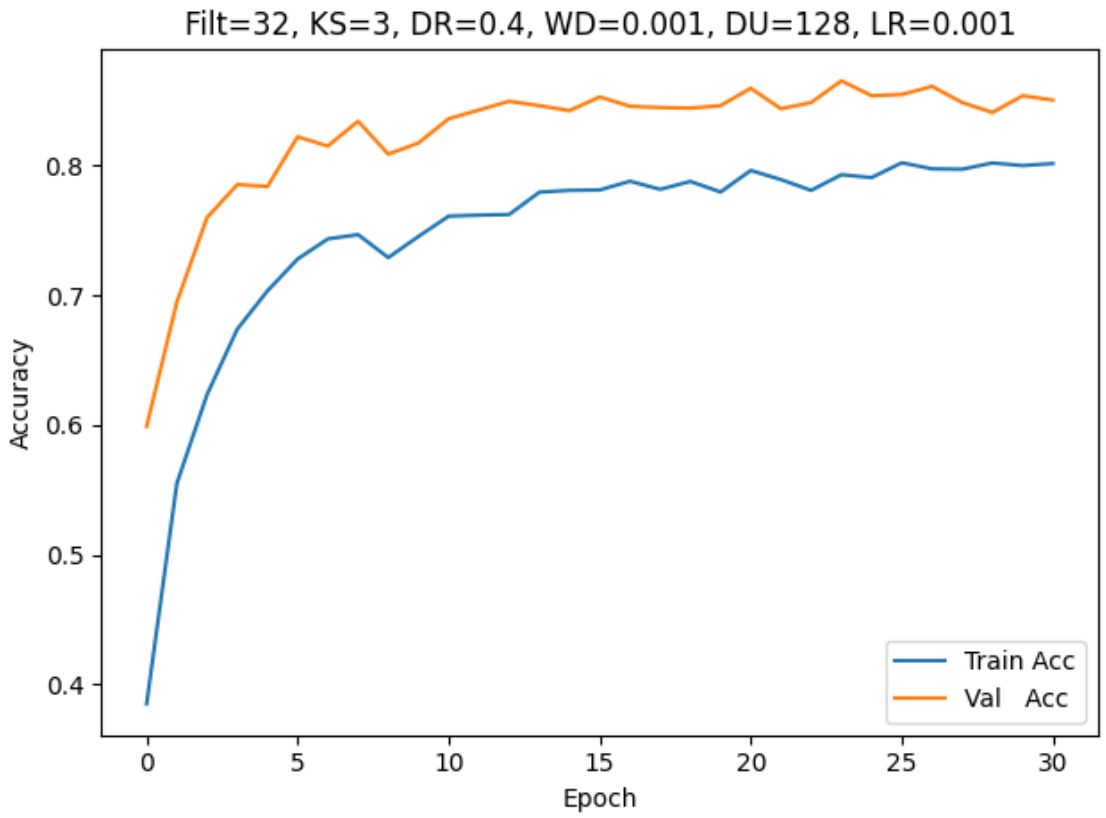
--> Best val_acc: 0.8519 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}



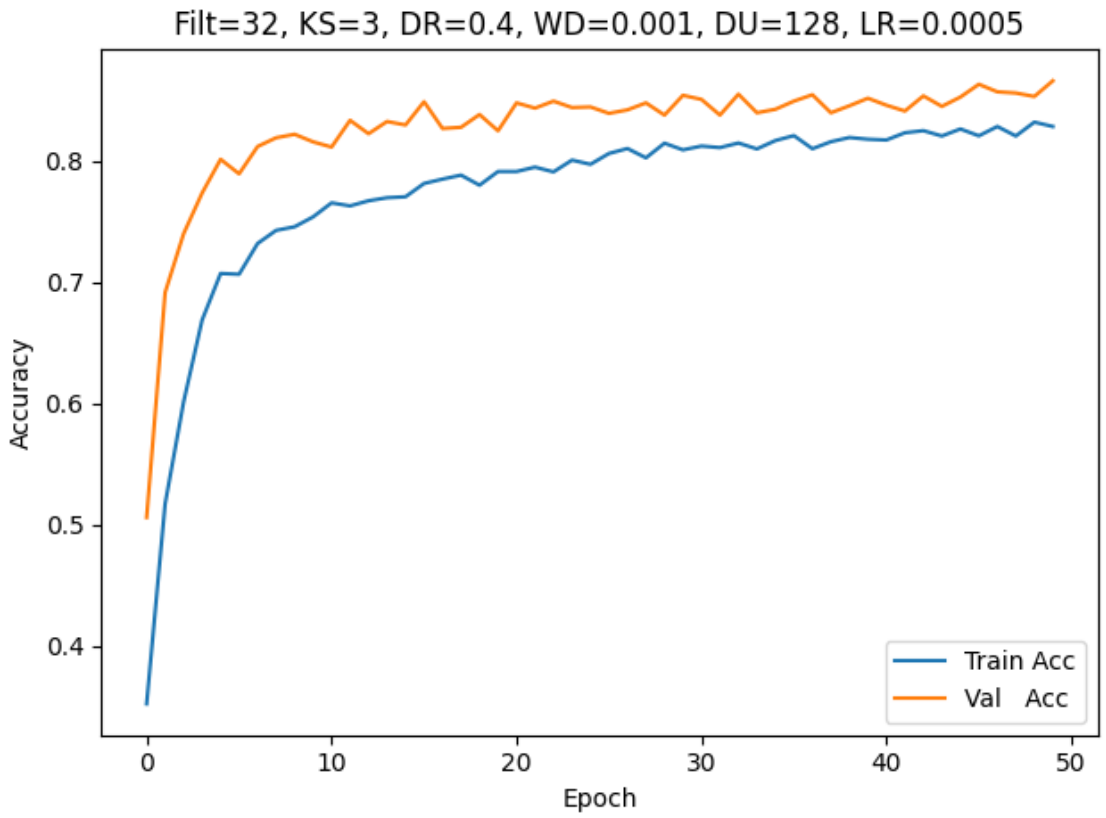

```
--> Best val_acc: 0.8601 for config: {'filters': 32, 'kernel_size': 3,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}
```



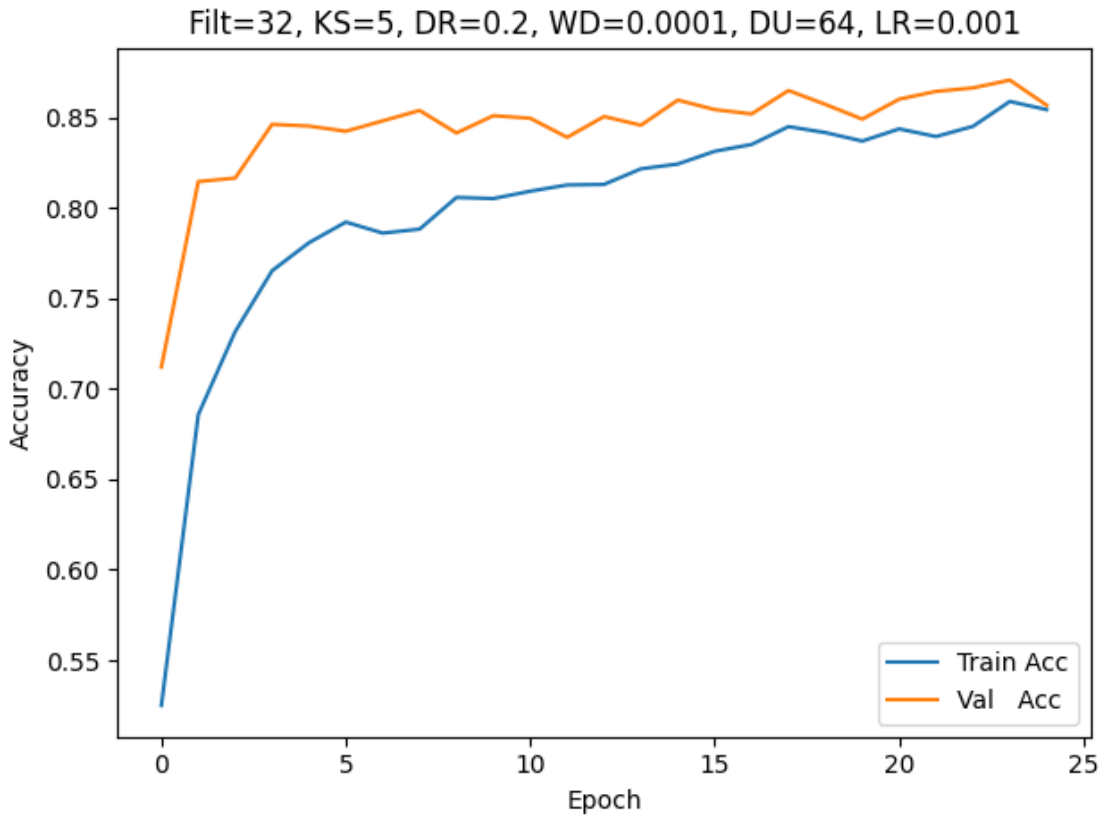
--> Best val_acc: 0.8553 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}



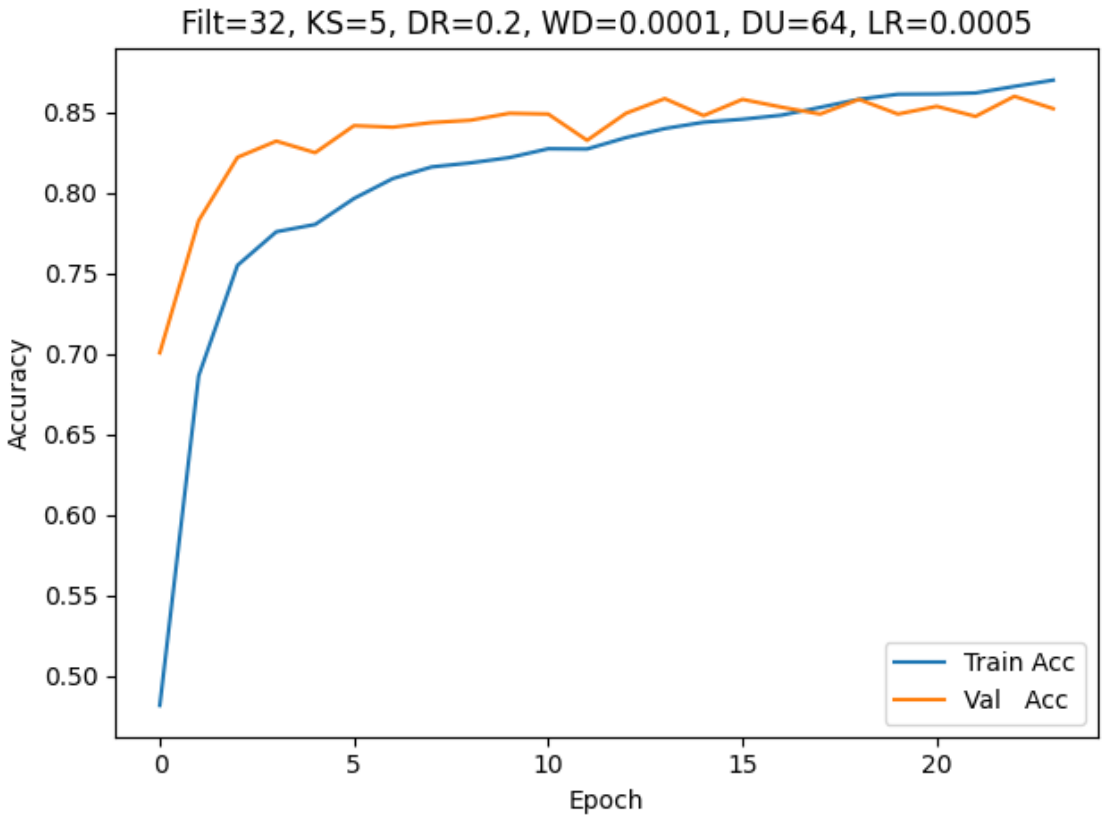
--> Best val_acc: 0.8654 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}



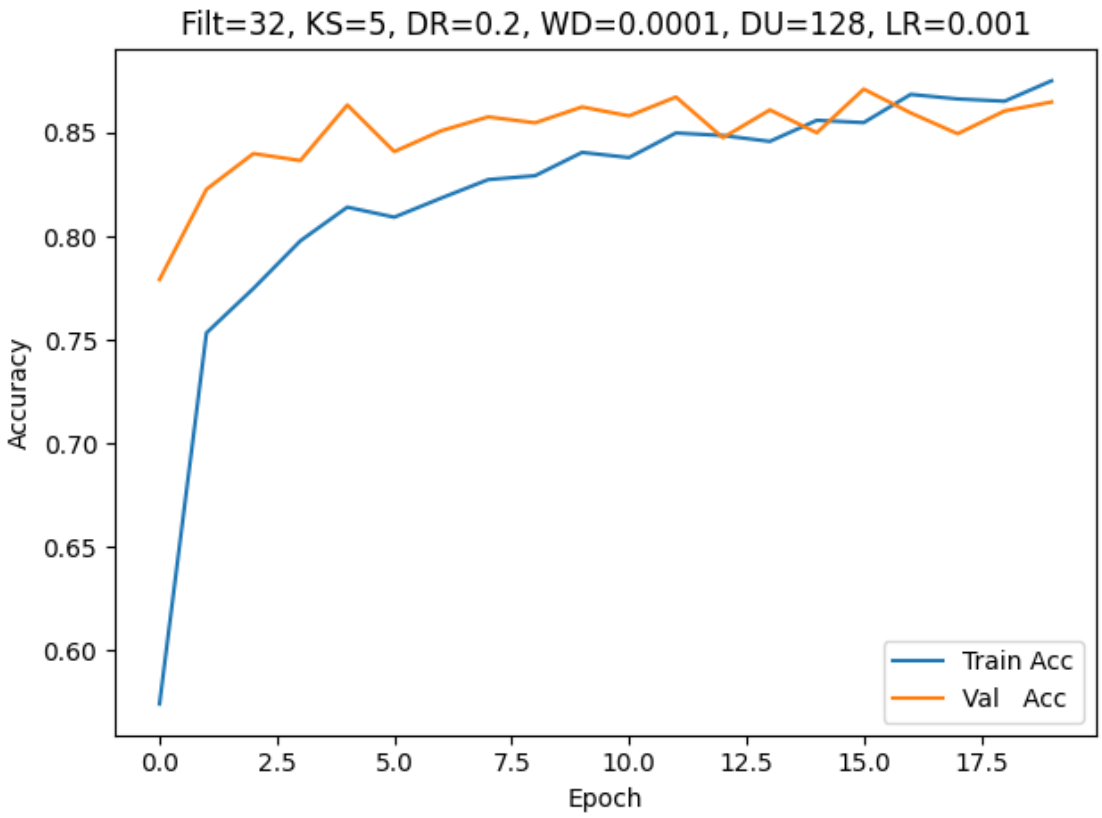
--> Best val_acc: 0.8658 for config: {'filters': 32, 'kernel_size': 3, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}



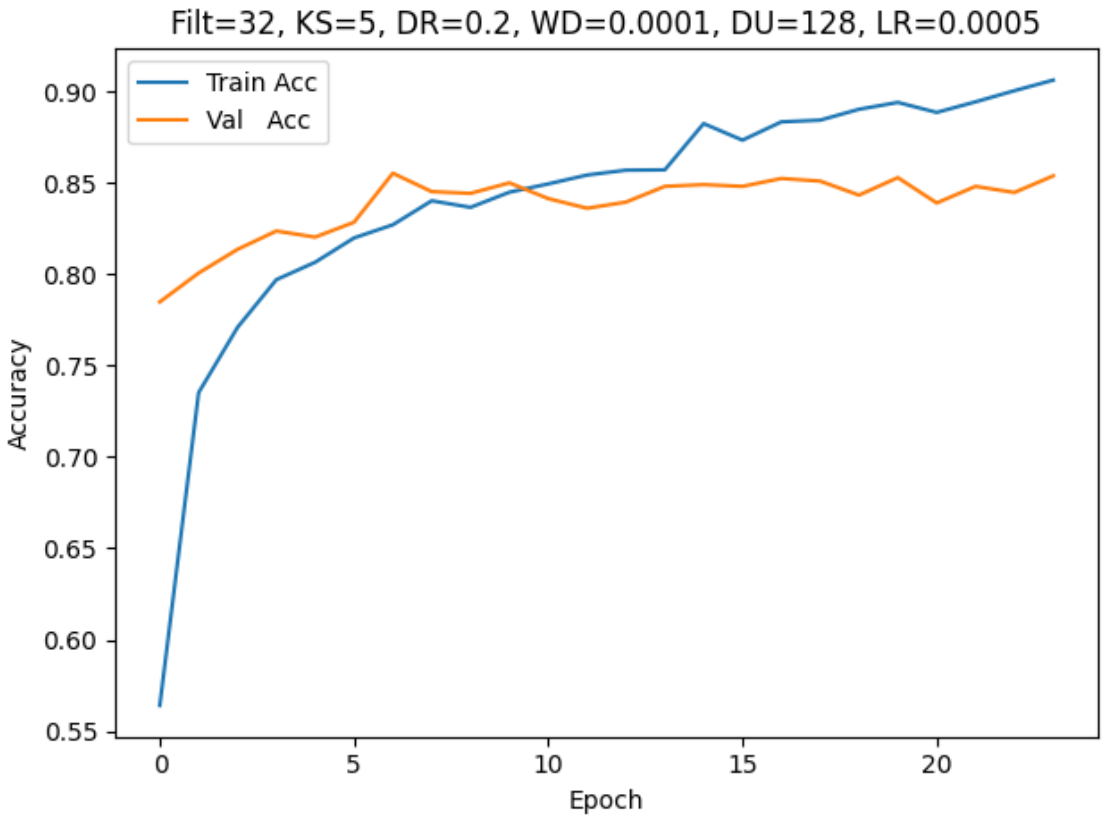

```
--> Best val_acc: 0.8706 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



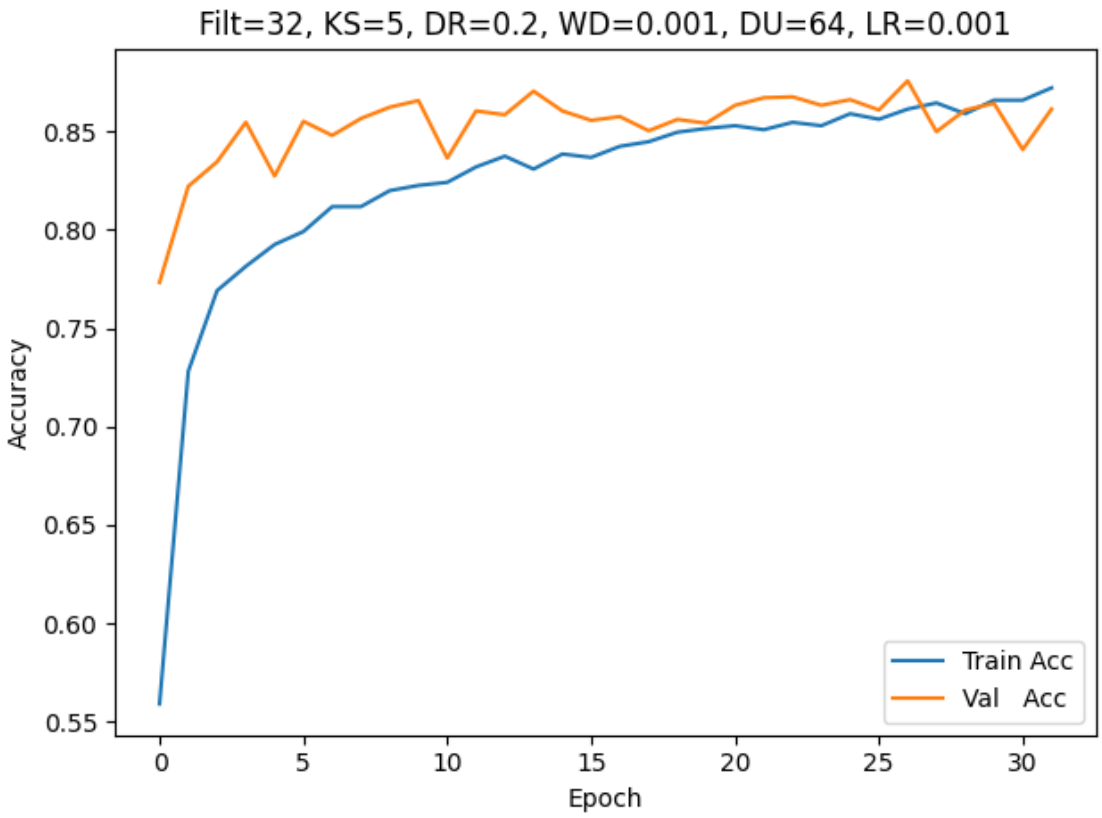
--> Best val_acc: 0.8596 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}



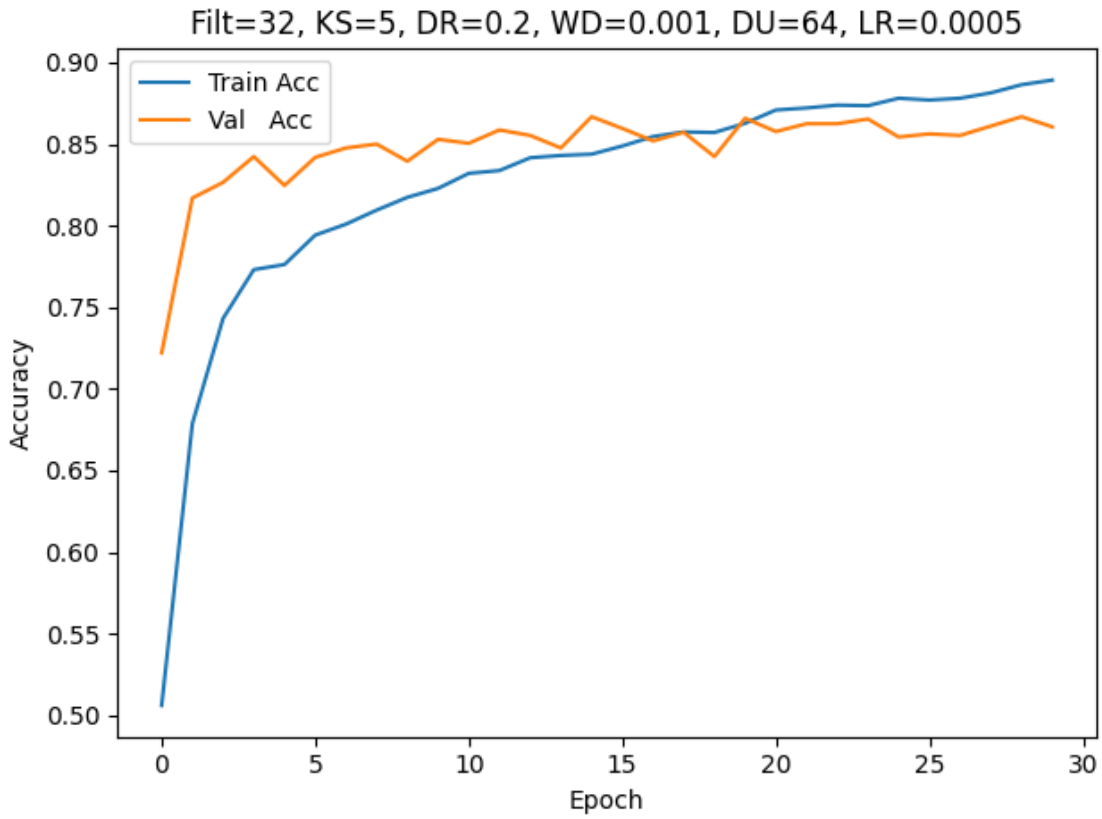
--> Best val_acc: 0.8711 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}



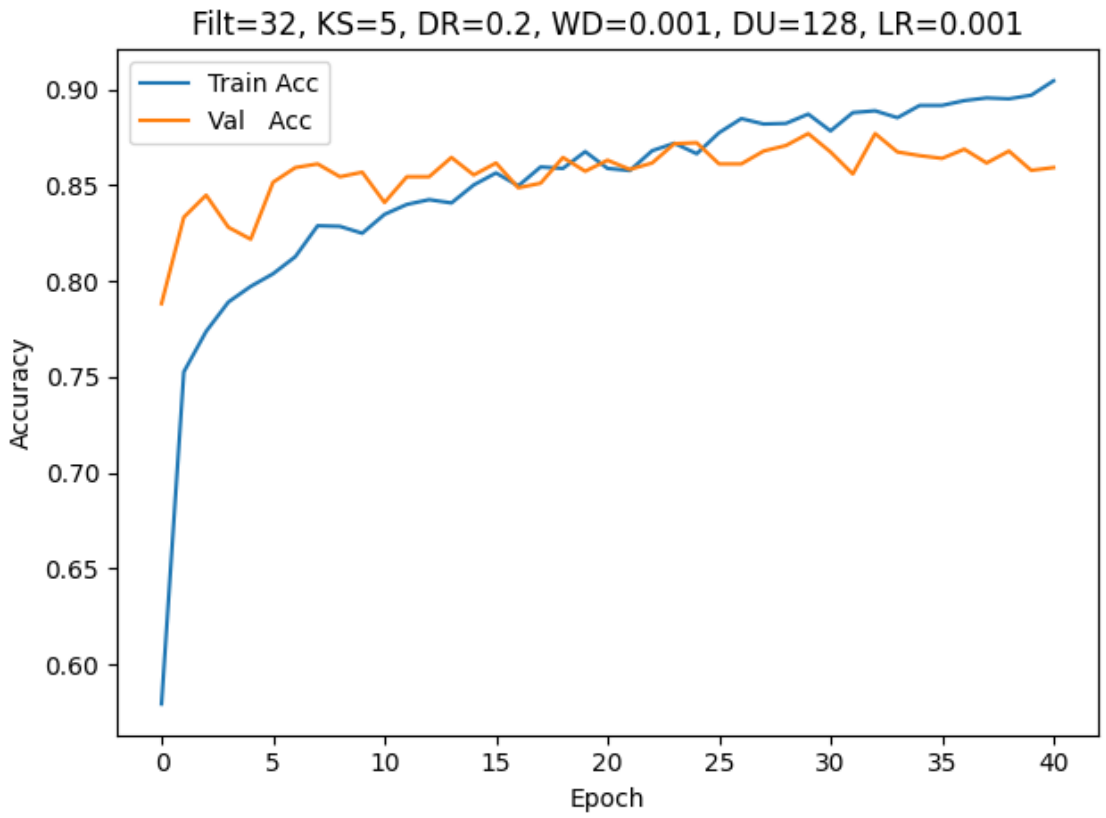
--> Best val_acc: 0.8553 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.2, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}



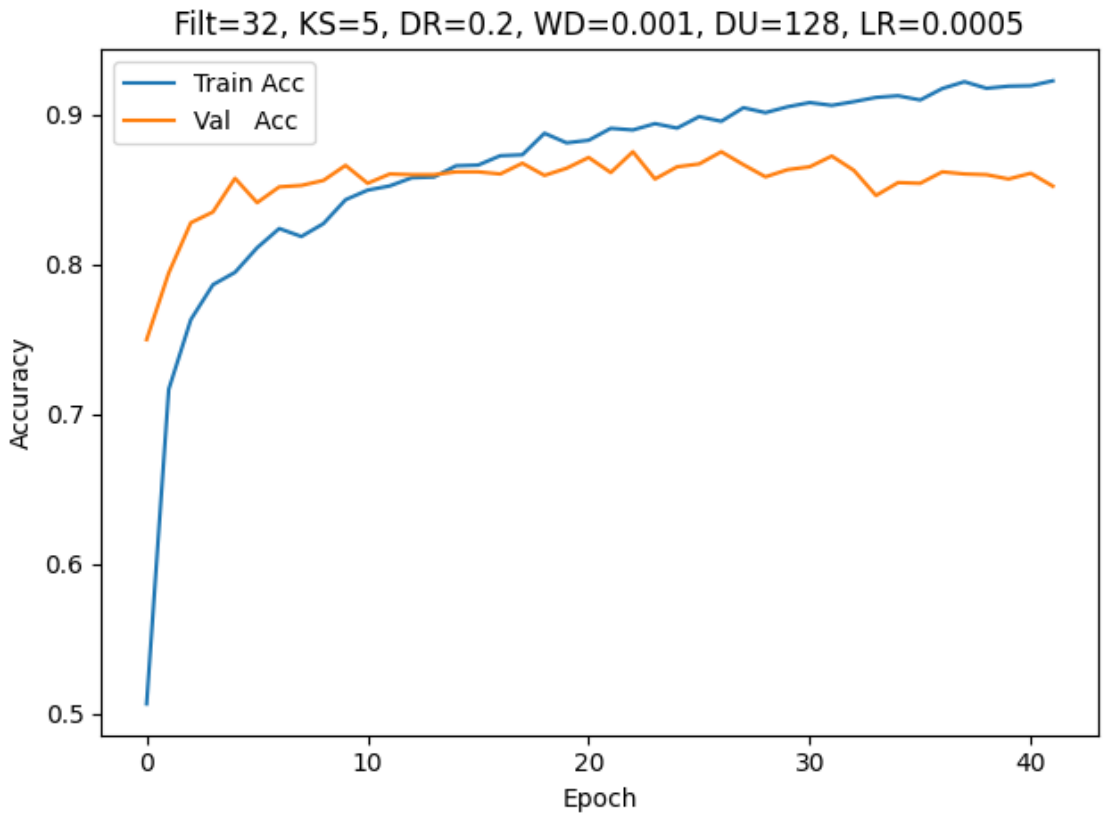

```
--> Best val_acc: 0.8759 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}
```



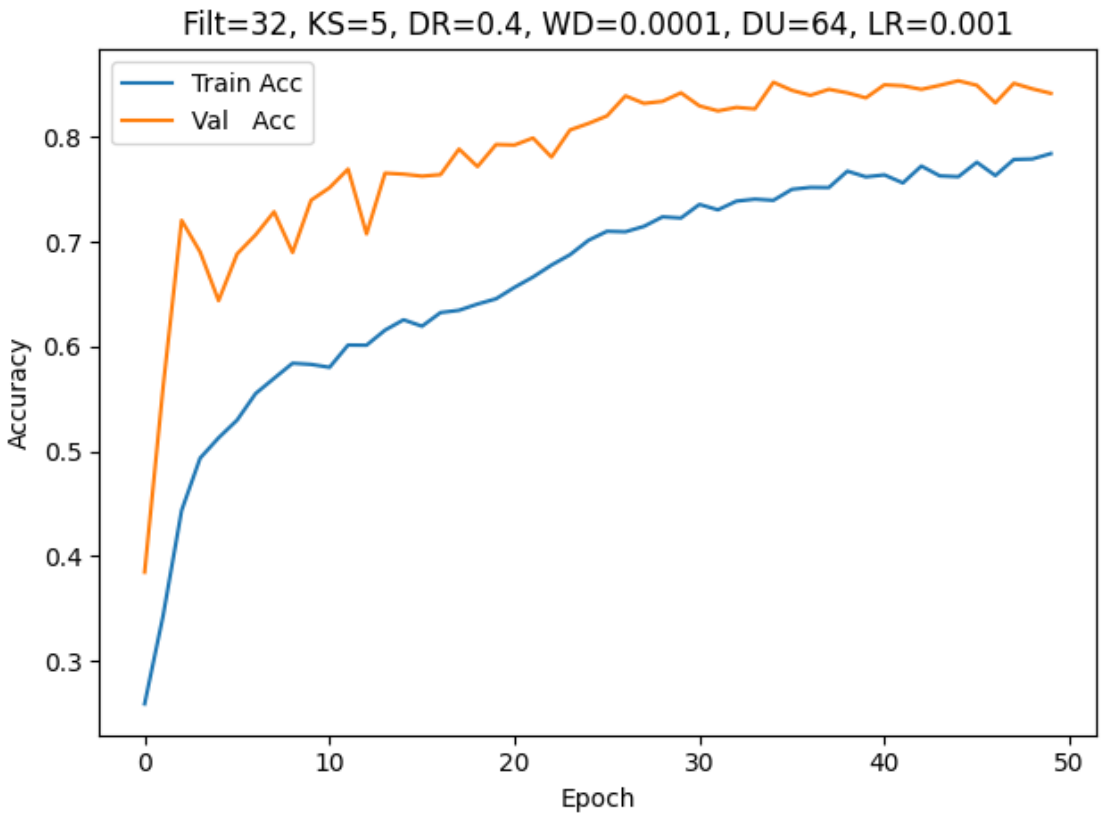

```
--> Best val_acc: 0.8668 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}
```



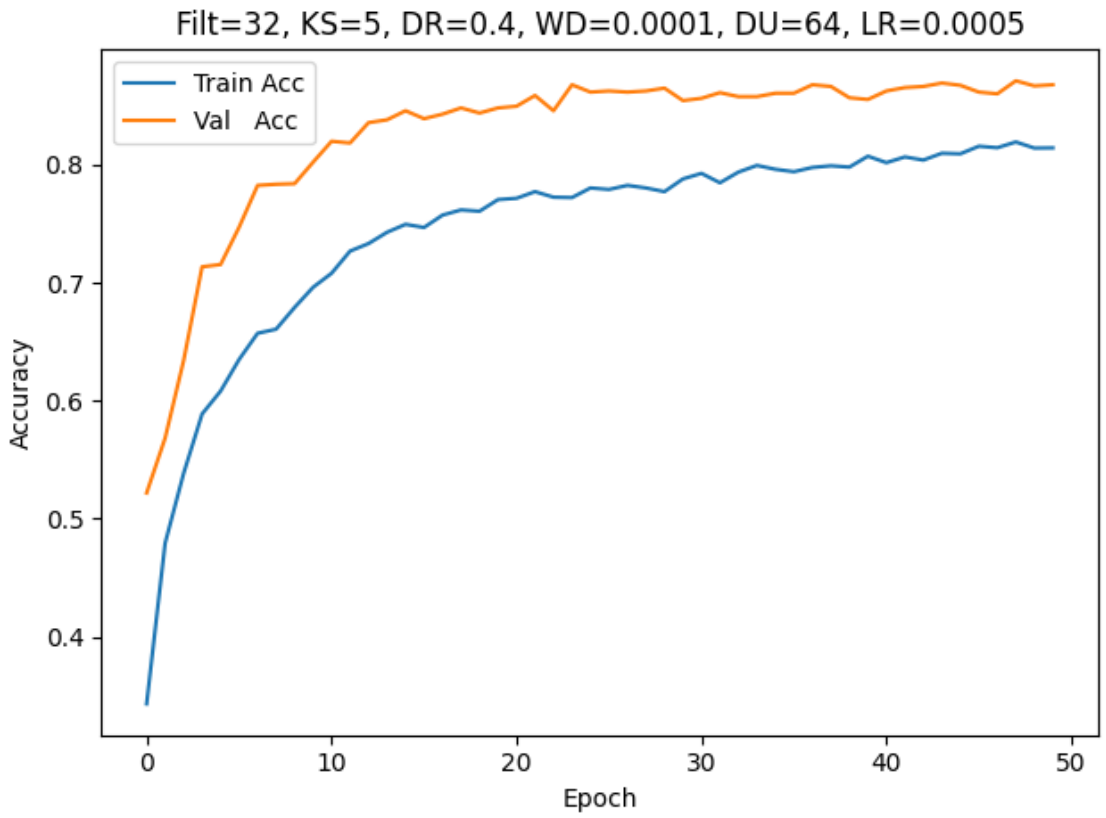
--> Best val_acc: 0.8769 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}



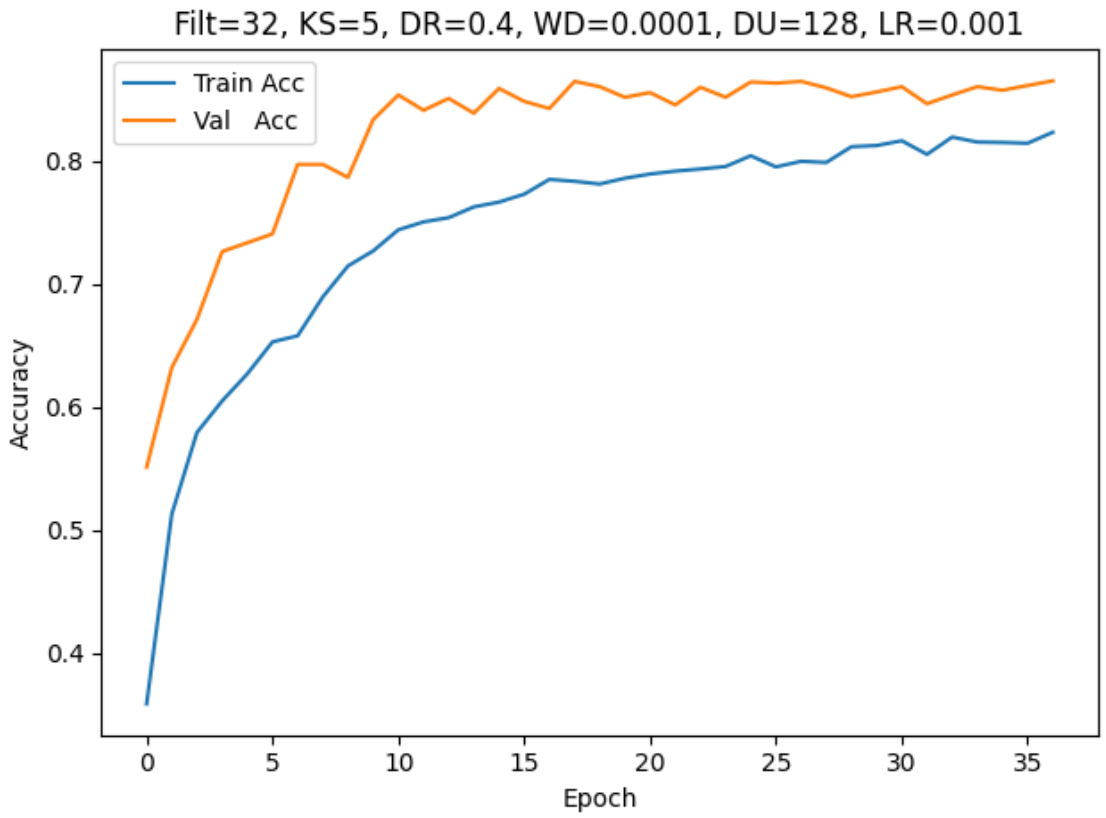

```
--> Best val_acc: 0.8754 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.2, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}
```



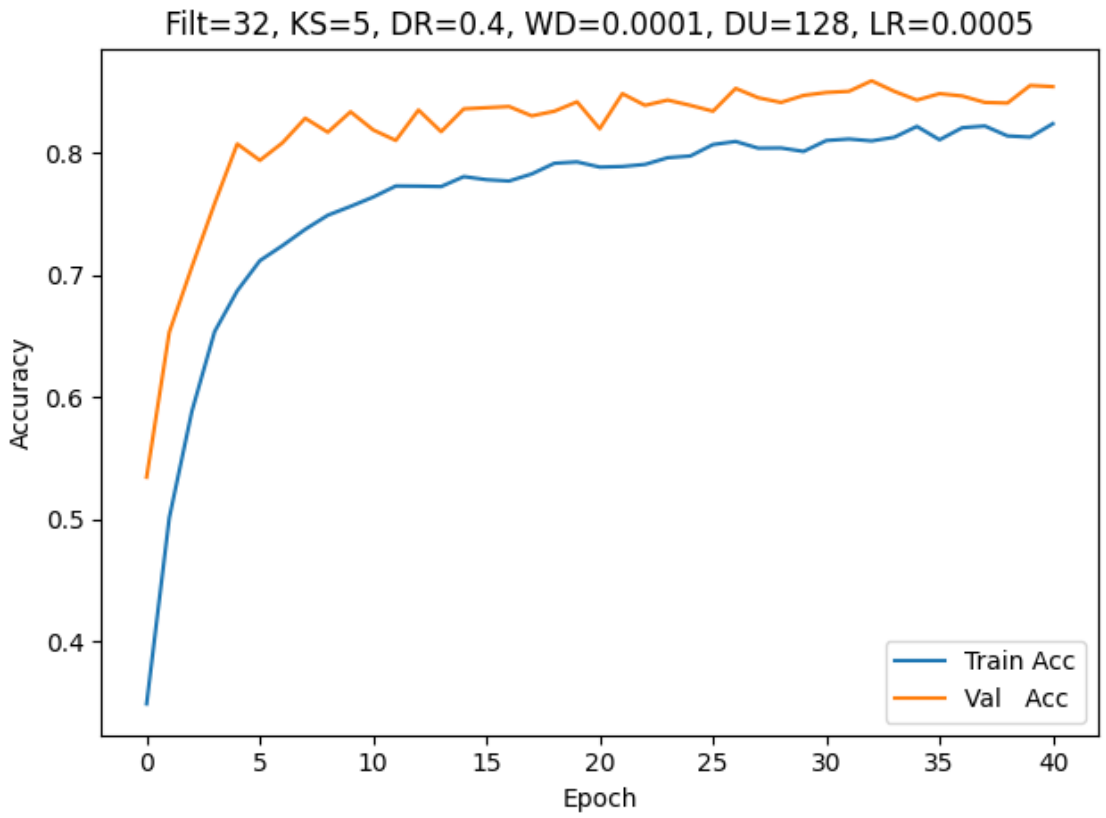

```
--> Best val_acc: 0.8539 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.001}
```



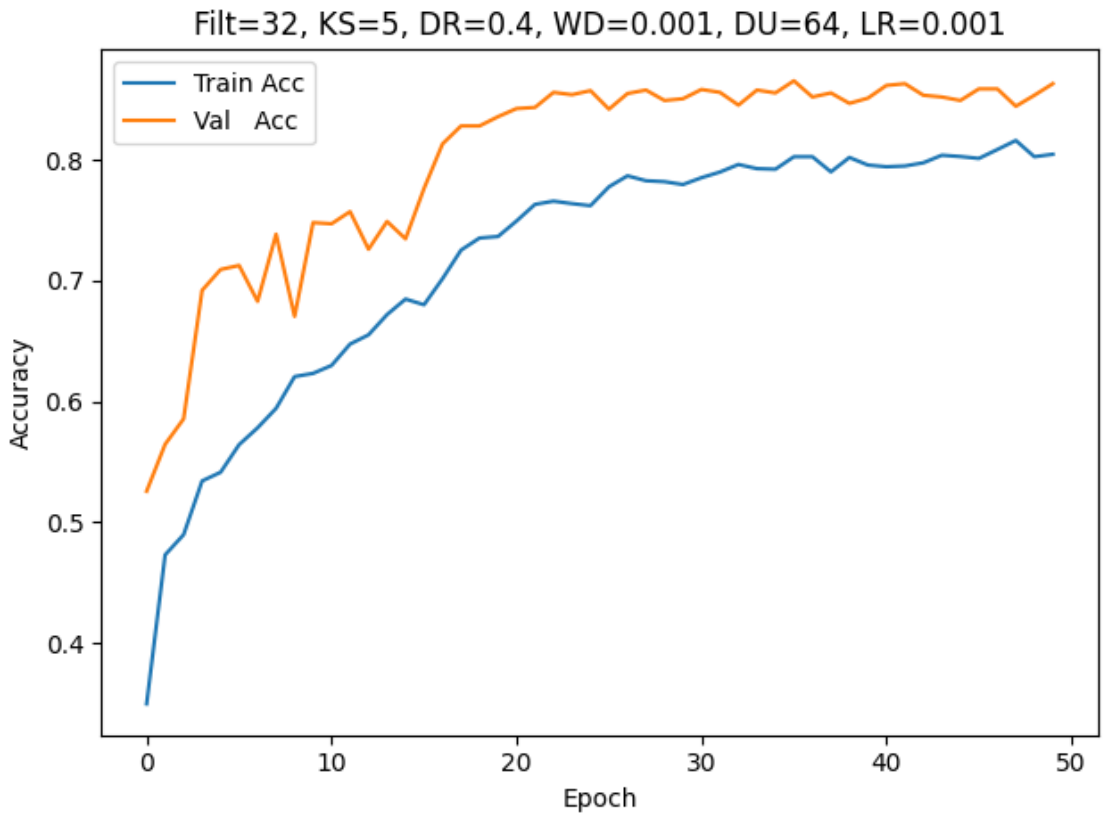

```
--> Best val_acc: 0.8711 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 64, 'lr': 0.0005}
```



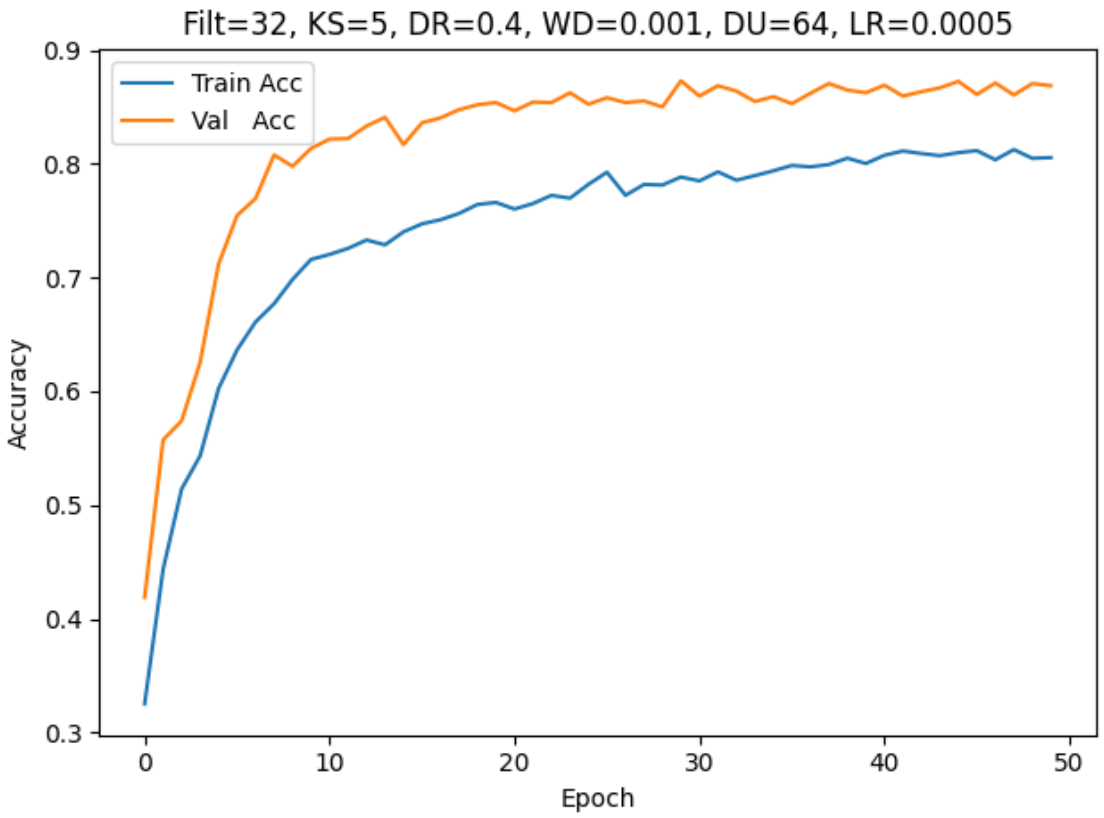
--> Best val_acc: 0.8654 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.001}



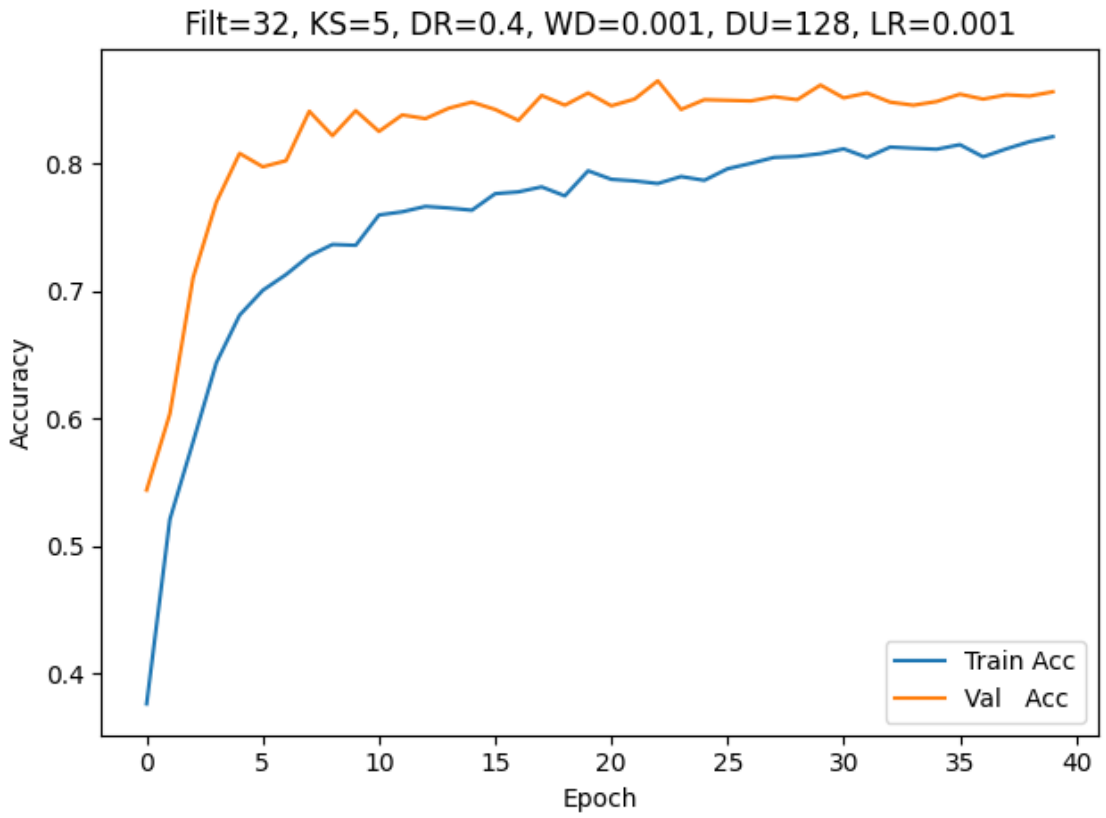
--> Best val_acc: 0.8591 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.4, 'weight_decay': 0.0001, 'dense_units': 128, 'lr': 0.0005}



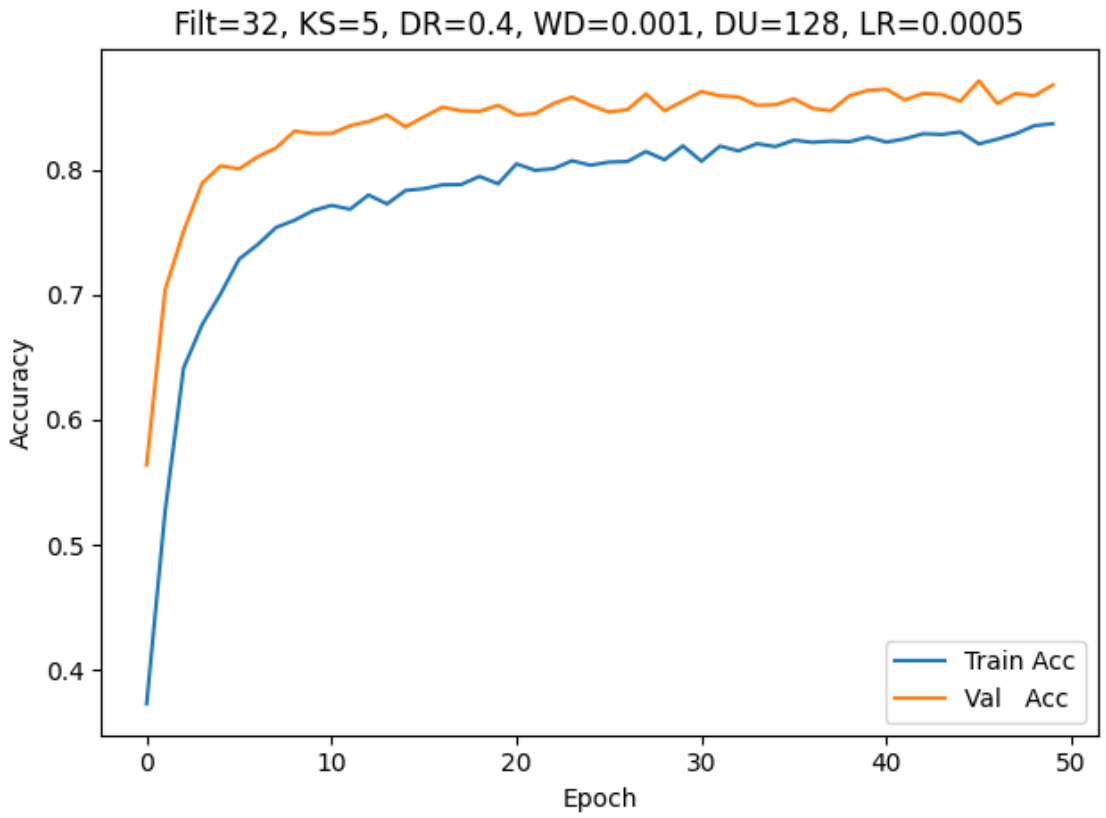

```
--> Best val_acc: 0.8654 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.001}
```



--> Best val_acc: 0.8730 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 64, 'lr': 0.0005}



--> Best val_acc: 0.8654 for config: {'filters': 32, 'kernel_size': 5, 'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.001}




```
--> Best val_acc: 0.8711 for config: {'filters': 32, 'kernel_size': 5,
'dropout_rate': 0.4, 'weight_decay': 0.001, 'dense_units': 128, 'lr': 0.0005}
```

1.5 Testing of the best model selection

Now the model is rebuilt using the chosen parameters selection.

[ ]:
```
# Model hyperparameters
filters       = 32          # Number of filters for Conv1D layers
kernel_size   = 3           # Kernel size for convolutions
dropout_rate  = 0.5         # Dropout rate for regularization
weight_decay  = 2e-3        # L2 regularization to prevent overfitting
dense_units   = 64          # Units in the dense (fully connected) layer
initial_lr    = 5e-4        # Initial learning rate for the optimizer

# Input shape info
timesteps, n_channels = X_train.shape[1], X_train.shape[2]

# Define the model architecture
model = Sequential([
```

```
    Input(shape=(timesteps, n_channels)),

    # First Conv block
    Conv1D(filters, kernel_size, activation='relu',␣
 ↪kernel_regularizer=l2(weight_decay)),
    BatchNormalization(),
    SpatialDropout1D(dropout_rate),

    # Second Conv block
    Conv1D(filters, kernel_size, activation='relu',␣
 ↪kernel_regularizer=l2(weight_decay)),
    BatchNormalization(),
    MaxPooling1D(2),
    Dropout(dropout_rate),

    # Third Conv block with double filters
    Conv1D(filters * 2, kernel_size, activation='relu',␣
 ↪kernel_regularizer=l2(weight_decay)),
    MaxPooling1D(2),
    Dropout(dropout_rate),

    # Flatten and dense layers
    Flatten(),
    Dense(dense_units, activation='relu', kernel_regularizer=l2(weight_decay)),
    Dropout(dropout_rate),

    # Output layer with softmax for multi-class classification
    Dense(num_classes, activation='softmax')
])

# Compile the model with Adam optimizer and cross-entropy loss
optimizer = Adam(learning_rate=initial_lr)
model.compile(optimizer, 'categorical_crossentropy', metrics=['accuracy'])

# Print model summary
model.summary()

# Set up training callbacks
cb_es  = EarlyStopping('val_loss', patience=15, restore_best_weights=True)    #␣
 ↪Stop early if no improvement
cb_rlr = ReduceLROnPlateau('val_loss', factor=0.5, patience=10, min_lr=1e-6)  #␣
 ↪Reduce LR on plateau
eval_cb = EvalTrainVal(train_data=(X_train, y_train_enc), val_data=(X_val,␣
 ↪y_val_enc))  # Custom evaluation

# Train the model
```

```
history = model.fit(
    X_train, y_train_enc,
    epochs=99,
    batch_size=32,
    validation_data=(X_val, y_val_enc),
    callbacks=[cb_es, cb_rlr, eval_cb],
    verbose=1
)

# Evaluate on test data
test_loss, test_acc = model.evaluate(X_test, y_test_enc, verbose=0)
print(f"\nTest Loss:     {test_loss:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")

# Predict and evaluate performance
y_pred = np.argmax(model.predict(X_test, verbose=0), axis=1) + offset
cm     = confusion_matrix(y_test, y_pred, labels=classes)

print("\nClassification Report:")
print(classification_report(y_test, y_pred, labels=classes))
```

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
conv1d_9 (Conv1D)	(None, 398, 32)	800
batch_normalization_6 (BatchNormalization)	(None, 398, 32)	128
spatial_dropout1d_3 (SpatialDropout1D)	(None, 398, 32)	0
conv1d_10 (Conv1D)	(None, 396, 32)	3,104
batch_normalization_7 (BatchNormalization)	(None, 396, 32)	128
max_pooling1d_6 (MaxPooling1D)	(None, 198, 32)	0
dropout_9 (Dropout)	(None, 198, 32)	0
conv1d_11 (Conv1D)	(None, 196, 64)	6,208
max_pooling1d_7 (MaxPooling1D)	(None, 98, 64)	0

```
 dropout_10 (Dropout)             (None, 98, 64)                  0

 flatten_3 (Flatten)              (None, 6272)                    0

 dense_6 (Dense)                  (None, 64)                401,472

 dropout_11 (Dropout)             (None, 64)                      0

 dense_7 (Dense)                  (None, 6)                     390


 Total params: 412,230 (1.57 MB)

 Trainable params: 412,102 (1.57 MB)

 Non-trainable params: 128 (512.00 B)

Epoch 1/99
223/223             0s 50ms/step -
accuracy: 0.1749 - loss: 2.4340 - eval_train_acc: 0.2987, eval_val_acc: 0.2868
223/223             17s 63ms/step -
accuracy: 0.1750 - loss: 2.4331 - val_accuracy: 0.2868 - val_loss: 2.0362 -
learning_rate: 5.0000e-04
Epoch 2/99
223/223             0s 49ms/step -
accuracy: 0.2721 - loss: 2.0248 - eval_train_acc: 0.3612, eval_val_acc: 0.3495
223/223             14s 65ms/step -
accuracy: 0.2722 - loss: 2.0246 - val_accuracy: 0.3495 - val_loss: 1.8722 -
learning_rate: 5.0000e-04
Epoch 3/99
223/223             0s 48ms/step -
accuracy: 0.3358 - loss: 1.8632 - eval_train_acc: 0.4882, eval_val_acc: 0.4734
223/223             20s 64ms/step -
accuracy: 0.3358 - loss: 1.8631 - val_accuracy: 0.4734 - val_loss: 1.7817 -
learning_rate: 5.0000e-04
Epoch 4/99
222/223             0s 49ms/step -
accuracy: 0.3634 - loss: 1.7466 - eval_train_acc: 0.5714, eval_val_acc: 0.5078
223/223             20s 62ms/step -
accuracy: 0.3635 - loss: 1.7462 - val_accuracy: 0.5078 - val_loss: 1.3655 -
learning_rate: 5.0000e-04
Epoch 5/99
223/223             0s 50ms/step -
accuracy: 0.3988 - loss: 1.5578 - eval_train_acc: 0.5509, eval_val_acc: 0.4890
223/223             20s 62ms/step -
accuracy: 0.3989 - loss: 1.5577 - val_accuracy: 0.4890 - val_loss: 1.2859 -
```

```
learning_rate: 5.0000e-04
Epoch 6/99
222/223             0s 50ms/step -
accuracy: 0.4424 - loss: 1.4625 - eval_train_acc: 0.5423, eval_val_acc: 0.4671
223/223             21s 62ms/step -
accuracy: 0.4424 - loss: 1.4622 - val_accuracy: 0.4671 - val_loss: 1.2513 -
learning_rate: 5.0000e-04
Epoch 7/99
223/223             0s 50ms/step -
accuracy: 0.4710 - loss: 1.3967 - eval_train_acc: 0.6184, eval_val_acc: 0.5502
223/223             21s 65ms/step -
accuracy: 0.4710 - loss: 1.3966 - val_accuracy: 0.5502 - val_loss: 1.1756 -
learning_rate: 5.0000e-04
Epoch 8/99
223/223             0s 49ms/step -
accuracy: 0.4826 - loss: 1.3394 - eval_train_acc: 0.5708, eval_val_acc: 0.5031
223/223             21s 65ms/step -
accuracy: 0.4826 - loss: 1.3393 - val_accuracy: 0.5031 - val_loss: 1.2148 -
learning_rate: 5.0000e-04
Epoch 9/99
222/223             0s 50ms/step -
accuracy: 0.4939 - loss: 1.3036 - eval_train_acc: 0.5644, eval_val_acc: 0.5517
223/223             21s 65ms/step -
accuracy: 0.4939 - loss: 1.3035 - val_accuracy: 0.5517 - val_loss: 1.1611 -
learning_rate: 5.0000e-04
Epoch 10/99
222/223             0s 50ms/step -
accuracy: 0.5138 - loss: 1.2149 - eval_train_acc: 0.6189, eval_val_acc: 0.5784
223/223             20s 62ms/step -
accuracy: 0.5138 - loss: 1.2150 - val_accuracy: 0.5784 - val_loss: 1.1769 -
learning_rate: 5.0000e-04
Epoch 11/99
223/223             0s 49ms/step -
accuracy: 0.5370 - loss: 1.1947 - eval_train_acc: 0.6433, eval_val_acc: 0.5909
223/223             21s 65ms/step -
accuracy: 0.5371 - loss: 1.1946 - val_accuracy: 0.5909 - val_loss: 1.0714 -
learning_rate: 5.0000e-04
Epoch 12/99
222/223             0s 49ms/step -
accuracy: 0.5457 - loss: 1.1678 - eval_train_acc: 0.6901, eval_val_acc: 0.6207
223/223             20s 61ms/step -
accuracy: 0.5457 - loss: 1.1677 - val_accuracy: 0.6207 - val_loss: 0.9934 -
learning_rate: 5.0000e-04
Epoch 13/99
223/223             0s 50ms/step -
accuracy: 0.5606 - loss: 1.1332 - eval_train_acc: 0.6774, eval_val_acc: 0.6176
223/223             20s 62ms/step -
accuracy: 0.5606 - loss: 1.1332 - val_accuracy: 0.6176 - val_loss: 1.0177 -
```

```
learning_rate: 5.0000e-04
Epoch 14/99
222/223             0s 47ms/step -
accuracy: 0.5691 - loss: 1.0905 - eval_train_acc: 0.6804, eval_val_acc: 0.6160
223/223             21s 64ms/step -
accuracy: 0.5691 - loss: 1.0906 - val_accuracy: 0.6160 - val_loss: 1.0001 -
learning_rate: 5.0000e-04
Epoch 15/99
222/223             0s 49ms/step -
accuracy: 0.5738 - loss: 1.1051 - eval_train_acc: 0.7176, eval_val_acc: 0.6536
223/223             21s 65ms/step -
accuracy: 0.5738 - loss: 1.1050 - val_accuracy: 0.6536 - val_loss: 0.9764 -
learning_rate: 5.0000e-04
Epoch 16/99
223/223             0s 49ms/step -
accuracy: 0.6190 - loss: 1.0490 - eval_train_acc: 0.7177, eval_val_acc: 0.6160
223/223             14s 65ms/step -
accuracy: 0.6190 - loss: 1.0490 - val_accuracy: 0.6160 - val_loss: 0.9876 -
learning_rate: 5.0000e-04
Epoch 17/99
223/223             0s 44ms/step -
accuracy: 0.5954 - loss: 1.0535 - eval_train_acc: 0.7132, eval_val_acc: 0.6364
223/223             19s 60ms/step -
accuracy: 0.5954 - loss: 1.0535 - val_accuracy: 0.6364 - val_loss: 0.9793 -
learning_rate: 5.0000e-04
Epoch 18/99
223/223             0s 50ms/step -
accuracy: 0.6082 - loss: 1.0779 - eval_train_acc: 0.7631, eval_val_acc: 0.7022
223/223             22s 65ms/step -
accuracy: 0.6082 - loss: 1.0778 - val_accuracy: 0.7022 - val_loss: 0.9084 -
learning_rate: 5.0000e-04
Epoch 19/99
223/223             0s 42ms/step -
accuracy: 0.6127 - loss: 1.0227 - eval_train_acc: 0.7737, eval_val_acc: 0.7179
223/223             21s 69ms/step -
accuracy: 0.6127 - loss: 1.0226 - val_accuracy: 0.7179 - val_loss: 0.9113 -
learning_rate: 5.0000e-04
Epoch 20/99
222/223             0s 50ms/step -
accuracy: 0.6376 - loss: 0.9825 - eval_train_acc: 0.8107, eval_val_acc: 0.7633
223/223             19s 62ms/step -
accuracy: 0.6376 - loss: 0.9826 - val_accuracy: 0.7633 - val_loss: 0.8372 -
learning_rate: 5.0000e-04
Epoch 21/99
223/223             0s 42ms/step -
accuracy: 0.6667 - loss: 0.9615 - eval_train_acc: 0.8136, eval_val_acc: 0.7539
223/223             20s 61ms/step -
accuracy: 0.6667 - loss: 0.9615 - val_accuracy: 0.7539 - val_loss: 0.8199 -
```

```
learning_rate: 5.0000e-04
Epoch 22/99
222/223            0s 51ms/step -
accuracy: 0.6693 - loss: 0.9432 - eval_train_acc: 0.8270, eval_val_acc: 0.7915
223/223            22s 66ms/step -
accuracy: 0.6693 - loss: 0.9432 - val_accuracy: 0.7915 - val_loss: 0.7882 -
learning_rate: 5.0000e-04
Epoch 23/99
223/223            0s 44ms/step -
accuracy: 0.6766 - loss: 0.9264 - eval_train_acc: 0.8327, eval_val_acc: 0.7837
223/223            19s 62ms/step -
accuracy: 0.6766 - loss: 0.9264 - val_accuracy: 0.7837 - val_loss: 0.7483 -
learning_rate: 5.0000e-04
Epoch 24/99
223/223            0s 44ms/step -
accuracy: 0.6820 - loss: 0.9106 - eval_train_acc: 0.8418, eval_val_acc: 0.7712
223/223            14s 61ms/step -
accuracy: 0.6820 - loss: 0.9106 - val_accuracy: 0.7712 - val_loss: 0.7443 -
learning_rate: 5.0000e-04
Epoch 25/99
223/223            0s 43ms/step -
accuracy: 0.6984 - loss: 0.8647 - eval_train_acc: 0.8331, eval_val_acc: 0.7806
223/223            14s 62ms/step -
accuracy: 0.6984 - loss: 0.8647 - val_accuracy: 0.7806 - val_loss: 0.7308 -
learning_rate: 5.0000e-04
Epoch 26/99
223/223            0s 50ms/step -
accuracy: 0.7042 - loss: 0.8803 - eval_train_acc: 0.8509, eval_val_acc: 0.8182
223/223            21s 65ms/step -
accuracy: 0.7042 - loss: 0.8803 - val_accuracy: 0.8182 - val_loss: 0.6927 -
learning_rate: 5.0000e-04
Epoch 27/99
223/223            0s 45ms/step -
accuracy: 0.7070 - loss: 0.8553 - eval_train_acc: 0.8544, eval_val_acc: 0.8041
223/223            20s 63ms/step -
accuracy: 0.7070 - loss: 0.8553 - val_accuracy: 0.8041 - val_loss: 0.6962 -
learning_rate: 5.0000e-04
Epoch 28/99
222/223            0s 50ms/step -
accuracy: 0.7076 - loss: 0.8449 - eval_train_acc: 0.8575, eval_val_acc: 0.8025
223/223            21s 65ms/step -
accuracy: 0.7076 - loss: 0.8450 - val_accuracy: 0.8025 - val_loss: 0.6895 -
learning_rate: 5.0000e-04
Epoch 29/99
223/223            0s 50ms/step -
accuracy: 0.7117 - loss: 0.8529 - eval_train_acc: 0.8517, eval_val_acc: 0.8197
223/223            14s 62ms/step -
accuracy: 0.7116 - loss: 0.8529 - val_accuracy: 0.8197 - val_loss: 0.6464 -
```

```
learning_rate: 5.0000e-04
Epoch 30/99
223/223            0s 47ms/step -
accuracy: 0.7285 - loss: 0.8344 - eval_train_acc: 0.8638, eval_val_acc: 0.8307
223/223            21s 65ms/step -
accuracy: 0.7285 - loss: 0.8345 - val_accuracy: 0.8307 - val_loss: 0.6570 -
learning_rate: 5.0000e-04
Epoch 31/99
222/223            0s 50ms/step -
accuracy: 0.7236 - loss: 0.8349 - eval_train_acc: 0.8562, eval_val_acc: 0.8103
223/223            20s 62ms/step -
accuracy: 0.7236 - loss: 0.8350 - val_accuracy: 0.8103 - val_loss: 0.6627 -
learning_rate: 5.0000e-04
Epoch 32/99
223/223            0s 49ms/step -
accuracy: 0.7211 - loss: 0.8327 - eval_train_acc: 0.8678, eval_val_acc: 0.8182
223/223            21s 66ms/step -
accuracy: 0.7211 - loss: 0.8326 - val_accuracy: 0.8182 - val_loss: 0.6302 -
learning_rate: 5.0000e-04
Epoch 33/99
222/223            0s 50ms/step -
accuracy: 0.7343 - loss: 0.8050 - eval_train_acc: 0.8537, eval_val_acc: 0.8213
223/223            20s 65ms/step -
accuracy: 0.7342 - loss: 0.8051 - val_accuracy: 0.8213 - val_loss: 0.6273 -
learning_rate: 5.0000e-04
Epoch 34/99
223/223            0s 50ms/step -
accuracy: 0.7324 - loss: 0.8293 - eval_train_acc: 0.8583, eval_val_acc: 0.8103
223/223            20s 63ms/step -
accuracy: 0.7324 - loss: 0.8293 - val_accuracy: 0.8103 - val_loss: 0.6607 -
learning_rate: 5.0000e-04
Epoch 35/99
222/223            0s 51ms/step -
accuracy: 0.7269 - loss: 0.7955 - eval_train_acc: 0.8617, eval_val_acc: 0.8260
223/223            21s 66ms/step -
accuracy: 0.7269 - loss: 0.7954 - val_accuracy: 0.8260 - val_loss: 0.6315 -
learning_rate: 5.0000e-04
Epoch 36/99
223/223            0s 50ms/step -
accuracy: 0.7409 - loss: 0.7739 - eval_train_acc: 0.8796, eval_val_acc: 0.8401
223/223            20s 66ms/step -
accuracy: 0.7409 - loss: 0.7740 - val_accuracy: 0.8401 - val_loss: 0.6136 -
learning_rate: 5.0000e-04
Epoch 37/99
222/223            0s 50ms/step -
accuracy: 0.7318 - loss: 0.8087 - eval_train_acc: 0.8624, eval_val_acc: 0.8213
223/223            20s 63ms/step -
accuracy: 0.7319 - loss: 0.8085 - val_accuracy: 0.8213 - val_loss: 0.6251 -
```

```
learning_rate: 5.0000e-04
Epoch 38/99
223/223            0s 50ms/step -
accuracy: 0.7378 - loss: 0.8056 - eval_train_acc: 0.8544, eval_val_acc: 0.8213
223/223            15s 66ms/step -
accuracy: 0.7378 - loss: 0.8055 - val_accuracy: 0.8213 - val_loss: 0.6271 -
learning_rate: 5.0000e-04
Epoch 39/99
223/223            0s 48ms/step -
accuracy: 0.7480 - loss: 0.7738 - eval_train_acc: 0.8590, eval_val_acc: 0.8119
223/223            20s 63ms/step -
accuracy: 0.7479 - loss: 0.7739 - val_accuracy: 0.8119 - val_loss: 0.6428 -
learning_rate: 5.0000e-04
Epoch 40/99
222/223            0s 50ms/step -
accuracy: 0.7428 - loss: 0.7831 - eval_train_acc: 0.8719, eval_val_acc: 0.8433
223/223            20s 63ms/step -
accuracy: 0.7428 - loss: 0.7831 - val_accuracy: 0.8433 - val_loss: 0.5890 -
learning_rate: 5.0000e-04
Epoch 41/99
223/223            0s 50ms/step -
accuracy: 0.7498 - loss: 0.7812 - eval_train_acc: 0.8604, eval_val_acc: 0.8401
223/223            15s 65ms/step -
accuracy: 0.7498 - loss: 0.7812 - val_accuracy: 0.8401 - val_loss: 0.6129 -
learning_rate: 5.0000e-04
Epoch 42/99
223/223            0s 49ms/step -
accuracy: 0.7482 - loss: 0.7598 - eval_train_acc: 0.8664, eval_val_acc: 0.8370
223/223            20s 63ms/step -
accuracy: 0.7482 - loss: 0.7599 - val_accuracy: 0.8370 - val_loss: 0.6293 -
learning_rate: 5.0000e-04
Epoch 43/99
223/223            0s 51ms/step -
accuracy: 0.7463 - loss: 0.7709 - eval_train_acc: 0.8570, eval_val_acc: 0.8197
223/223            20s 63ms/step -
accuracy: 0.7463 - loss: 0.7710 - val_accuracy: 0.8197 - val_loss: 0.6228 -
learning_rate: 5.0000e-04
Epoch 44/99
222/223            0s 51ms/step -
accuracy: 0.7416 - loss: 0.7773 - eval_train_acc: 0.8619, eval_val_acc: 0.8229
223/223            14s 64ms/step -
accuracy: 0.7416 - loss: 0.7773 - val_accuracy: 0.8229 - val_loss: 0.6074 -
learning_rate: 5.0000e-04
Epoch 45/99
222/223            0s 51ms/step -
accuracy: 0.7487 - loss: 0.7702 - eval_train_acc: 0.8677, eval_val_acc: 0.8417
223/223            15s 67ms/step -
accuracy: 0.7488 - loss: 0.7702 - val_accuracy: 0.8417 - val_loss: 0.5854 -
```

```
learning_rate: 5.0000e-04
Epoch 46/99
223/223            0s 50ms/step -
accuracy: 0.7581 - loss: 0.7486 - eval_train_acc: 0.8676, eval_val_acc: 0.8354
223/223            20s 64ms/step -
accuracy: 0.7581 - loss: 0.7486 - val_accuracy: 0.8354 - val_loss: 0.6067 -
learning_rate: 5.0000e-04
Epoch 47/99
223/223            0s 51ms/step -
accuracy: 0.7504 - loss: 0.7478 - eval_train_acc: 0.8656, eval_val_acc: 0.8354
223/223            21s 66ms/step -
accuracy: 0.7504 - loss: 0.7479 - val_accuracy: 0.8354 - val_loss: 0.5921 -
learning_rate: 5.0000e-04
Epoch 48/99
223/223            0s 51ms/step -
accuracy: 0.7578 - loss: 0.7524 - eval_train_acc: 0.8694, eval_val_acc: 0.8292
223/223            14s 63ms/step -
accuracy: 0.7578 - loss: 0.7524 - val_accuracy: 0.8292 - val_loss: 0.5825 -
learning_rate: 5.0000e-04
Epoch 49/99
223/223            0s 50ms/step -
accuracy: 0.7528 - loss: 0.7584 - eval_train_acc: 0.8792, eval_val_acc: 0.8354
223/223            21s 63ms/step -
accuracy: 0.7528 - loss: 0.7584 - val_accuracy: 0.8354 - val_loss: 0.5900 -
learning_rate: 5.0000e-04
Epoch 50/99
223/223            0s 51ms/step -
accuracy: 0.7657 - loss: 0.7322 - eval_train_acc: 0.8566, eval_val_acc: 0.8072
223/223            15s 65ms/step -
accuracy: 0.7657 - loss: 0.7322 - val_accuracy: 0.8072 - val_loss: 0.6202 -
learning_rate: 5.0000e-04
Epoch 51/99
223/223            0s 51ms/step -
accuracy: 0.7591 - loss: 0.7577 - eval_train_acc: 0.8698, eval_val_acc: 0.8417
223/223            21s 66ms/step -
accuracy: 0.7591 - loss: 0.7576 - val_accuracy: 0.8417 - val_loss: 0.5839 -
learning_rate: 5.0000e-04
Epoch 52/99
223/223            0s 51ms/step -
accuracy: 0.7537 - loss: 0.7526 - eval_train_acc: 0.8584, eval_val_acc: 0.8323
223/223            20s 63ms/step -
accuracy: 0.7537 - loss: 0.7526 - val_accuracy: 0.8323 - val_loss: 0.5892 -
learning_rate: 5.0000e-04
Epoch 53/99
223/223            0s 50ms/step -
accuracy: 0.7588 - loss: 0.7651 - eval_train_acc: 0.8684, eval_val_acc: 0.8511
223/223            21s 66ms/step -
accuracy: 0.7588 - loss: 0.7650 - val_accuracy: 0.8511 - val_loss: 0.6053 -
```

```
learning_rate: 5.0000e-04
Epoch 54/99
223/223            0s 51ms/step -
accuracy: 0.7653 - loss: 0.7393 - eval_train_acc: 0.8719, eval_val_acc: 0.8401
223/223            20s 63ms/step -
accuracy: 0.7654 - loss: 0.7393 - val_accuracy: 0.8401 - val_loss: 0.5698 -
learning_rate: 5.0000e-04
Epoch 55/99
222/223            0s 51ms/step -
accuracy: 0.7532 - loss: 0.7533 - eval_train_acc: 0.8580, eval_val_acc: 0.8307
223/223            20s 63ms/step -
accuracy: 0.7532 - loss: 0.7533 - val_accuracy: 0.8307 - val_loss: 0.6208 -
learning_rate: 5.0000e-04
Epoch 56/99
223/223            0s 50ms/step -
accuracy: 0.7535 - loss: 0.7728 - eval_train_acc: 0.8683, eval_val_acc: 0.8229
223/223            21s 66ms/step -
accuracy: 0.7535 - loss: 0.7727 - val_accuracy: 0.8229 - val_loss: 0.5881 -
learning_rate: 5.0000e-04
Epoch 57/99
223/223            0s 51ms/step -
accuracy: 0.7667 - loss: 0.7450 - eval_train_acc: 0.8702, eval_val_acc: 0.8197
223/223            20s 63ms/step -
accuracy: 0.7667 - loss: 0.7449 - val_accuracy: 0.8197 - val_loss: 0.5893 -
learning_rate: 5.0000e-04
Epoch 58/99
223/223            0s 50ms/step -
accuracy: 0.7734 - loss: 0.7345 - eval_train_acc: 0.8402, eval_val_acc: 0.8025
223/223            21s 66ms/step -
accuracy: 0.7734 - loss: 0.7345 - val_accuracy: 0.8025 - val_loss: 0.6441 -
learning_rate: 5.0000e-04
Epoch 59/99
222/223            0s 51ms/step -
accuracy: 0.7581 - loss: 0.7411 - eval_train_acc: 0.8781, eval_val_acc: 0.8386
223/223            14s 63ms/step -
accuracy: 0.7581 - loss: 0.7412 - val_accuracy: 0.8386 - val_loss: 0.5879 -
learning_rate: 5.0000e-04
Epoch 60/99
222/223            0s 51ms/step -
accuracy: 0.7639 - loss: 0.7340 - eval_train_acc: 0.8868, eval_val_acc: 0.8354
223/223            21s 63ms/step -
accuracy: 0.7639 - loss: 0.7340 - val_accuracy: 0.8354 - val_loss: 0.5922 -
learning_rate: 5.0000e-04
Epoch 61/99
223/223            0s 51ms/step -
accuracy: 0.7670 - loss: 0.7319 - eval_train_acc: 0.8645, eval_val_acc: 0.8245
223/223            21s 67ms/step -
accuracy: 0.7670 - loss: 0.7319 - val_accuracy: 0.8245 - val_loss: 0.5818 -
```

```
learning_rate: 5.0000e-04
Epoch 62/99
223/223             0s 51ms/step -
accuracy: 0.7687 - loss: 0.7256 - eval_train_acc: 0.8755, eval_val_acc: 0.8213
223/223             14s 64ms/step -
accuracy: 0.7686 - loss: 0.7256 - val_accuracy: 0.8213 - val_loss: 0.5877 -
learning_rate: 5.0000e-04
Epoch 63/99
222/223             0s 51ms/step -
accuracy: 0.7567 - loss: 0.7516 - eval_train_acc: 0.8842, eval_val_acc: 0.8370
223/223             21s 66ms/step -
accuracy: 0.7567 - loss: 0.7514 - val_accuracy: 0.8370 - val_loss: 0.5871 -
learning_rate: 5.0000e-04
Epoch 64/99
223/223             0s 51ms/step -
accuracy: 0.7576 - loss: 0.7318 - eval_train_acc: 0.8776, eval_val_acc: 0.8386
223/223             20s 67ms/step -
accuracy: 0.7575 - loss: 0.7318 - val_accuracy: 0.8386 - val_loss: 0.5788 -
learning_rate: 5.0000e-04
Epoch 65/99
222/223             0s 51ms/step -
accuracy: 0.7887 - loss: 0.7123 - eval_train_acc: 0.8862, eval_val_acc: 0.8386
223/223             20s 66ms/step -
accuracy: 0.7886 - loss: 0.7124 - val_accuracy: 0.8386 - val_loss: 0.5628 -
learning_rate: 2.5000e-04
Epoch 66/99
222/223             0s 50ms/step -
accuracy: 0.7602 - loss: 0.7265 - eval_train_acc: 0.8783, eval_val_acc: 0.8370
223/223             14s 63ms/step -
accuracy: 0.7603 - loss: 0.7263 - val_accuracy: 0.8370 - val_loss: 0.5712 -
learning_rate: 2.5000e-04
Epoch 67/99
222/223             0s 50ms/step -
accuracy: 0.7794 - loss: 0.6931 - eval_train_acc: 0.8778, eval_val_acc: 0.8260
223/223             21s 65ms/step -
accuracy: 0.7794 - loss: 0.6931 - val_accuracy: 0.8260 - val_loss: 0.5625 -
learning_rate: 2.5000e-04
Epoch 68/99
223/223             0s 51ms/step -
accuracy: 0.7749 - loss: 0.6974 - eval_train_acc: 0.8806, eval_val_acc: 0.8323
223/223             21s 68ms/step -
accuracy: 0.7749 - loss: 0.6974 - val_accuracy: 0.8323 - val_loss: 0.5591 -
learning_rate: 2.5000e-04
Epoch 69/99
223/223             0s 51ms/step -
accuracy: 0.7815 - loss: 0.6686 - eval_train_acc: 0.8782, eval_val_acc: 0.8354
223/223             20s 66ms/step -
accuracy: 0.7815 - loss: 0.6687 - val_accuracy: 0.8354 - val_loss: 0.5711 -
```

```
learning_rate: 2.5000e-04
Epoch 70/99
222/223            0s 51ms/step -
accuracy: 0.7885 - loss: 0.6739 - eval_train_acc: 0.8734, eval_val_acc: 0.8370
223/223            20s 63ms/step -
accuracy: 0.7884 - loss: 0.6740 - val_accuracy: 0.8370 - val_loss: 0.5557 -
learning_rate: 2.5000e-04
Epoch 71/99
222/223            0s 51ms/step -
accuracy: 0.7781 - loss: 0.6811 - eval_train_acc: 0.8917, eval_val_acc: 0.8495
223/223            15s 66ms/step -
accuracy: 0.7781 - loss: 0.6811 - val_accuracy: 0.8495 - val_loss: 0.5299 -
learning_rate: 2.5000e-04
Epoch 72/99
223/223            0s 50ms/step -
accuracy: 0.7702 - loss: 0.6956 - eval_train_acc: 0.8870, eval_val_acc: 0.8401
223/223            15s 66ms/step -
accuracy: 0.7702 - loss: 0.6955 - val_accuracy: 0.8401 - val_loss: 0.5385 -
learning_rate: 2.5000e-04
Epoch 73/99
223/223            0s 51ms/step -
accuracy: 0.7808 - loss: 0.6687 - eval_train_acc: 0.8917, eval_val_acc: 0.8433
223/223            20s 63ms/step -
accuracy: 0.7808 - loss: 0.6687 - val_accuracy: 0.8433 - val_loss: 0.5304 -
learning_rate: 2.5000e-04
Epoch 74/99
223/223            0s 52ms/step -
accuracy: 0.7894 - loss: 0.6596 - eval_train_acc: 0.8779, eval_val_acc: 0.8260
223/223            15s 67ms/step -
accuracy: 0.7894 - loss: 0.6597 - val_accuracy: 0.8260 - val_loss: 0.5600 -
learning_rate: 2.5000e-04
Epoch 75/99
222/223            0s 51ms/step -
accuracy: 0.7802 - loss: 0.6833 - eval_train_acc: 0.8872, eval_val_acc: 0.8354
223/223            15s 67ms/step -
accuracy: 0.7802 - loss: 0.6832 - val_accuracy: 0.8354 - val_loss: 0.5354 -
learning_rate: 2.5000e-04
Epoch 76/99
223/223            0s 51ms/step -
accuracy: 0.7912 - loss: 0.6465 - eval_train_acc: 0.8785, eval_val_acc: 0.8260
223/223            20s 63ms/step -
accuracy: 0.7912 - loss: 0.6465 - val_accuracy: 0.8260 - val_loss: 0.5517 -
learning_rate: 2.5000e-04
Epoch 77/99
223/223            0s 51ms/step -
accuracy: 0.7756 - loss: 0.6815 - eval_train_acc: 0.8894, eval_val_acc: 0.8323
223/223            15s 66ms/step -
accuracy: 0.7757 - loss: 0.6815 - val_accuracy: 0.8323 - val_loss: 0.5308 -
```

```
learning_rate: 2.5000e-04
Epoch 78/99
222/223            0s 51ms/step -
accuracy: 0.7899 - loss: 0.6394 - eval_train_acc: 0.8789, eval_val_acc: 0.8401
223/223            15s 67ms/step -
accuracy: 0.7899 - loss: 0.6396 - val_accuracy: 0.8401 - val_loss: 0.5437 -
learning_rate: 2.5000e-04
Epoch 79/99
222/223            0s 51ms/step -
accuracy: 0.7804 - loss: 0.6731 - eval_train_acc: 0.8804, eval_val_acc: 0.8386
223/223            20s 67ms/step -
accuracy: 0.7804 - loss: 0.6731 - val_accuracy: 0.8386 - val_loss: 0.5291 -
learning_rate: 2.5000e-04
Epoch 80/99
222/223            0s 51ms/step -
accuracy: 0.7877 - loss: 0.6440 - eval_train_acc: 0.8856, eval_val_acc: 0.8339
223/223            21s 67ms/step -
accuracy: 0.7877 - loss: 0.6442 - val_accuracy: 0.8339 - val_loss: 0.5397 -
learning_rate: 2.5000e-04
Epoch 81/99
222/223            0s 52ms/step -
accuracy: 0.7802 - loss: 0.6692 - eval_train_acc: 0.8754, eval_val_acc: 0.8354
223/223            21s 68ms/step -
accuracy: 0.7802 - loss: 0.6692 - val_accuracy: 0.8354 - val_loss: 0.5534 -
learning_rate: 2.5000e-04
Epoch 82/99
223/223            0s 52ms/step -
accuracy: 0.7857 - loss: 0.6580 - eval_train_acc: 0.8796, eval_val_acc: 0.8401
223/223            20s 64ms/step -
accuracy: 0.7857 - loss: 0.6580 - val_accuracy: 0.8401 - val_loss: 0.5439 -
learning_rate: 2.5000e-04
Epoch 83/99
222/223            0s 52ms/step -
accuracy: 0.7884 - loss: 0.6494 - eval_train_acc: 0.8907, eval_val_acc: 0.8386
223/223            14s 65ms/step -
accuracy: 0.7884 - loss: 0.6494 - val_accuracy: 0.8386 - val_loss: 0.5271 -
learning_rate: 2.5000e-04
Epoch 84/99
223/223            0s 50ms/step -
accuracy: 0.7870 - loss: 0.6599 - eval_train_acc: 0.8896, eval_val_acc: 0.8339
223/223            20s 64ms/step -
accuracy: 0.7870 - loss: 0.6599 - val_accuracy: 0.8339 - val_loss: 0.5441 -
learning_rate: 2.5000e-04
Epoch 85/99
223/223            0s 51ms/step -
accuracy: 0.7833 - loss: 0.6675 - eval_train_acc: 0.8730, eval_val_acc: 0.8166
223/223            21s 66ms/step -
accuracy: 0.7833 - loss: 0.6675 - val_accuracy: 0.8166 - val_loss: 0.5529 -
```

```
learning_rate: 2.5000e-04
Epoch 86/99
223/223            0s 49ms/step -
accuracy: 0.7934 - loss: 0.6409 - eval_train_acc: 0.8813, eval_val_acc: 0.8323
223/223            21s 66ms/step -
accuracy: 0.7934 - loss: 0.6409 - val_accuracy: 0.8323 - val_loss: 0.5323 -
learning_rate: 2.5000e-04
Epoch 87/99
222/223            0s 52ms/step -
accuracy: 0.7893 - loss: 0.6441 - eval_train_acc: 0.8940, eval_val_acc: 0.8323
223/223            21s 68ms/step -
accuracy: 0.7893 - loss: 0.6442 - val_accuracy: 0.8323 - val_loss: 0.5314 -
learning_rate: 2.5000e-04
Epoch 88/99
223/223            0s 49ms/step -
accuracy: 0.7828 - loss: 0.6684 - eval_train_acc: 0.8834, eval_val_acc: 0.8323
223/223            20s 63ms/step -
accuracy: 0.7828 - loss: 0.6683 - val_accuracy: 0.8323 - val_loss: 0.5287 -
learning_rate: 2.5000e-04
Epoch 89/99
223/223            0s 51ms/step -
accuracy: 0.8077 - loss: 0.6271 - eval_train_acc: 0.8963, eval_val_acc: 0.8401
223/223            20s 63ms/step -
accuracy: 0.8076 - loss: 0.6271 - val_accuracy: 0.8401 - val_loss: 0.5356 -
learning_rate: 2.5000e-04
Epoch 90/99
223/223            0s 49ms/step -
accuracy: 0.8014 - loss: 0.6371 - eval_train_acc: 0.8851, eval_val_acc: 0.8323
223/223            21s 66ms/step -
accuracy: 0.8013 - loss: 0.6372 - val_accuracy: 0.8323 - val_loss: 0.5385 -
learning_rate: 2.5000e-04
Epoch 91/99
223/223            0s 50ms/step -
accuracy: 0.7890 - loss: 0.6442 - eval_train_acc: 0.8868, eval_val_acc: 0.8401
223/223            14s 63ms/step -
accuracy: 0.7890 - loss: 0.6442 - val_accuracy: 0.8401 - val_loss: 0.5098 -
learning_rate: 2.5000e-04
Epoch 92/99
222/223            0s 51ms/step -
accuracy: 0.7899 - loss: 0.6484 - eval_train_acc: 0.8970, eval_val_acc: 0.8307
223/223            21s 67ms/step -
accuracy: 0.7899 - loss: 0.6483 - val_accuracy: 0.8307 - val_loss: 0.5168 -
learning_rate: 2.5000e-04
Epoch 93/99
223/223            0s 49ms/step -
accuracy: 0.7911 - loss: 0.6444 - eval_train_acc: 0.9001, eval_val_acc: 0.8307
223/223            20s 67ms/step -
accuracy: 0.7911 - loss: 0.6444 - val_accuracy: 0.8307 - val_loss: 0.5217 -
```

```
learning_rate: 2.5000e-04
Epoch 94/99
222/223             0s 51ms/step -
accuracy: 0.7880 - loss: 0.6469 - eval_train_acc: 0.8981, eval_val_acc: 0.8386
223/223             20s 67ms/step -
accuracy: 0.7880 - loss: 0.6468 - val_accuracy: 0.8386 - val_loss: 0.5162 -
learning_rate: 2.5000e-04
Epoch 95/99
223/223             0s 49ms/step -
accuracy: 0.7964 - loss: 0.6401 - eval_train_acc: 0.8917, eval_val_acc: 0.8417
223/223             20s 63ms/step -
accuracy: 0.7964 - loss: 0.6401 - val_accuracy: 0.8417 - val_loss: 0.5221 -
learning_rate: 2.5000e-04
Epoch 96/99
223/223             0s 51ms/step -
accuracy: 0.7780 - loss: 0.6542 - eval_train_acc: 0.9006, eval_val_acc: 0.8354
223/223             21s 67ms/step -
accuracy: 0.7781 - loss: 0.6541 - val_accuracy: 0.8354 - val_loss: 0.5168 -
learning_rate: 2.5000e-04
Epoch 97/99
223/223             0s 50ms/step -
accuracy: 0.7989 - loss: 0.6262 - eval_train_acc: 0.9013, eval_val_acc: 0.8386
223/223             20s 67ms/step -
accuracy: 0.7989 - loss: 0.6263 - val_accuracy: 0.8386 - val_loss: 0.5052 -
learning_rate: 2.5000e-04
Epoch 98/99
222/223             0s 52ms/step -
accuracy: 0.7943 - loss: 0.6330 - eval_train_acc: 0.8964, eval_val_acc: 0.8370
223/223             21s 68ms/step -
accuracy: 0.7943 - loss: 0.6330 - val_accuracy: 0.8370 - val_loss: 0.5111 -
learning_rate: 2.5000e-04
Epoch 99/99
223/223             0s 50ms/step -
accuracy: 0.7899 - loss: 0.6311 - eval_train_acc: 0.9048, eval_val_acc: 0.8511
223/223             20s 64ms/step -
accuracy: 0.7899 - loss: 0.6311 - val_accuracy: 0.8511 - val_loss: 0.5049 -
learning_rate: 2.5000e-04

Test Loss:     0.5081
Test Accuracy: 0.8454

Classification Report:
              precision    recall  f1-score   support

           1       0.98      0.99      0.98       121
           2       0.87      0.88      0.88       109
           3       0.75      0.80      0.77       114
           4       0.82      0.85      0.84       116
```

```
           5       0.83      0.88      0.86       119
           6       0.82      0.65      0.73       113

    accuracy                           0.85       692
   macro avg       0.84      0.84      0.84       692
weighted avg       0.85      0.85      0.84       692
```

1.6 Window Size and Step Tunning

Grid search over different combinations of window sizes and step values to optimize segmentation

[ ]:
```python
# Define grid of (window size, step) combinations to test
window_sizes = [250, 400, 500]
steps        = [100, 175, 250]
grid = list(itertools.product(window_sizes, steps))


# Build windowed train/val/test datasets for a given window + step config
def make_datasets(window_size, train_step, val_step, test_step):
    train_w, val_w, test_w = {}, {}, {}

    for key, gdf in gesture_datasets.items():
        cols = [c for c in gdf.columns if c.startswith("channel")]
        sig  = gdf[cols].to_numpy().T  # shape: (channels, samples)

        # Route each gesture to its split and extract windows
        if   key in train_segs: step, target = train_step, train_w
        elif key in val_segs:   step, target = val_step,   val_w
        else:                   step, target = test_step,  test_w

        wins = extract_deterministic_windows(sig, window_size, step)
        for i, w in enumerate(wins):
            target[f"{key}_win_{i}"] = w

    # Helper to stack data and extract class labels
    def prepare(wdict):
        X = np.stack([w.T for w in wdict.values()], axis=0)
        y = np.array([int(k.split('_')[1]) for k in wdict.keys()])
        return X, y

    # Prepare all three splits
    X_tr, y_tr = prepare(train_w)
    X_va, y_va = prepare(val_w)
    X_te, y_te = prepare(test_w)

    # One-hot encode class labels
    classes_arr = np.unique(y_tr)
```

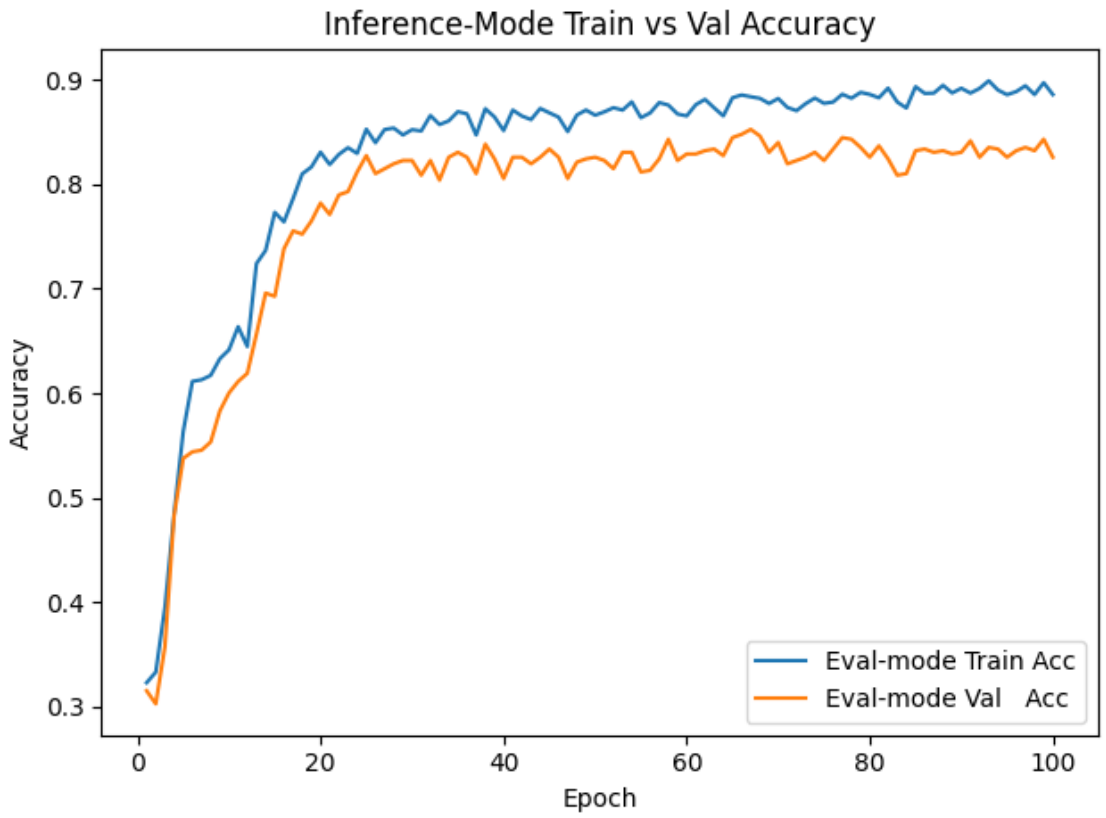
```
    offset      = classes_arr.min()
    num_classes = len(classes_arr)
    y_tr_enc = to_categorical(y_tr - offset, num_classes)
    y_va_enc = to_categorical(y_va - offset, num_classes)
    y_te_enc = to_categorical(y_te - offset, num_classes)

    # Normalize each split using train set statistics
    mean = X_tr.mean(axis=(0,1), keepdims=True)
    std  = X_tr.std(axis=(0,1), keepdims=True) + 1e-8
    for arr in (X_tr, X_va, X_te):
        arr[:] = (arr - mean) / std

    return (X_tr, y_tr_enc), (X_va, y_va_enc), (X_te, y_te_enc), (y_te, offset,␣
 ↪num_classes)


# Build a 1D-CNN for gesture classification
def get_model(timesteps, n_channels, num_classes,
              filters=32, kernel_size=3, dropout_rate=0.5, weight_decay=2e-3,␣
 ↪dense_units=64, lr=5e-4):
    model = Sequential([
        Input(shape=(timesteps, n_channels)),
        Conv1D(filters, kernel_size, activation='relu',␣
 ↪kernel_regularizer=l2(weight_decay)),
        BatchNormalization(), SpatialDropout1D(dropout_rate),

        Conv1D(filters, kernel_size, activation='relu',␣
 ↪kernel_regularizer=l2(weight_decay)),
        BatchNormalization(), MaxPooling1D(2), Dropout(dropout_rate),

        Conv1D(filters*2, kernel_size, activation='relu',␣
 ↪kernel_regularizer=l2(weight_decay)),
        MaxPooling1D(2), Dropout(dropout_rate),

        Flatten(),
        Dense(dense_units, activation='relu',␣
 ↪kernel_regularizer=l2(weight_decay)),
        Dropout(dropout_rate),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(optimizer=Adam(learning_rate=lr),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```
# Run tuning over all (window, step) combinations
results = []
for win, step in grid:
    print(f"\n GRID: window={win}, step={step}")

    # Build datasets for current window config
    (X_tr, y_tr_enc), (X_va, y_va_enc), (X_te, y_te_enc), (y_te, offset,␣
 ↪num_classes) = \
        make_datasets(window_size=win, train_step=step, val_step=step,␣
 ↪test_step=step)

    # Build CNN model
    timesteps, n_ch = X_tr.shape[1], X_tr.shape[2]
    model = get_model(timesteps, n_ch, num_classes)

    # Callbacks: early stopping and learning rate reduction
    cb_es  = EarlyStopping('val_loss', patience=10, restore_best_weights=True,␣
 ↪verbose=1)
    cb_rlr = ReduceLROnPlateau('val_loss', factor=0.5, patience=5, min_lr=1e-6,␣
 ↪verbose=1)

    # Train the model
    history = model.fit(
        X_tr, y_tr_enc,
        validation_data=(X_va, y_va_enc),
        epochs=50,
        batch_size=32,
        callbacks=[cb_es, cb_rlr],
        verbose=2
    )

    # Plot training and validation accuracy
    plt.figure(figsize=(8, 5))
    plt.plot(history.history['accuracy'],     label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title(f'Accuracy (window={win}, step={step})')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Evaluate on test set
    loss, acc = model.evaluate(X_te, y_te_enc, verbose=0)
    print(f"--> Test Acc: {acc:.4f}")
```

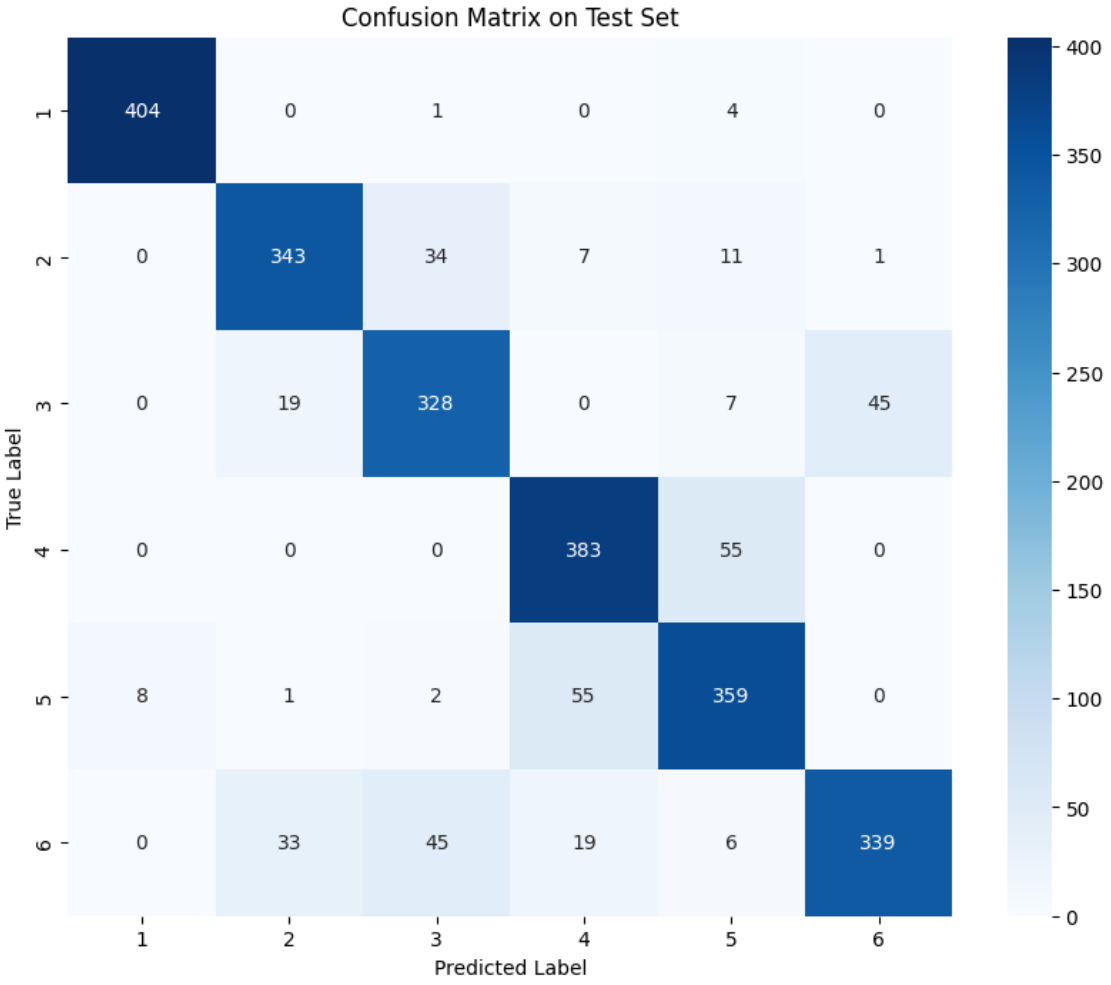
```
    # Save results
    results.append({'window_size': win, 'step': step, 'test_acc': acc})


# Summarize results in table
df_results = pd.DataFrame(results)
print("\nGrid search results:")
print(df_results.pivot(index='window_size', columns='step', values='test_acc'))
```

This final model reflects the best segmentation strategy and network architecture found during tuning

[ ]:
```
# Plot eval-mode accuracies

plt.figure()
epochs = range(1, len(eval_cb.eval_metrics['train_acc'])+1)
plt.plot(epochs, eval_cb.eval_metrics['train_acc'], label='Eval-mode Train Acc')
plt.plot(epochs, eval_cb.eval_metrics['val_acc'],   label='Eval-mode Val   Acc')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Inference-Mode Train vs Val Accuracy')
plt.legend()
plt.tight_layout()
plt.show()
```



[ ]:
```
# Predict class labels for test set
y_pred = np.argmax(model.predict(X_test, verbose=0), axis=1) + offset

# Compute confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=classes)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix on Test Set")
plt.show()

# Print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, labels=classes))
```



Classification Report:

	precision	recall	f1-score	support
1	0.98	0.99	0.98	409
2	0.87	0.87	0.87	396
3	0.80	0.82	0.81	399
4	0.83	0.87	0.85	438
5	0.81	0.84	0.83	425
6	0.88	0.77	0.82	442
accuracy			0.86	2509
macro avg	0.86	0.86	0.86	2509
weighted avg	0.86	0.86	0.86	2509

1.7 Segment-level evaluation

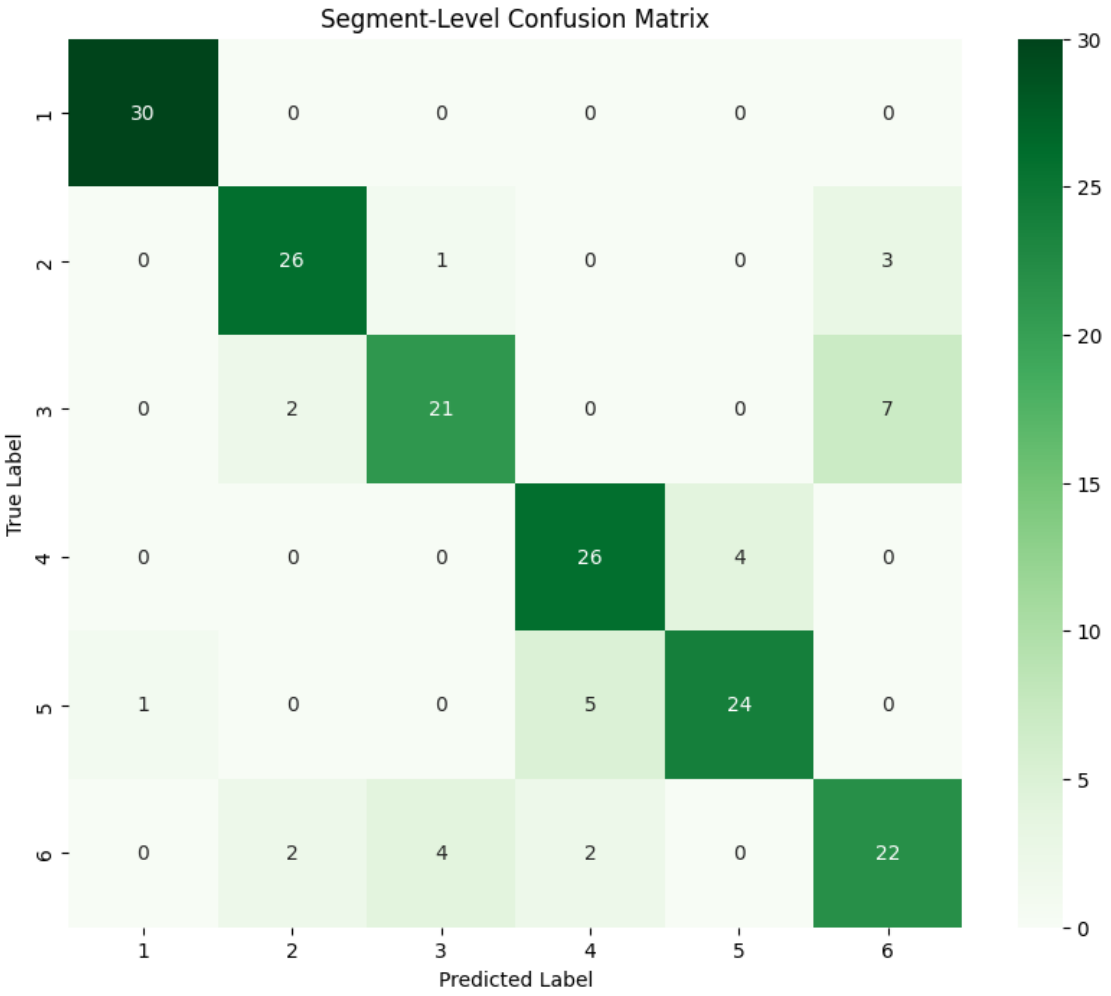
Instead of looking at the windows, the original segments are recovered and to determine the label of each segment, majority vote is applied.

[ ]:
```
# Predict window-wise classes
y_pred_test = np.argmax(model.predict(X_test, verbose=0), axis=1) + offset

# Map predictions back to segments
segment_preds = defaultdict(list)
segment_truth = {}

# Reconstruct segment keys from test_w keys
for i, key in enumerate(test_w.keys()):
    segment_id = '_'.join(key.split('_')[:3])
    true_class = int(segment_id.split('_')[1])
    segment_preds[segment_id].append(y_pred_test[i])
    segment_truth[segment_id] = true_class
```

[ ]:
```
# Apply majority vote
segment_final_preds = {}
for segment_id, preds in segment_preds.items():
    most_common = Counter(preds).most_common(1)[0][0]
    segment_final_preds[segment_id] = most_common
```

[ ]:
```
# Extract true and predicted labels
y_true_seg = [segment_truth[sid] for sid in segment_final_preds]
y_pred_seg = [segment_final_preds[sid] for sid in segment_final_preds]

# Accuracy
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
seg_acc = accuracy_score(y_true_seg, y_pred_seg)
print(f"Segment-Level Accuracy: {seg_acc:.4f}")

# Confusion matrix
cm_seg = confusion_matrix(y_true_seg, y_pred_seg, labels=classes)
plt.figure(figsize=(10, 8))
sns.heatmap(cm_seg, annot=True, fmt='d', cmap='Greens',
            xticklabels=classes, yticklabels=classes)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Segment-Level Confusion Matrix")
plt.show()

# report
print("\nSegment-Level Classification Report:")
print(classification_report(y_true_seg, y_pred_seg, labels=classes))
```

```
Segment-Level Accuracy: 0.8278
```




```
Segment-Level Classification Report:
              precision    recall  f1-score   support

           1       0.97      1.00      0.98        30
           2       0.87      0.87      0.87        30
           3       0.81      0.70      0.75        30
           4       0.79      0.87      0.83        30
           5       0.86      0.80      0.83        30
           6       0.69      0.73      0.71        30

    accuracy                           0.83       180
   macro avg       0.83      0.83      0.83       180
weighted avg       0.83      0.83      0.83       180
```