Vibhu Vanjari Module 8 Assignment 2 – Leaf Classification

Part 1 Code (Data Prep) - Link

Part 2 Code (NN DoE) - Link

Part 3 Code (CNN Models) - Link

Data taken from - https://www.kaggle.com/competitions/leaf-classification/data

Initial Data Setup and EDA (Part 1)

To begin the leaf classification project, I imported the necessary libraries and loaded the Kaggle dataset into the environment. I merged the training and testing data to allow consistent preprocessing and added a 'train' column to distinguish between the two. After confirming there were no missing values, I conducted basic EDA to examine the distribution of species. I encoded the species labels into numeric format using label encoding and then performed one-hot encoding to prepare the target variable for multi-class classification. I scaled the feature data using StandardScaler to normalize the input and split the data back into training and testing sets. These steps ensured the dataset was clean, properly formatted, and ready for training convolutional neural network models in the next phase.

Neural Network DOE (Part 2)

In the second phase of the project, I built and evaluated multiple neural network models using the provided tabular features (margin, shape, and texture) rather than raw image data. To improve model generalization, I applied data augmentation techniques using TensorFlow's sequential preprocessing layer, including random flips, rotations, and zooms. I then created a function to systematically construct and train neural networks with varying hyperparameters such as the number of layers, number of nodes per layer, and dropout rates. I ran a design of experiments (DoE) to train models with different combinations of these hyperparameters, recording training time, accuracy, and loss metrics for both training and validation sets.

After training all models, I compiled the results into a summary table to identify top-performing configurations. I selected models based on validation accuracy and examined their training curves to assess convergence and potential overfitting. I also generated multi-class ROC and precision-recall plots to further evaluate model performance across all species classes. These evaluations helped highlight tradeoffs between model complexity and generalization. Overall, this phase focused on tuning dense feedforward networks to make the most of the pre-extracted features and prepare for final model selection and Kaggle submission.

The key takeaways from this phase are:

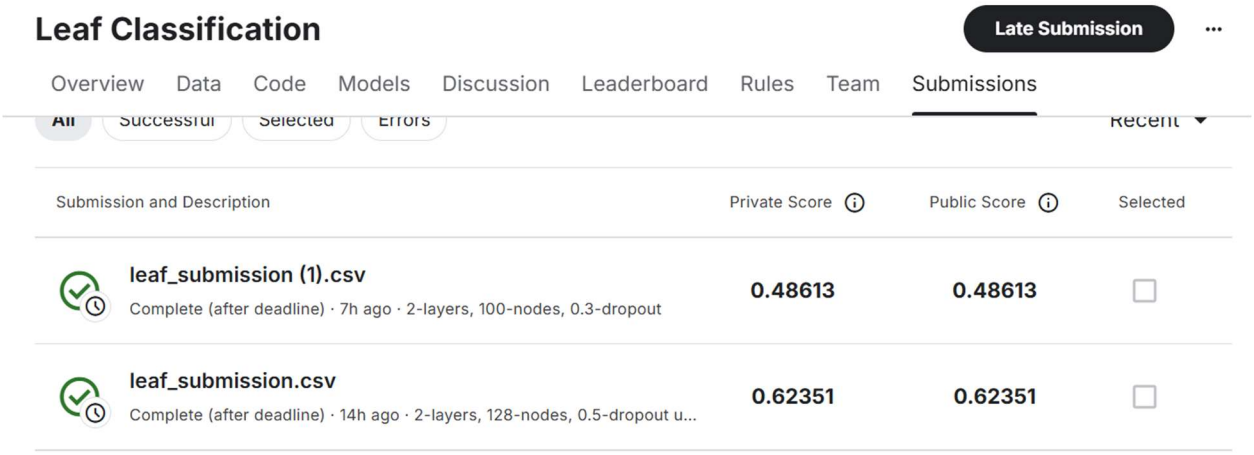
1. Deeper models with more layers and nodes did not always yield better performance; simpler architectures often performed just as well or better.
2. Dropout played a critical role in preventing overfitting—models with insufficient dropout showed high training accuracy but poor validation results.
3. Increasing the number of filters or nodes did not consistently improve validation accuracy, highlighting the importance of balanced model complexity.

Convolution Neural Networks (Part 3)

In the third phase of the project, I shifted from using tabular features to training convolutional neural networks (CNNs) directly on the raw image data. I built a tf.data pipeline to efficiently load and preprocess the leaf images, including normalization and batching. I then created a CNN model-building function that allowed flexible tuning of key architectural components such as the number of convolutional layers, filter sizes, and dropout rates. Using this framework, I trained several models with different hyperparameter configurations and tracked training accuracy, validation accuracy, loss, and training time for each.

Model performance was evaluated using a summary table that compared all runs. I observed that increasing the number of layers generally improved model performance, while excessive dropout led to underfitting. Next steps here would be to optimize the CNN

models further, due to computing constraints the models took quite some time to run. Models with 3-5 layers should be tested next and number of training epochs could be increased by 2-3x from current numbers. Models can be further generalized by randomly flipping the images, rotating the images and zooming in on the images.



Introduction

Part 1 Code (Data Prep) - https://colab.research.google.com/drive/1dN0y6HYqIjKFhq7pgYYhlCY2ecmYSoN6?usp=sharing

Part 2 Code (NN DoE) - https://colab.research.google.com/drive/1xpimK1T2i8VQ9clx9Jx69ZKy6m0844Vu?usp=sharing

Part 3 Code (CNN Models) - https://colab.research.google.com/drive/1XCXj9PZTnriaDO9sqpb-PucG1fMslN4B?usp=sharing

Data taken from - https://www.kaggle.com/competitions/leaf-classification/data

Ready data files

Import modules and load data

```
# Import modules
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import numpy as np
import statsmodels.formula.api as smf
import time

from sklearn import tree
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error, make_scorer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import cross_val_score, KFold
from sklearn.model_selection import cross_val_predict
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import confusion_matrix

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Figures inline and set visualization style
%matplotlib inline
sns.set()

# To ensure all columns are displayed when calling data
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)


df_train_original = pd.read_csv('train.csv')
df_test_original = pd.read_csv('test.csv')


df_train_original.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 990 entries, 0 to 989
Columns: 194 entries, id to texture64
dtypes: float64(192), int64(1), object(1)
memory usage: 1.5+ MB
```

```
df_train_original.head()
```

	id	species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margin11	marg
0	1	Acer_Opalus	0.007812	0.023438	0.023438	0.003906	0.011719	0.009766	0.027344	0.0	0.001953	0.033203	0.013672	0.01
1	2	Pterocarya_Stenoptera	0.005859	0.000000	0.031250	0.015625	0.025391	0.001953	0.019531	0.0	0.000000	0.007812	0.003906	0.02
2	3	Quercus_Hartwissiana	0.005859	0.009766	0.019531	0.007812	0.003906	0.005859	0.068359	0.0	0.000000	0.044922	0.007812	0.01
3	5	Tilia_Tomentosa	0.000000	0.003906	0.023438	0.005859	0.021484	0.019531	0.023438	0.0	0.013672	0.017578	0.001953	0.01
4	6	Quercus_Variabilis	0.005859	0.003906	0.048828	0.009766	0.013672	0.015625	0.005859	0.0	0.000000	0.005859	0.001953	0.04

```
df_test_original.head()
```

	id	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margin11	margin12	margin13	margi
0	4	0.019531	0.009766	0.078125	0.011719	0.003906	0.015625	0.005859	0.0	0.005859	0.023438	0.005859	0.021484	0.076172	0.001
1	7	0.007812	0.005859	0.064453	0.009766	0.003906	0.013672	0.007812	0.0	0.033203	0.023438	0.009766	0.019531	0.039062	0.027
2	9	0.000000	0.000000	0.001953	0.021484	0.041016	0.000000	0.023438	0.0	0.011719	0.005859	0.001953	0.021484	0.001953	0.019
3	12	0.000000	0.000000	0.009766	0.011719	0.017578	0.000000	0.003906	0.0	0.003906	0.001953	0.000000	0.029297	0.000000	0.039
4	13	0.001953	0.000000	0.015625	0.009766	0.039062	0.000000	0.009766	0.0	0.005859	0.000000	0.001953	0.033203	0.000000	0.003

```
#df_train_original.describe()
df_train_original.describe()
```

	id	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	
count	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	9
mean	799.595960	0.017412	0.028539	0.031988	0.023280	0.014264	0.038579	0.019202	0.001083	0.007167	0.018639	
std	452.477568	0.019739	0.038855	0.025847	0.028411	0.018390	0.052030	0.017511	0.002743	0.008933	0.016071	
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	415.250000	0.001953	0.001953	0.013672	0.005859	0.001953	0.000000	0.005859	0.000000	0.001953	0.005859	
50%	802.500000	0.009766	0.011719	0.025391	0.013672	0.007812	0.015625	0.015625	0.000000	0.005859	0.015625	
75%	1195.500000	0.025391	0.041016	0.044922	0.029297	0.017578	0.056153	0.029297	0.000000	0.007812	0.027344	
max	1584.000000	0.087891	0.205080	0.156250	0.169920	0.111330	0.310550	0.091797	0.031250	0.076172	0.097656	

```
df_test_original.describe()
```

	id	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	
count	594.000000	594.000000	594.000000	594.000000	594.000000	594.000000	594.000000	594.000000	594.000000	594.000000	594.000000	5
mean	780.673401	0.017562	0.028425	0.031858	0.022556	0.014527	0.037497	0.019222	0.001085	0.007092	0.018798	
std	465.646977	0.019585	0.038351	0.025719	0.028797	0.018029	0.051372	0.017122	0.002697	0.009515	0.016229	
min	4.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	368.500000	0.001953	0.001953	0.013672	0.005859	0.001953	0.000000	0.005859	0.000000	0.001953	0.005859	
50%	774.000000	0.009766	0.010743	0.023438	0.013672	0.007812	0.013672	0.015625	0.000000	0.005859	0.015625	
75%	1184.500000	0.028809	0.041016	0.042969	0.027344	0.019531	0.056641	0.029297	0.000000	0.007812	0.027344	
max	1583.000000	0.085938	0.189450	0.167970	0.164060	0.093750	0.271480	0.087891	0.021484	0.083984	0.083984	

```
# Check missing data in df_train_original and df_test_original, output TRUE if missing data exists or FALSE if no missing data
print(df_train_original.isnull().values.any())
print(df_test_original.isnull().values.any())
```

```
False
False
```

```
# Count df_train_original species, organize A-Z
df_train_original['species'].value_counts().sort_index()
```

Show hidden output

Scaling variables

Since the goal is to use neural networks and each variable has a different min/max especially across margin vs shape vs texture, it makes more sense to implement standard scaling.

```
# Identify columns to scale
# Exclude 'id' and 'species' from training features
train_feature_cols = [col for col in df_train_original.columns if col not in ['id', 'species']]
# Exclude only 'id' from test features
test_feature_cols = [col for col in df_test_original.columns if col != 'id']

# Initialize and fit scaler on training data
scaler = StandardScaler()
scaler.fit(df_train_original[train_feature_cols])  # fit only on train to avoid leakage

# Create scaled train dataset
df_train_scaled = df_train_original.copy()
df_train_scaled[train_feature_cols] = scaler.transform(df_train_original[train_feature_cols])

# Create scaled test dataset
df_test_scaled = df_test_original.copy()
df_test_scaled[test_feature_cols] = scaler.transform(df_test_original[test_feature_cols])
```

Create merged dataset for EDA

```
# Create merged dataset for EDA but first add a variable called train to indicate test or train data
df_train_scaled = df_train_scaled.copy()
df_train_scaled['train'] = 1

df_test_scaled = df_test_scaled.copy()
df_test_scaled['train'] = 0

# Create merged dataset for EDA
df_merged = pd.concat([df_train_scaled, df_test_scaled], axis=0).reset_index(drop=True)
```

Creating dummy variables

```
# Organize df_merged by "id"
df_merged = df_merged.sort_values(by=['id'])
df_merged.head()
```

	id	species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margi
0	1	Acer_Opalus	-0.486611	-0.131357	-0.330956	-0.682244	-0.138444	-0.554066	0.465218	-0.395064	-0.584019	0.906636	-0.404
1	2	Pterocarya_Stenoptera	-0.585602	-0.734880	-0.028561	-0.269558	0.605389	-0.704306	0.018812	-0.395064	-0.802765	-0.674054	-0.778
2	3	Quercus_Hartwissiana	-0.585602	-0.483408	-0.482192	-0.544694	-0.563515	-0.629196	2.808662	-0.395064	-0.802765	1.636190	-0.628
990	4	NaN	0.107387	-0.483408	1.785924	-0.407108	-0.563515	-0.441400	-0.762355	-0.395064	-0.146528	0.298726	-0.703
3	5	Tilia_Tomentosa	-0.882575	-0.634302	-0.330956	-0.613469	0.392826	-0.366289	0.242044	-0.395064	0.728566	-0.066082	-0.853

```
# Create dummy variables for each species
df_merged = pd.get_dummies(df_merged, columns=['species'])

# Get a list of the new dummy column names
dummy_cols = [col for col in df_merged.columns if col.startswith('species_')]

# Create a dictionary to map old names to new names (without the prefix)
rename_dict = {col: col.replace('species_', '') for col in dummy_cols}

# Rename the columns
df_merged = df_merged.rename(columns=rename_dict)
```

```
# Convert dummy columns from bool to float
df_merged[list(rename_dict.values())] = df_merged[list(rename_dict.values())].astype(float)
```

```
df_merged.describe()
```

	id	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	m
count	1584.000000	1584.000000	1584.000000	1584.000000	1584.000000	1584.000000	1584.000000	1584.000000	1584.000000	1584.000000	158
mean	792.500000	0.002837	-0.001099	-0.001880	-0.009552	0.005366	-0.007801	0.000437	0.000270	-0.003149	(
std	457.405728	0.997273	0.995345	0.998334	1.005373	0.992910	0.995525	0.991925	0.993937	1.025130	1
min	1.000000	-0.882575	-0.734880	-1.238218	-0.819794	-0.776023	-0.741862	-1.097116	-0.395064	-0.802765	-1
25%	396.750000	-0.783584	-0.684591	-0.708988	-0.613469	-0.669769	-0.741862	-0.762355	-0.395064	-0.584019	-(
50%	792.500000	-0.387569	-0.433119	-0.330956	-0.338333	-0.351007	-0.478955	-0.204362	-0.395064	-0.146528	-(
75%	1188.250000	0.503402	0.321273	0.500668	0.211903	0.286572	0.347320	0.576805	-0.395064	0.072217	(
max	1584.000000	3.572332	4.545885	5.263734	5.163955	5.280951	5.229881	4.147822	11.004131	8.603852	4

```
df_merged.head()
```

	id	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margin11	margin12	margir
0	1	-0.486611	-0.131357	-0.330956	-0.682244	-0.138444	-0.554066	0.465218	-0.395064	-0.584019	0.906636	-0.404139	0.644338	0.543(
1	2	-0.585602	-0.734880	-0.028561	-0.269558	0.605389	-0.704306	0.018812	-0.395064	-0.802765	-0.674054	-0.778711	1.310607	-0.384(
2	3	-0.585602	-0.483408	-0.482192	-0.544694	-0.563515	-0.629196	2.808662	-0.395064	-0.802765	1.636190	-0.628898	-0.021845	-0.426(
990	4	0.107387	-0.483408	1.785924	-0.407108	-0.563515	-0.441400	-0.762355	-0.395064	-0.146528	0.298726	-0.703804	0.810884	0.753(
3	5	-0.882575	-0.634302	-0.330956	-0.613469	0.392826	-0.366289	0.242044	-0.395064	0.728566	-0.066082	-0.853618	0.644338	-0.848(

Check leakage

Ensure all dummy variables for test set add to 0

```
# Identify all species dummy columns
species_cols = [col for col in df_merged.columns if col not in ['id', 'train']
                and not col.startswith(('margin', 'shape', 'texture'))]

# Check sum of dummy columns row-wise
row_sums = df_merged[df_merged['train'] == 0][species_cols].sum(axis=1)

# Count how many rows are non-zero (should be 0)
leakage_rows = (row_sums != 0).sum()

print(f"Number of test rows with non-zero species dummy values: {leakage_rows}")
```

```
Number of test rows with non-zero species dummy values: 0
```

```
# Check sum of dummy columns row-wise
row_sums = df_merged[df_merged['train'] == 1][species_cols].sum(axis=1)

# Count how many rows are non-zero (should be 990)
leakage_rows = (row_sums != 0).sum()

print(f"Number of train rows with non-zero species dummy values: {leakage_rows}")
```

```
Number of train rows with non-zero species dummy values: 990
```

Split test and train data

```
df_train = df_merged[df_merged['train'] == 1].copy()
df_test = df_merged[df_merged['train'] == 0].copy()

# Remove variable train
df_train = df_train.drop(columns=['train'])
df_test = df_test.drop(columns=['train'])
```

```
# Export df_train as csv
df_train.to_csv('df_train.csv', index=False)

# Export df_test as csv
df_test.to_csv('df_test.csv', index=False)
```

Introduction

Part 1 Code (Data Prep) - https://colab.research.google.com/drive/1dN0y6HYqIjKFhq7pgYYhlCY2ecmYSoN6?usp=sharing

Part 2 Code (NN DoE) - https://colab.research.google.com/drive/1xpimK1T2i8VQ9clx9Jx69ZKy6m0844Vu?usp=sharing

Part 3 Code (CNN Models) - https://colab.research.google.com/drive/1XCXj9PZTnriaDO9sqpb-PucG1fMslN4B?usp=sharing

Data taken from - https://www.kaggle.com/competitions/leaf-classification/data

Ready data files

Import modules and load data

[ ] ↳ 4 cells hidden

Group variables

```
# Use df_train to define column groups
all_columns = df_train.columns

# Feature groups
margin_vars = [col for col in all_columns if col.startswith("margin")]
shape_vars = [col for col in all_columns if col.startswith("shape")]
texture_vars = [col for col in all_columns if col.startswith("texture")]

# Class labels: one-hot encoded columns for species (exclude id and features)
leaf_classes = [col for col in all_columns if col not in ['id'] + margin_vars + shape_vars +


print(f"# margin_vars: {len(margin_vars)}")
print(f"# shape_vars: {len(shape_vars)}")
print(f"# texture_vars: {len(texture_vars)}")
print(f"# leaf_classes: {len(leaf_classes)}")
```

```
# margin_vars: 64
# shape_vars: 64
# texture_vars: 64
# leaf_classes: 99
```

Neural Net DoE

Prep data for DoE

```
# Set of input features
input_vars = margin_vars + shape_vars + texture_vars

# Create input and output arrays
X = df_train[input_vars].values
y = df_train[leaf_classes].values  # Already one-hot encoded

# Train/validation split
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42, stra

# Count check
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_val shape:", X_val.shape)
print("y_val shape:", y_val.shape)
```

```
X_train shape: (792, 192)
y_train shape: (792, 99)
X_val shape: (198, 192)
y_val shape: (198, 99)
```

DoE grid search

```
# Design of Experiments grid
layers_list = [2, 5]
nodes_list = [50, 100]

results = []
best_model = None
best_val_accuracy = 0

# Loop through grid
for layers in layers_list:
    for nodes in nodes_list:
        print(f"\n Training model: {layers} layers × {nodes} nodes/layer")

        # Build model
        model = Sequential()
        model.add(Flatten(input_shape=(X_train.shape[1],)))

        for _ in range(layers):
            model.add(Dense(nodes, activation='relu'))
```

```
model.add(Dense(y_train.shape[1], activation='softmax'))

model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'

start_time = time.time()
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=30,
    batch_size=128,
    verbose=0
)
end_time = time.time()
training_time = end_time - start_time

# Plot accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.title(f'Accuracy: {layers} Layers × {nodes} Nodes')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# Plot loss
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title(f'Loss: {layers} Layers × {nodes} Nodes')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()

# Evaluate model
val_loss, val_accuracy = model.evaluate(X_val, y_val, verbose=0)
train_accuracy = history.history['accuracy'][-1]
train_loss = history.history['loss'][-1]

results.append({
    'layers': layers,
    'nodes': nodes,
    'train_accuracy': round(train_accuracy, 4),
    'train_loss': round(train_loss, 4),
    'val_accuracy': round(val_accuracy, 4),
    'val_loss': round(val_loss, 4),
    'training_time_sec': round(training_time, 2)
})

if val_accuracy > best_val_accuracy:
```

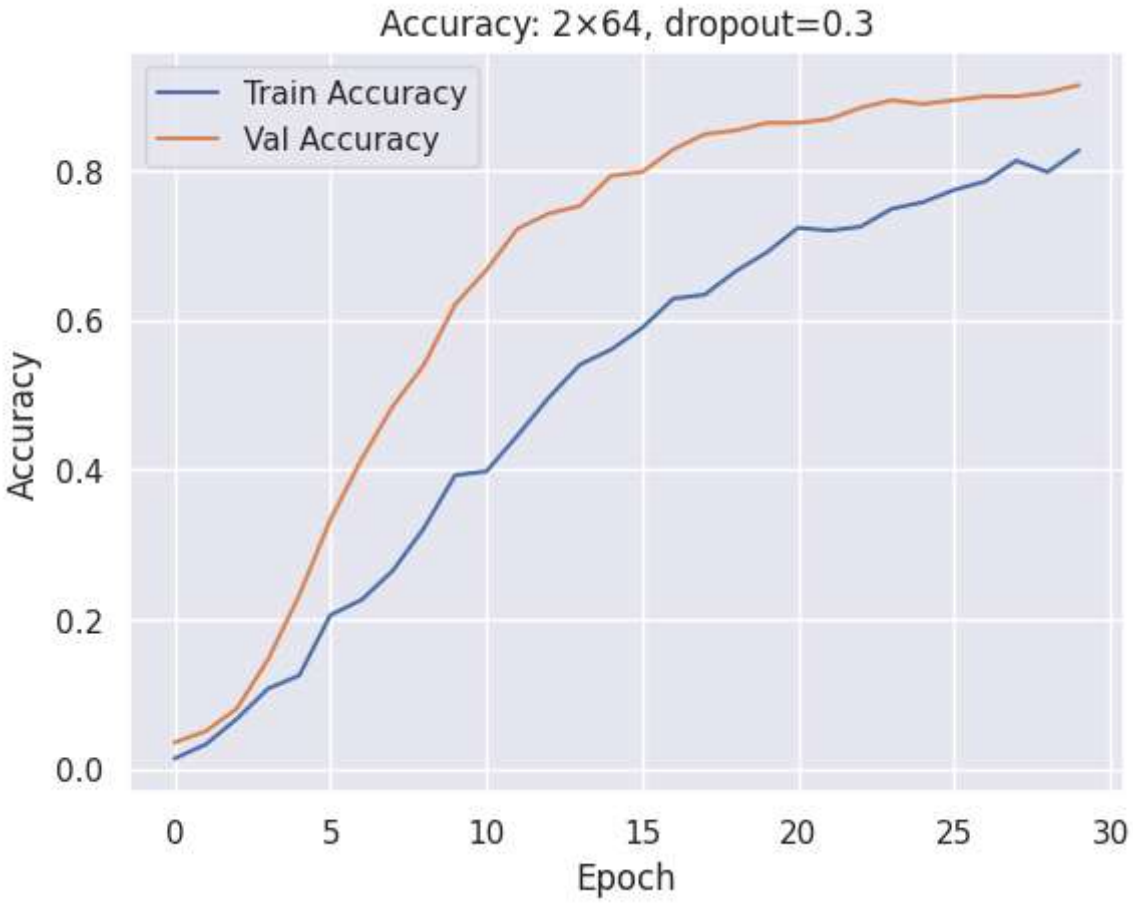
```
            best_val_accuracy = val_accuracy
            best_model = model

# Summary table
df_results = pd.DataFrame(results)
print(df_results)
```

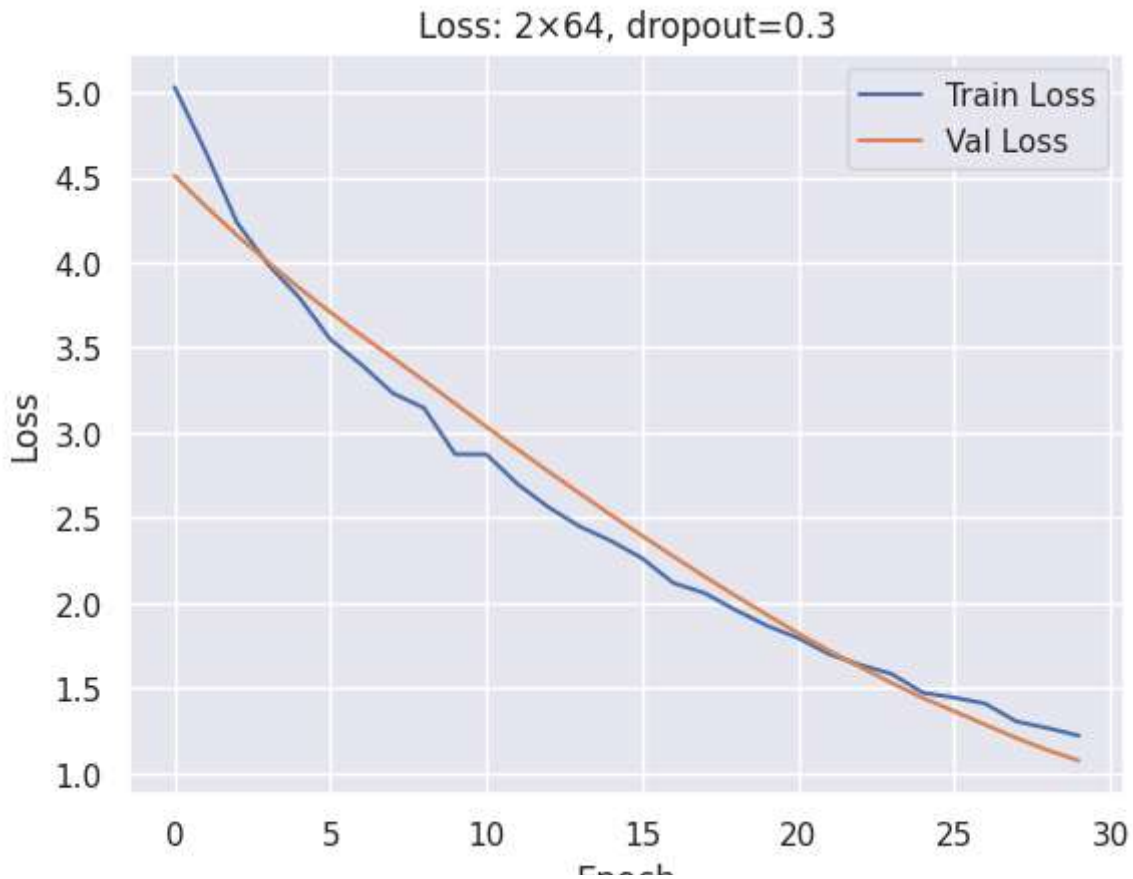
Show hidden output

```
# Define hyperparameter grid
layers_list = [2, 3]
nodes_list = [64, 100, 128]
dropout_rates = [0.3, 0.5]

results = []
best_model = None
best_val_accuracy = 0

# Loop through all combinations
for layers in layers_list:
    for nodes in nodes_list:
        for dropout_rate in dropout_rates:
            print(f"\n🔧 Training: {layers} layers × {nodes} nodes/layer × dropout={dropout

            # Build the model
            model = Sequential()
            model.add(Flatten(input_shape=(X_train.shape[1],)))

            for _ in range(layers):
                model.add(Dense(nodes, activation='relu'))
                model.add(BatchNormalization())
                model.add(Dropout(dropout_rate))

            model.add(Dense(y_train.shape[1], activation='softmax'))

            model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accur

            # Train the model
            start_time = time.time()
            history = model.fit(
                X_train, y_train,
                validation_data=(X_val, y_val),
                epochs=30,
                batch_size=128,
                verbose=0
            )
            end_time = time.time()
            training_time = round(end_time - start_time, 2)

            # Plot Accuracy
            plt.plot(history.history['accuracy'], label='Train Accuracy')
```

```
            plt.plot(history.history['val_accuracy'], label='Val Accuracy')
            plt.title(f'Accuracy: {layers}×{nodes}, dropout={dropout_rate}')
            plt.xlabel('Epoch')
            plt.ylabel('Accuracy')
            plt.legend()
            plt.grid(True)
            plt.show()

            # Plot Loss
            plt.plot(history.history['loss'], label='Train Loss')
            plt.plot(history.history['val_loss'], label='Val Loss')
            plt.title(f'Loss: {layers}×{nodes}, dropout={dropout_rate}')
            plt.xlabel('Epoch')
            plt.ylabel('Loss')
            plt.legend()
            plt.grid(True)
            plt.show()

            # Evaluate
            val_loss, val_accuracy = model.evaluate(X_val, y_val, verbose=0)
            train_accuracy = history.history['accuracy'][-1]
            train_loss = history.history['loss'][-1]

            results.append({
                'layers': layers,
                'nodes': nodes,
                'dropout': dropout_rate,
                'train_accuracy': round(train_accuracy, 4),
                'train_loss': round(train_loss, 4),
                'val_accuracy': round(val_accuracy, 4),
                'val_loss': round(val_loss, 4),
                'training_time_sec': training_time
            })

            if val_accuracy > best_val_accuracy:
                best_val_accuracy = val_accuracy
                best_model = model

# Summarize all runs
df_results = pd.DataFrame(results)
print("\n Final Results:")
print(df_results)
```

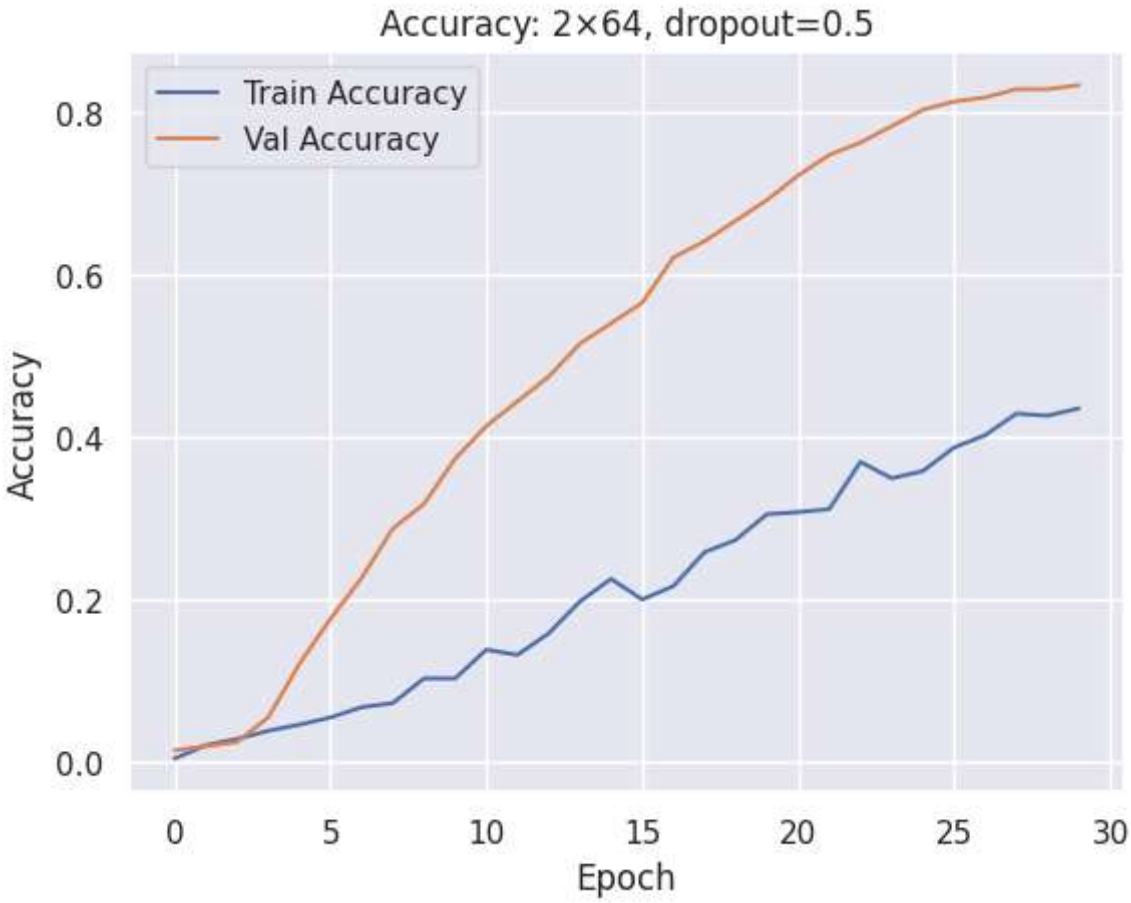
```
🔧 Training: 2 layers × 64 nodes/layer × dropout=0.3
/usr/local/lib/python3.11/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWa
  super().__init__(**kwargs)
```

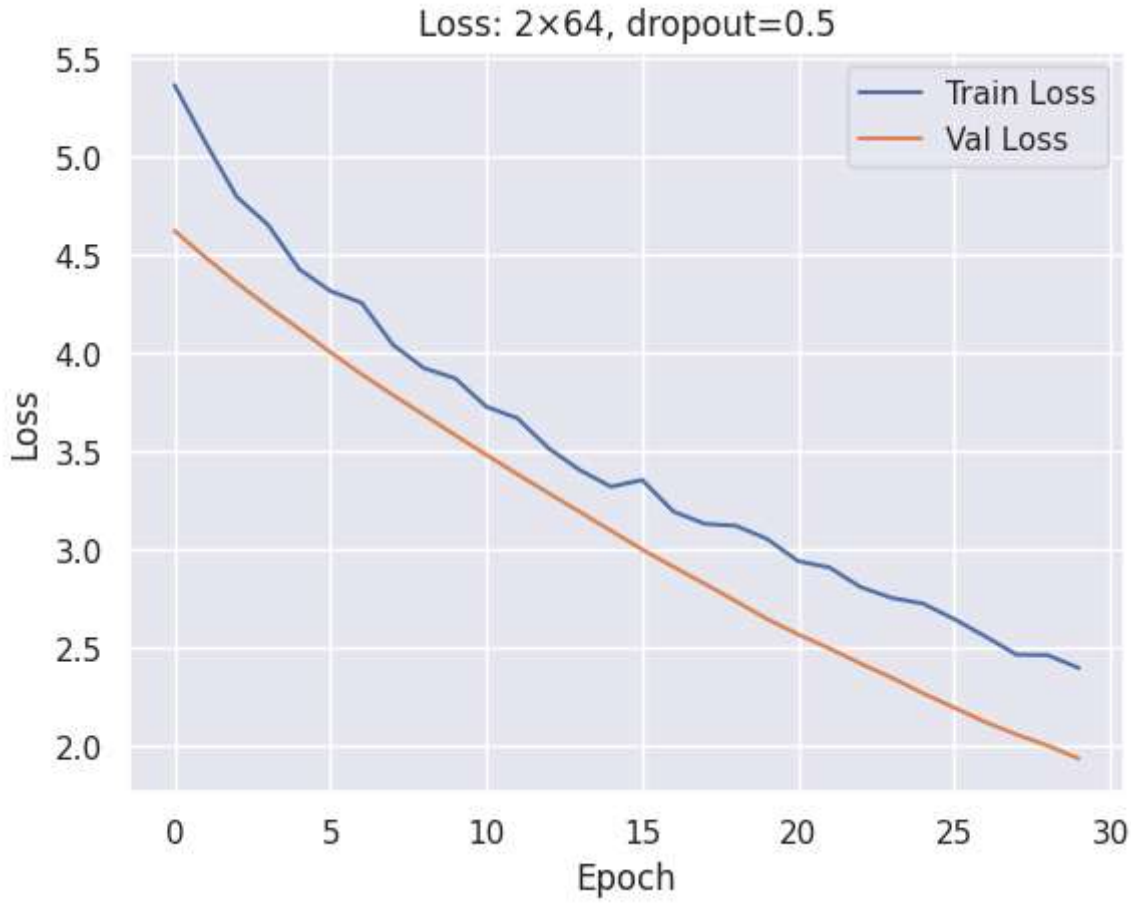




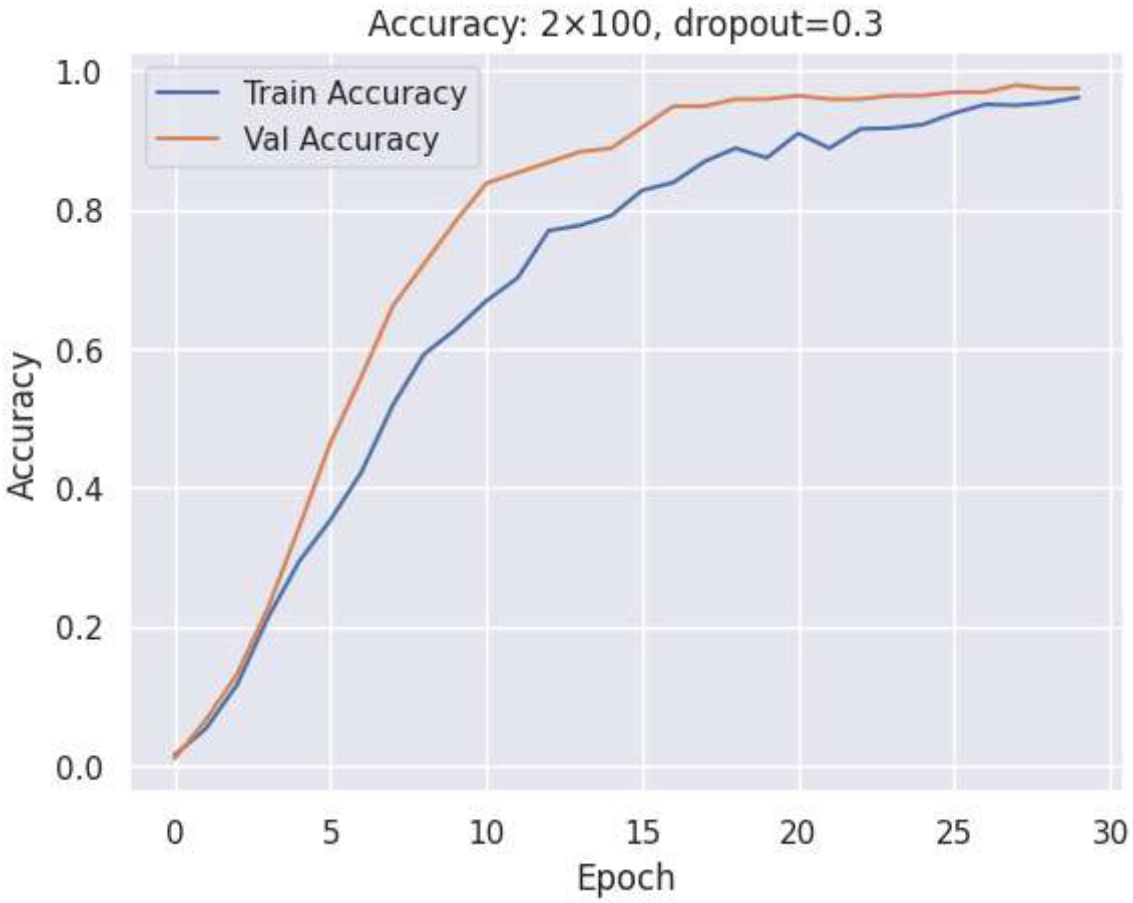
Epoch

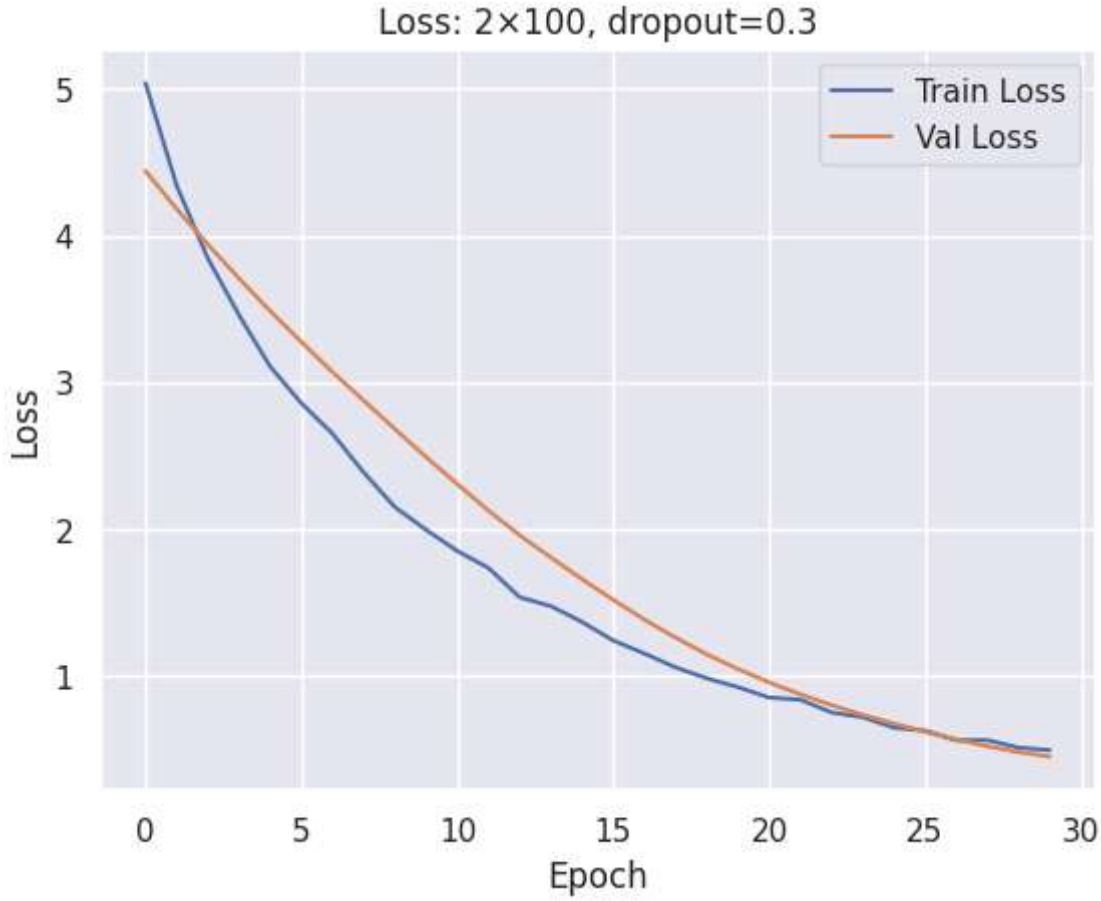
🔧 `Training: 2 layers × 64 nodes/layer × dropout=0.5`



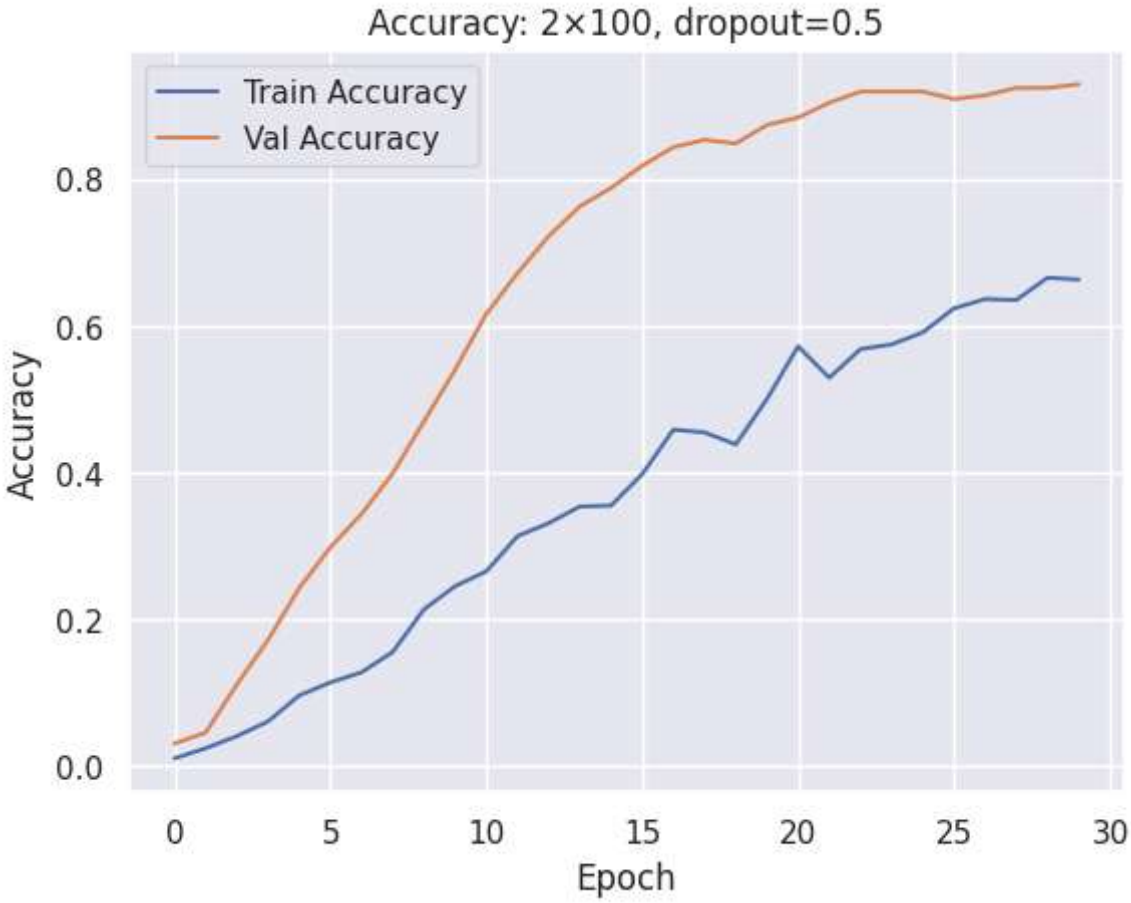


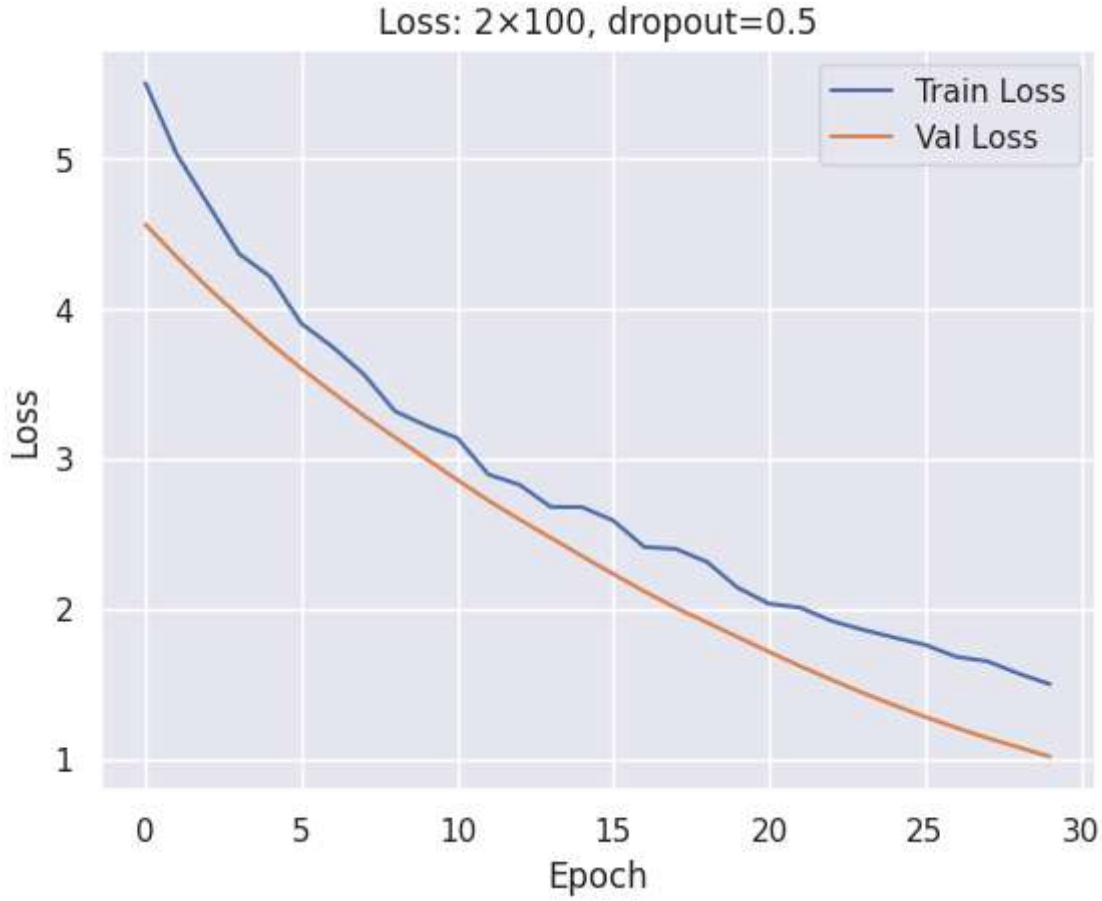
🔧 Training: 2 layers × 100 nodes/layer × dropout=0.3



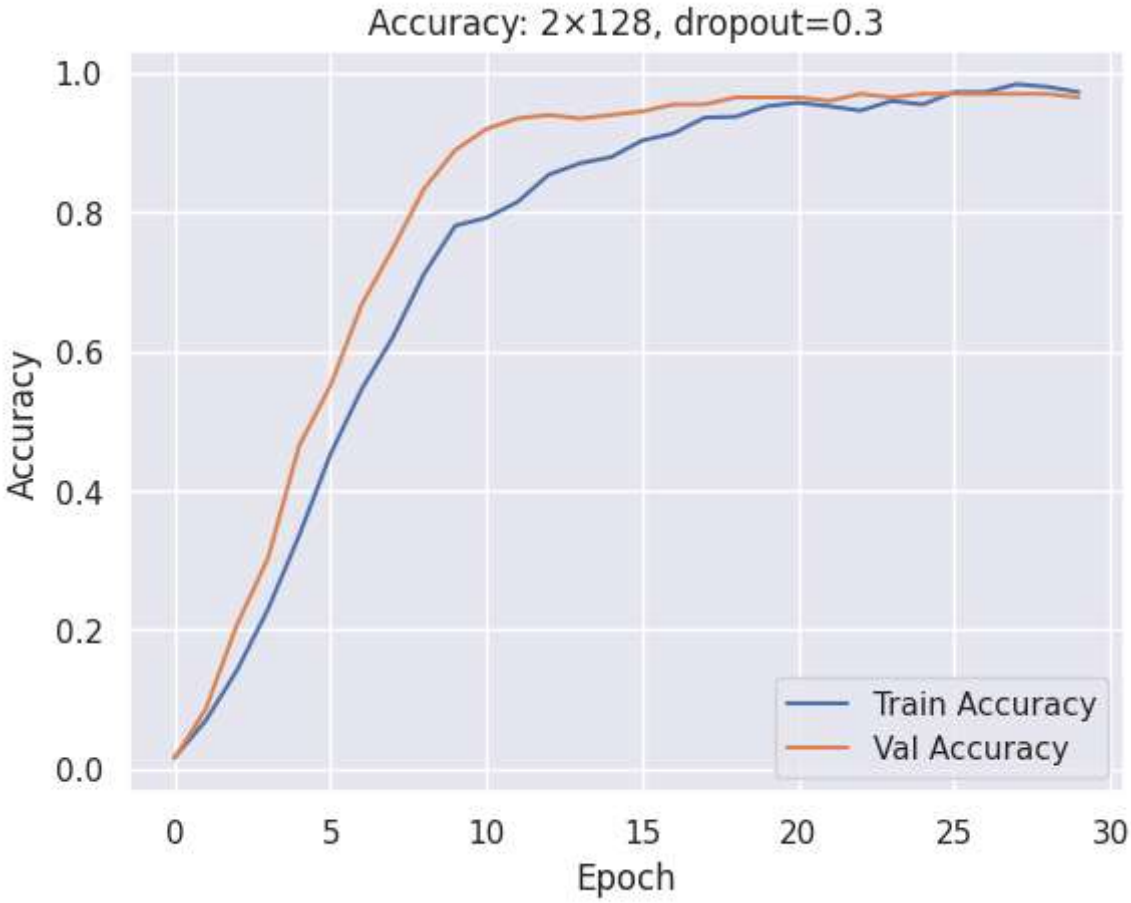


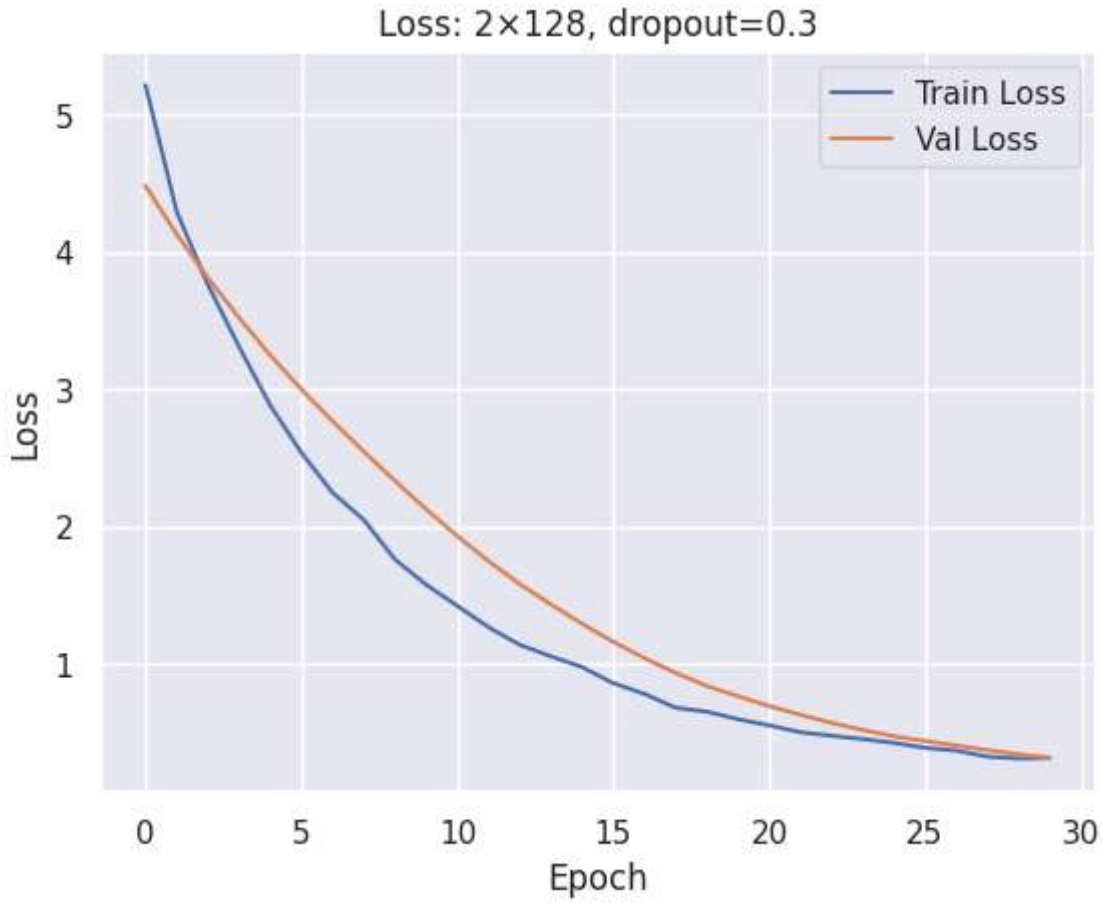
🔧 Training: 2 layers × 100 nodes/layer × dropout=0.5





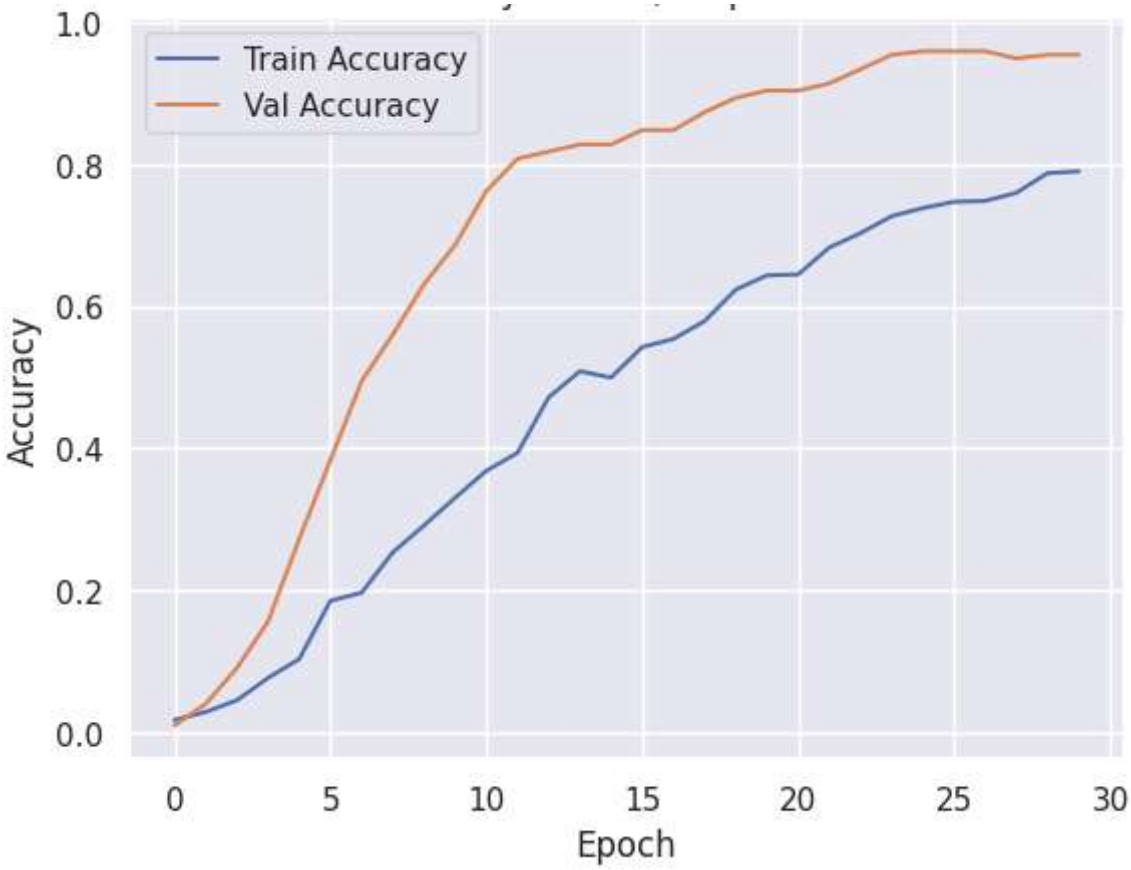
🔧 Training: 2 layers × 128 nodes/layer × dropout=0.3



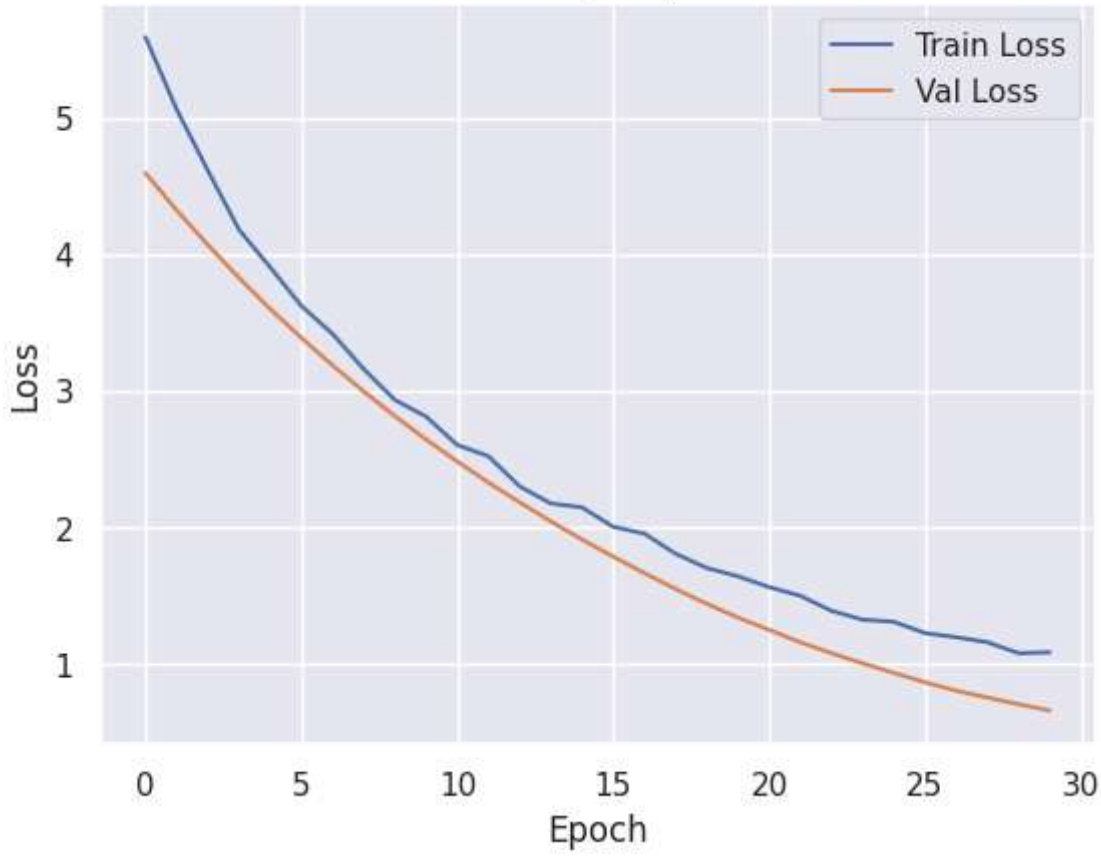


🔧 `Training: 2 layers × 128 nodes/layer × dropout=0.5`

Accuracy: 2×128, dropout=0.5

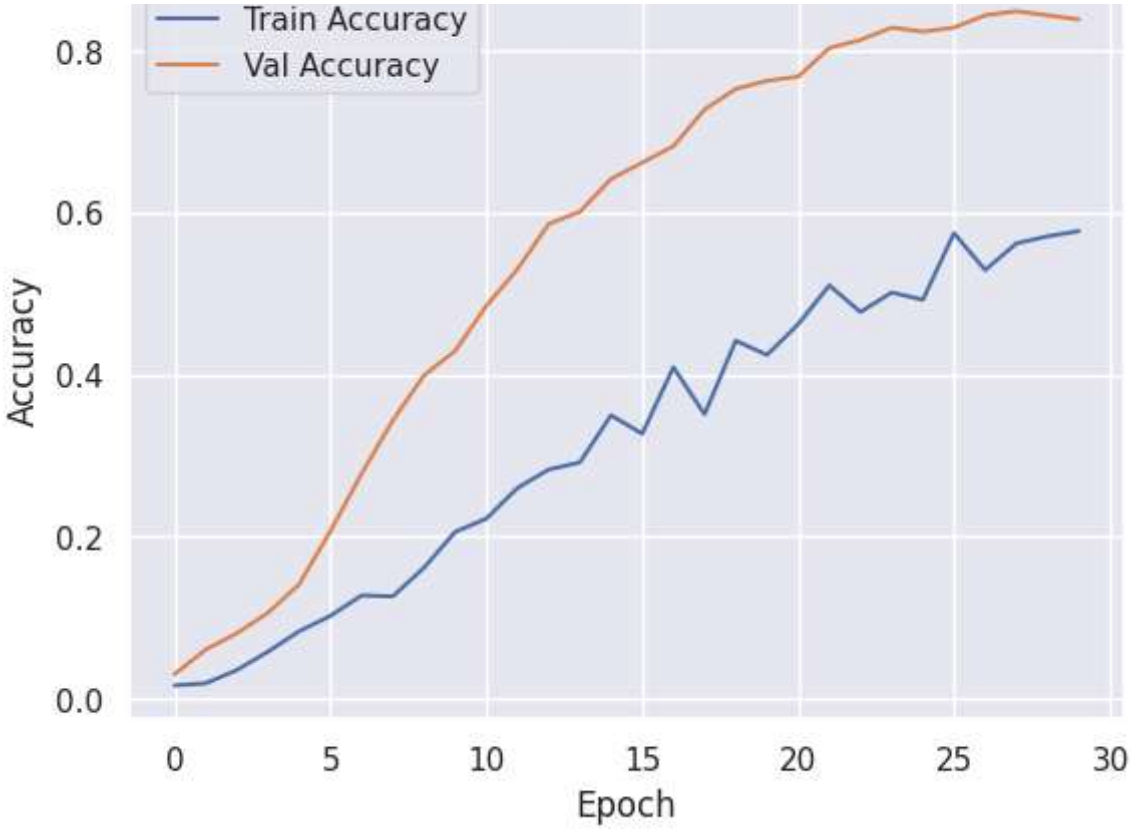


Loss: 2×128, dropout=0.5

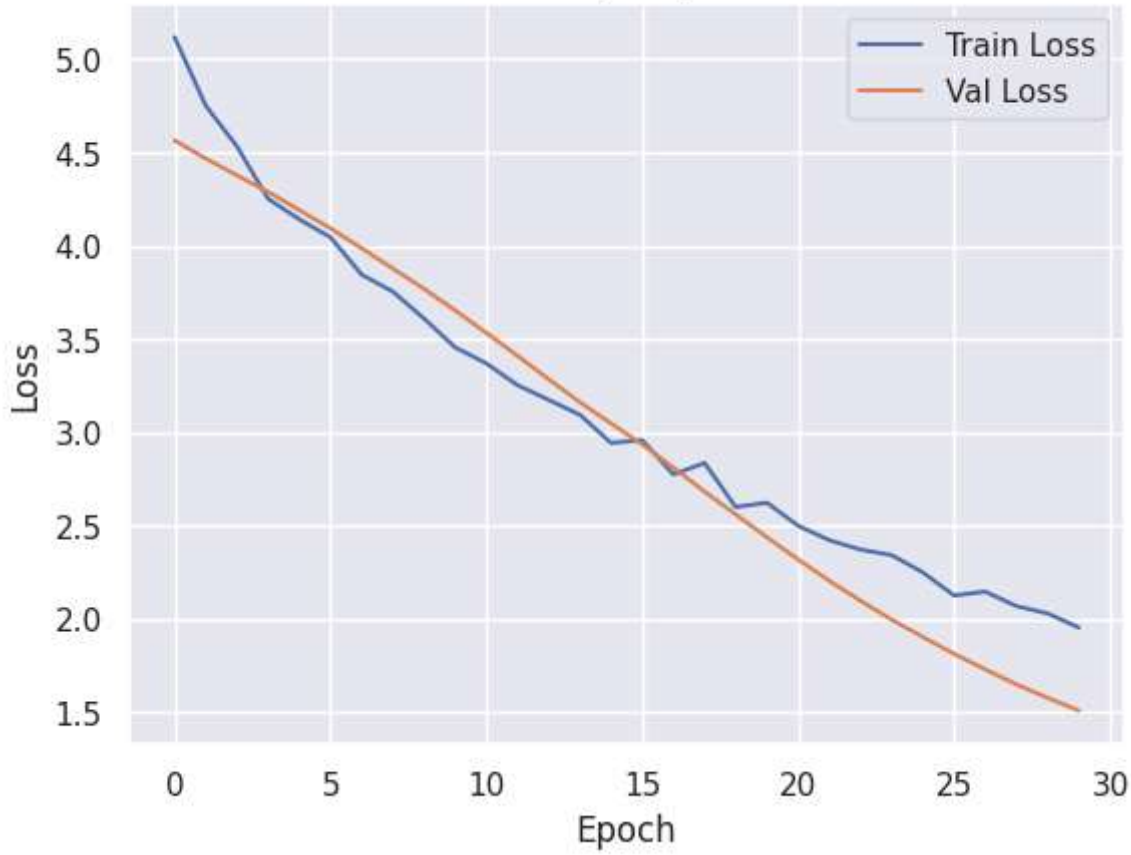


🔧 `Training: 3 layers × 64 nodes/layer × dropout=0.3`

Accuracy: 3×64, dropout=0.3

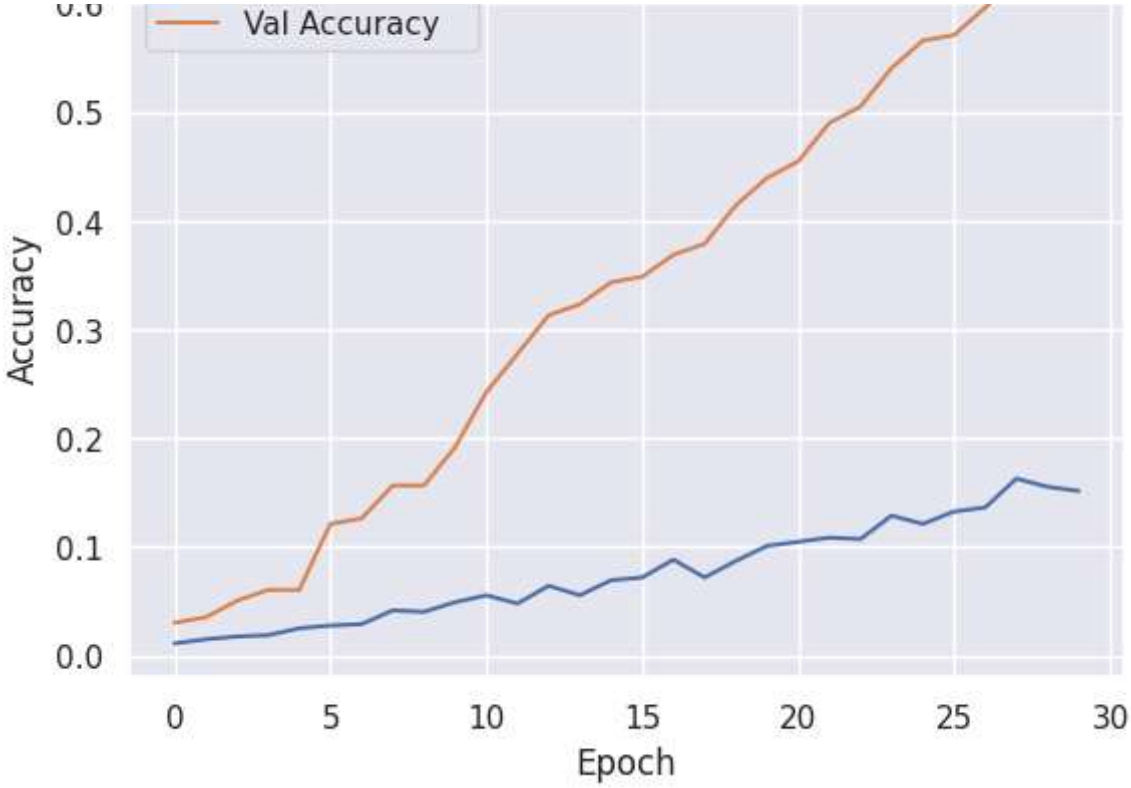


Loss: 3×64, dropout=0.3

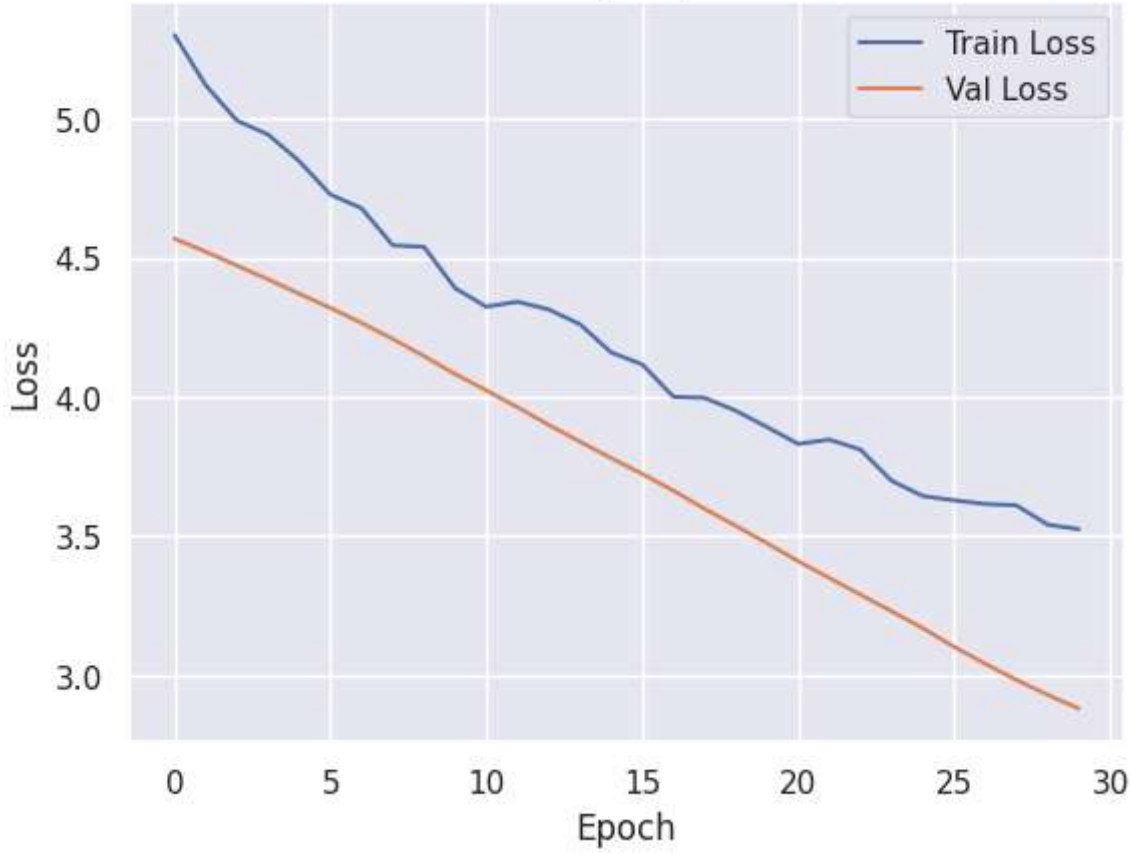


🔧 Training: 3 layers × 64 nodes/layer × dropout=0.5

Accuracy: 3×64, dropout=0.5

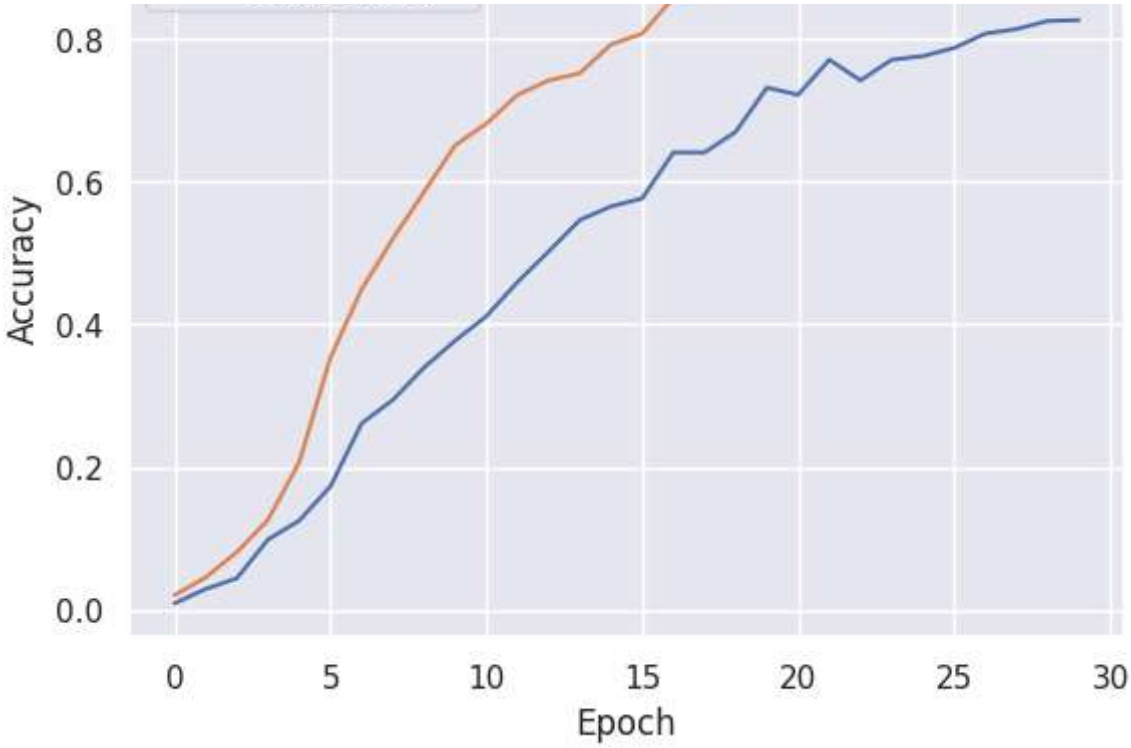


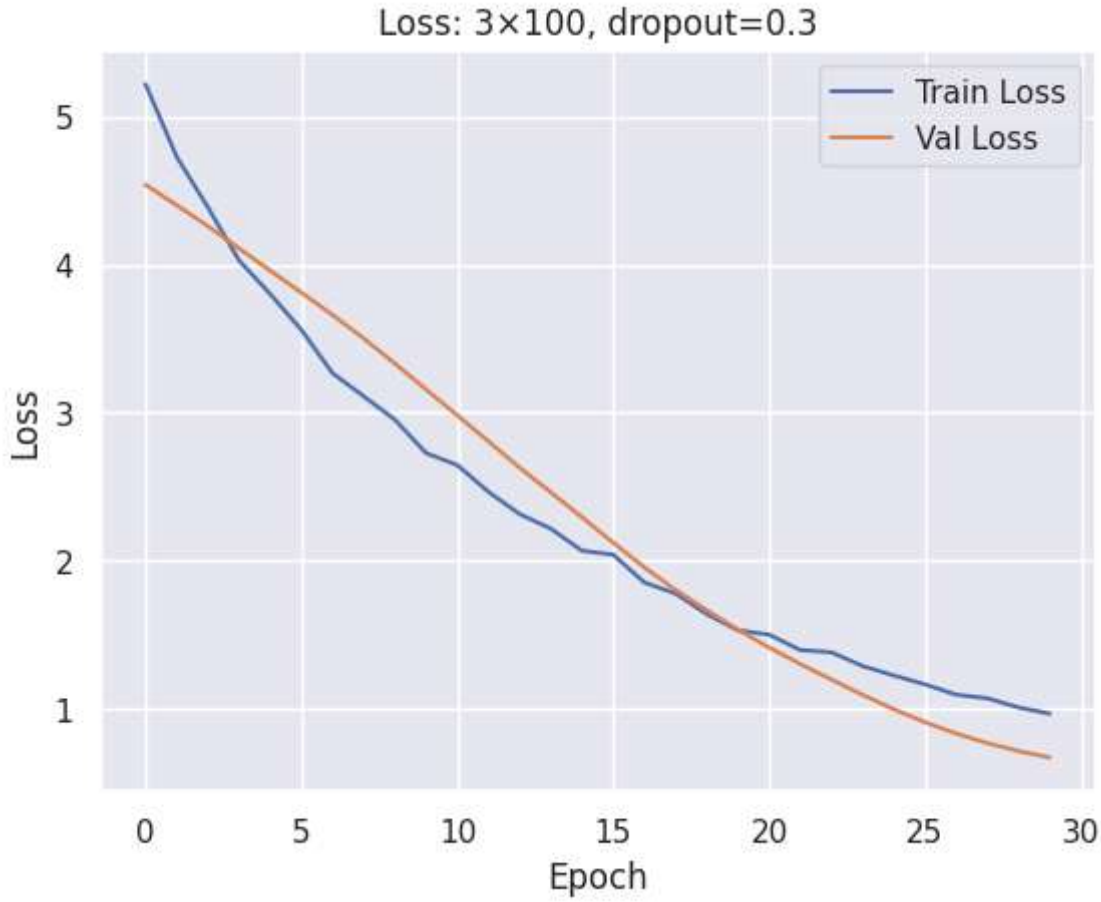
Loss: 3×64, dropout=0.5



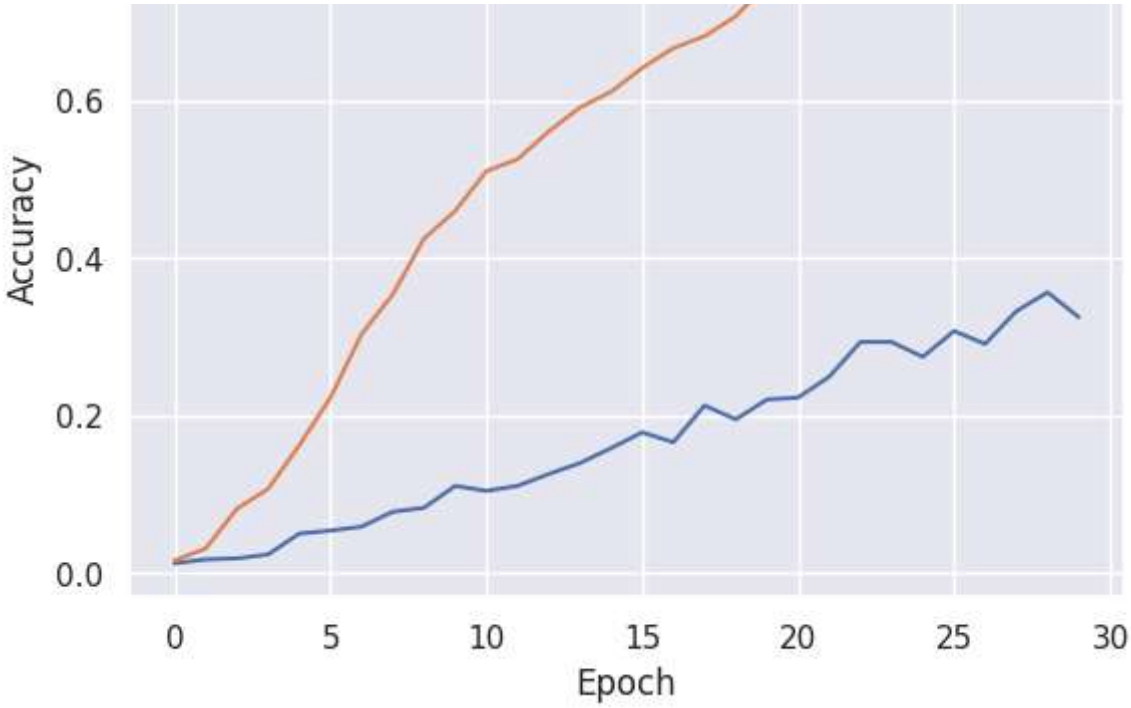
🔧 `Training: 3 layers × 100 nodes/layer × dropout=0.3`

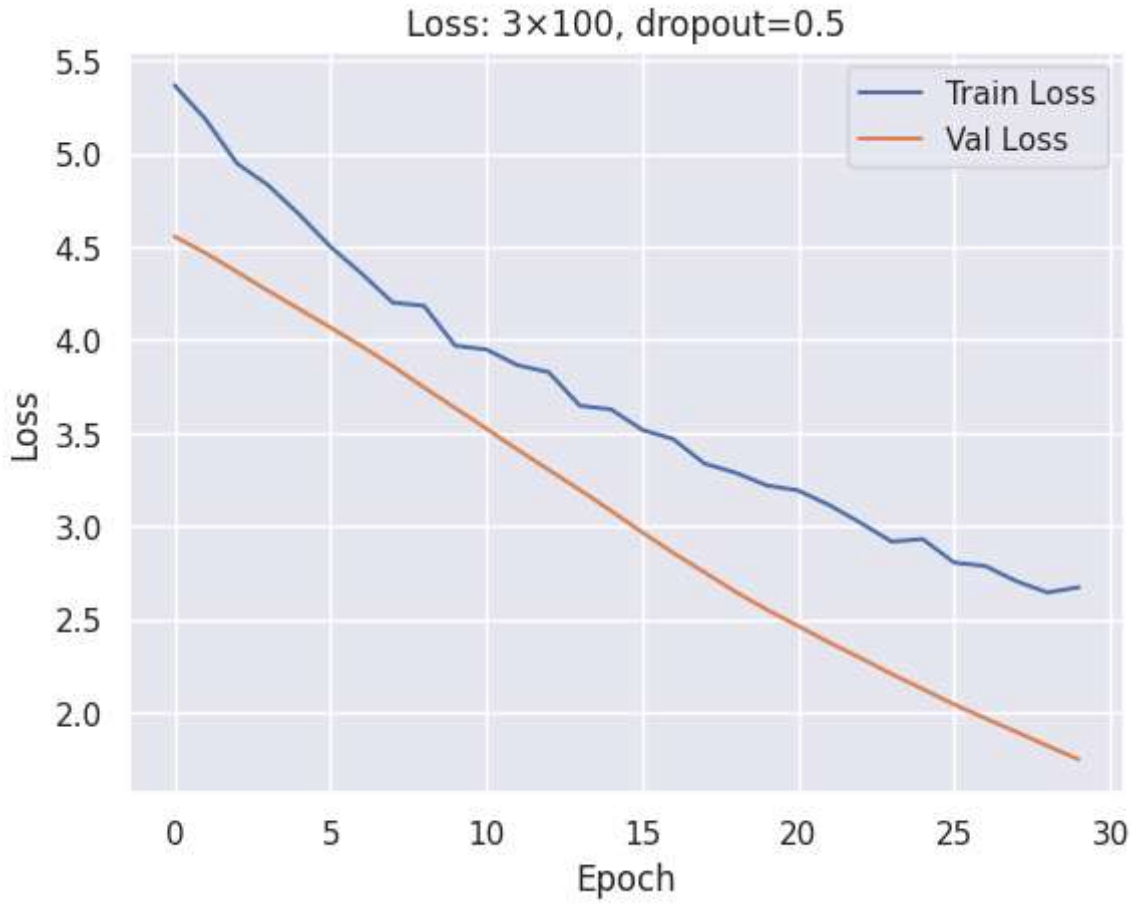
Accuracy: 3×100, dropout=0.3



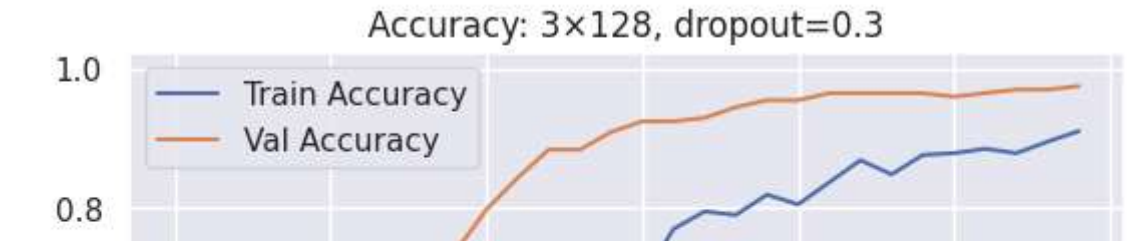


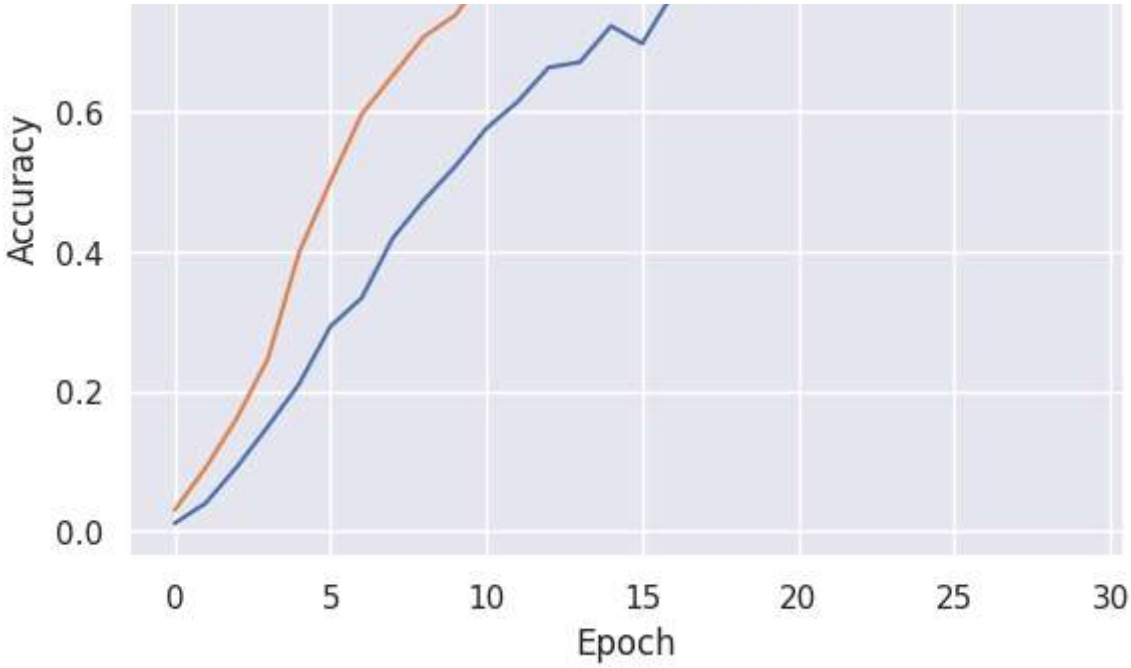
🔧 `Training: 3 layers × 100 nodes/layer × dropout=0.5`



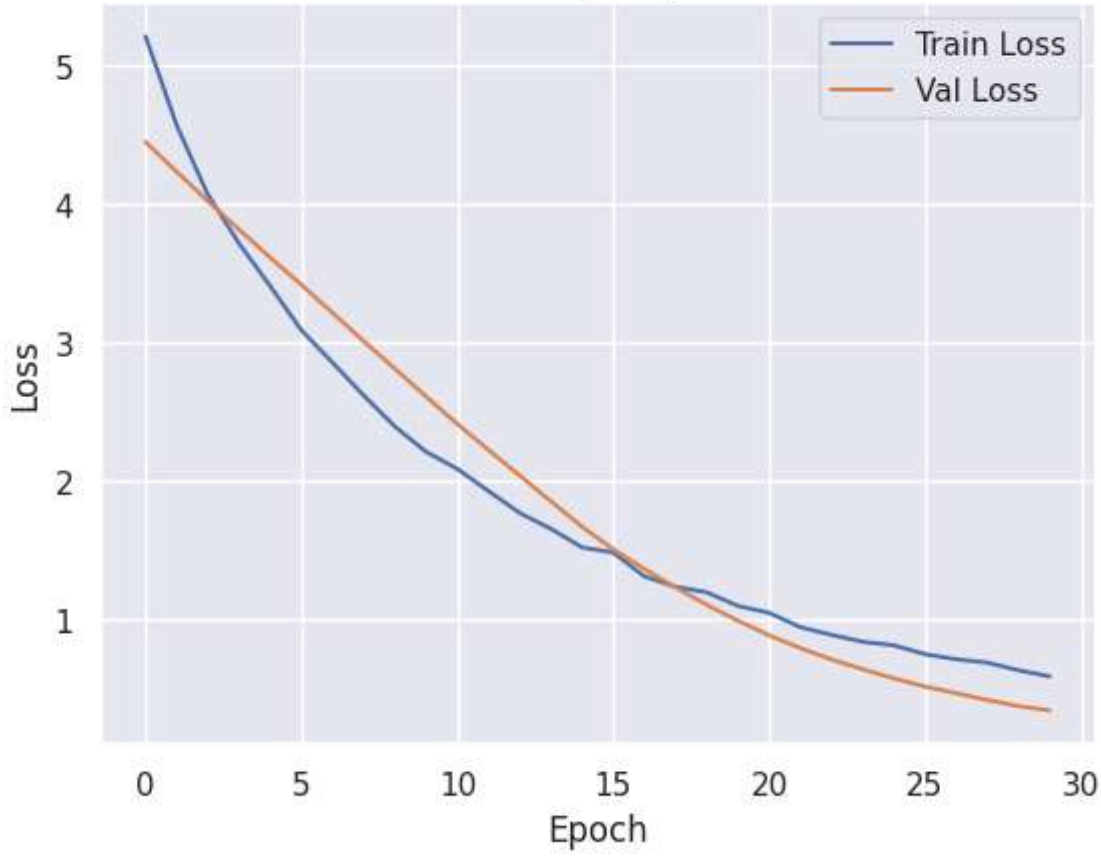


🔧 `Training: 3 layers × 128 nodes/layer × dropout=0.3`



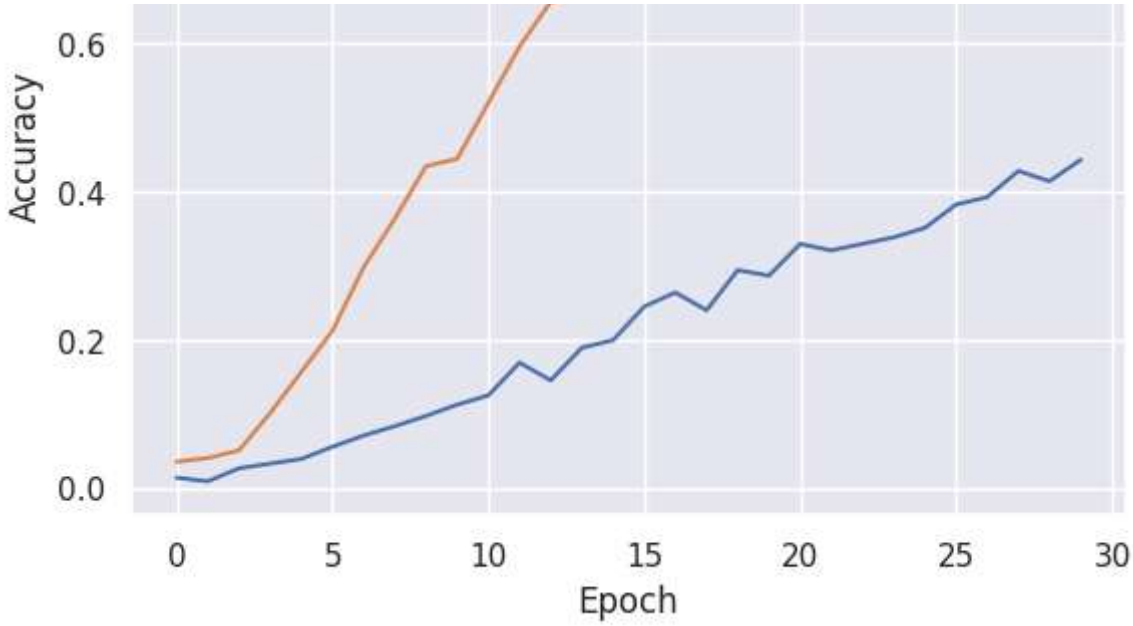


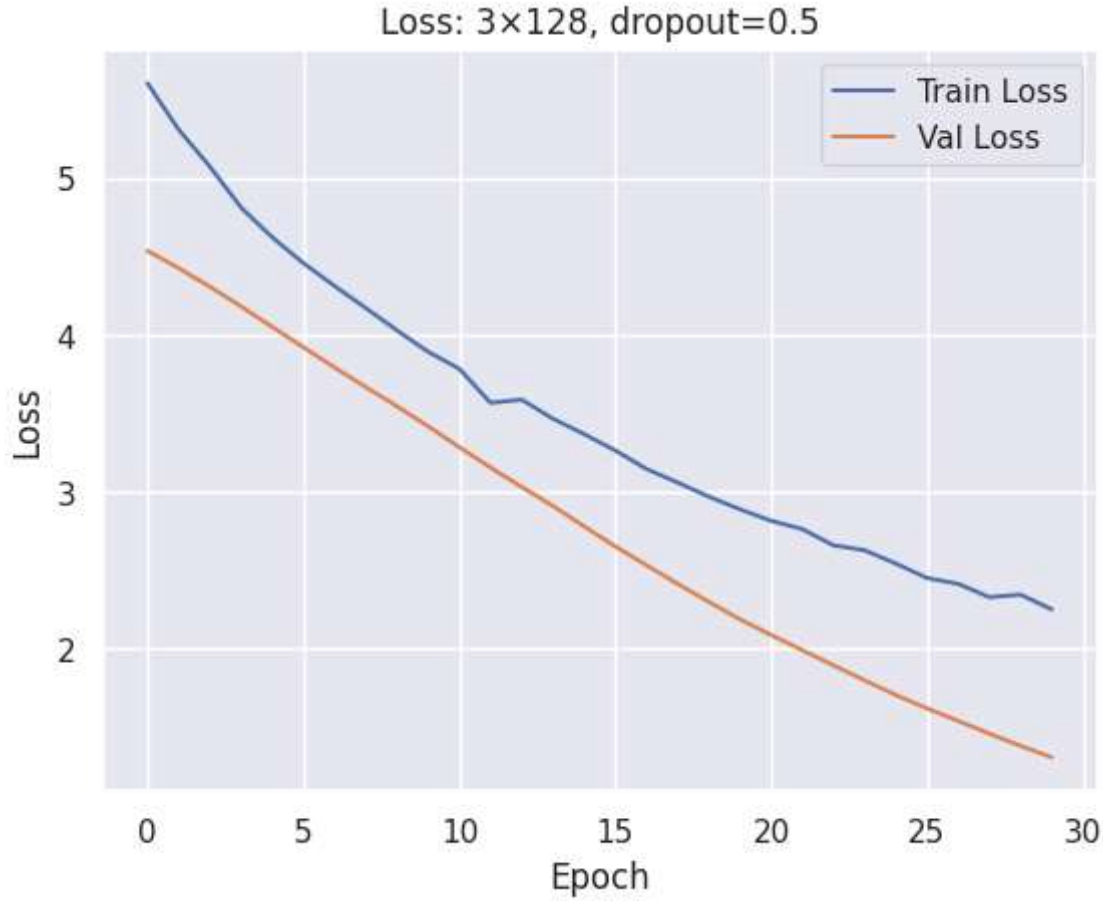
Loss: 3×128, dropout=0.3



🔧 `Training: 3 layers × 128 nodes/layer × dropout=0.5`

Accuracy: 3×128, dropout=0.5





Final Results:

	layers	nodes	dropout	train_accuracy	train_loss	val_accuracy	\
0	2	64	0.3	0.8270	1.2221	0.9141	
1	2	64	0.5	0.4356	2.4000	0.8333	
2	2	100	0.3	0.9621	0.4945	0.9747	
3	2	100	0.5	0.6629	1.4979	0.9293	
4	2	128	0.3	0.9722	0.3103	0.9646	
5	2	128	0.5	0.7904	1.0819	0.9545	
6	3	64	0.3	0.5770	1.9540	0.8384	
7	3	64	0.5	0.1515	3.5251	0.6313	
8	3	100	0.3	0.8270	0.9684	0.9394	

9	3	100	0.5	0.3245	2.6726	0.8687
10	3	128	0.3	0.9104	0.5935	0.9747
11	3	128	0.5	0.4432	2.2526	0.9192

	val_loss	training_time_sec
0	1.0759	6.73
1	1.9390	5.51
2	0.4513	6.95
3	1.0159	7.99
4	0.3121	6.09
5	0.6553	6.79
6	1.5095	6.53
7	2.8803	8.11
8	0.6726	7.80
9	1.7486	8.83
10	0.3482	7.72
11	1.3105	7.81

Grid Search Takeaways

First attempt: layers = [2, 5], nodes = [10, 20] and epochs = 10

- All models were underperforming with the best model on reaching ~13% validation accuracy.
- Models were too simply for the task 20 nodes for distinguishing 99 classes
- 10 epochs is too few

Second attempt: layers = [2, 5], nodes = [50, 100] and epochs = 30

- Models are significantly better, 2 layers and 100 nodes is the best model with training accuracy of 100% and validation accuracy of 96%.
- Increasing from 2 layers to 5 layers does not improve validation accuracy and actually decreases it showing signs on overfitting.

Third attempt: layers = [2, 3], nodes = [64, 100, 128], dropout_rate = [0.3, 0.5], enable EarlyStopping and epochs = 30

- Dropout sacrifices train_accuracy to improve val_accuracy by improving generalization
- 2-layer models consistently outperform 3-layer models
- Best model here is 2-layers, 100-nodes, 0.3-dropout

```
# Show the best-performing configuration
best_row = df_results[df_results['val_accuracy'] == df_results['val_accuracy'].max()]
print("Best Model Configuration Based on Validation Accuracy:")
print(best_row)
```

```
Best Model Configuration Based on Validation Accuracy:
    layers  nodes  dropout  train_accuracy  train_loss  val_accuracy  \
2        2    100      0.3          0.9621      0.4945        0.9747
10       3    128      0.3          0.9104      0.5935        0.9747

    val_loss  training_time_sec
2     0.4513               6.95
10    0.3482               7.72
```

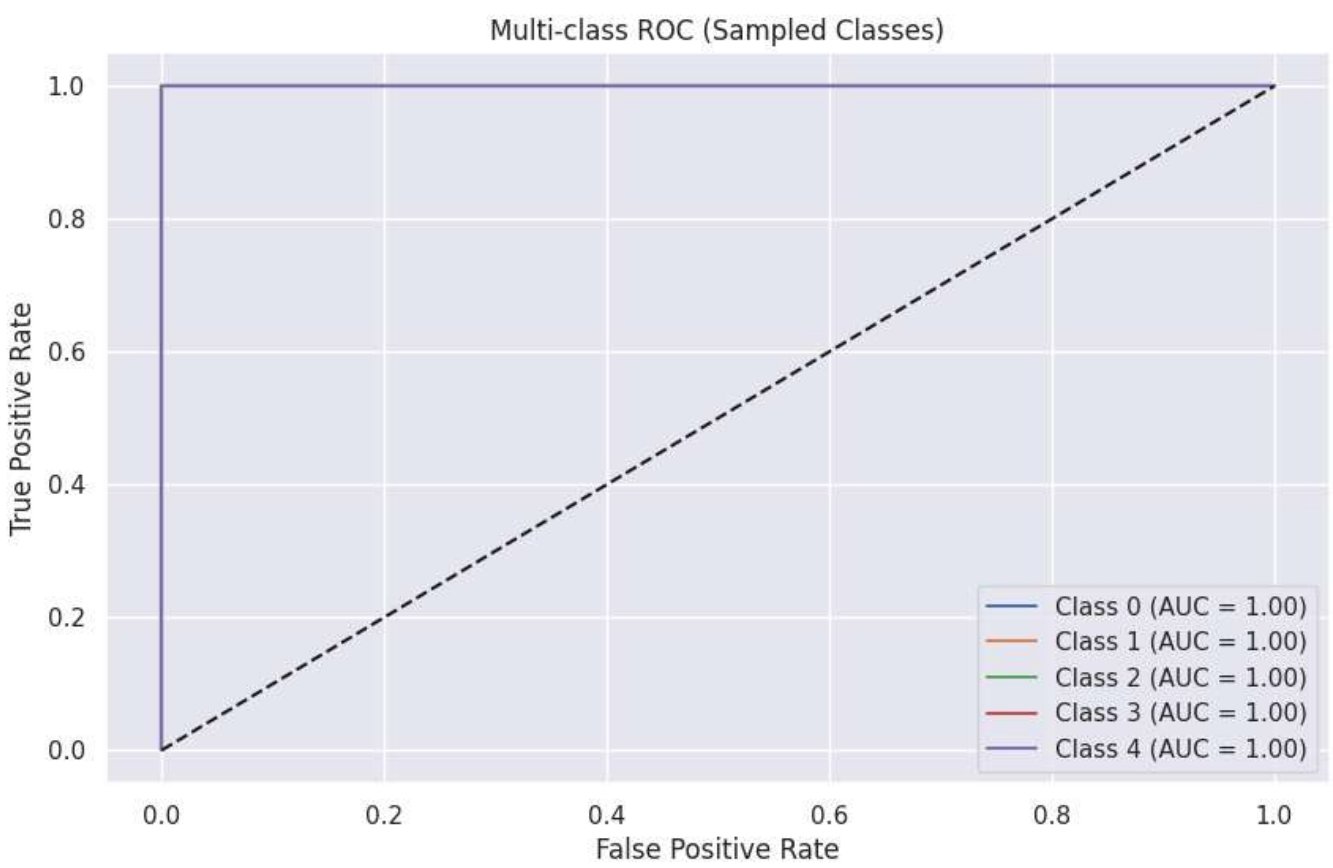
ROC and Precision-Recall graphs.

```
# Predict probabilities on validation set
y_val_pred_proba = best_model.predict(X_val)
```

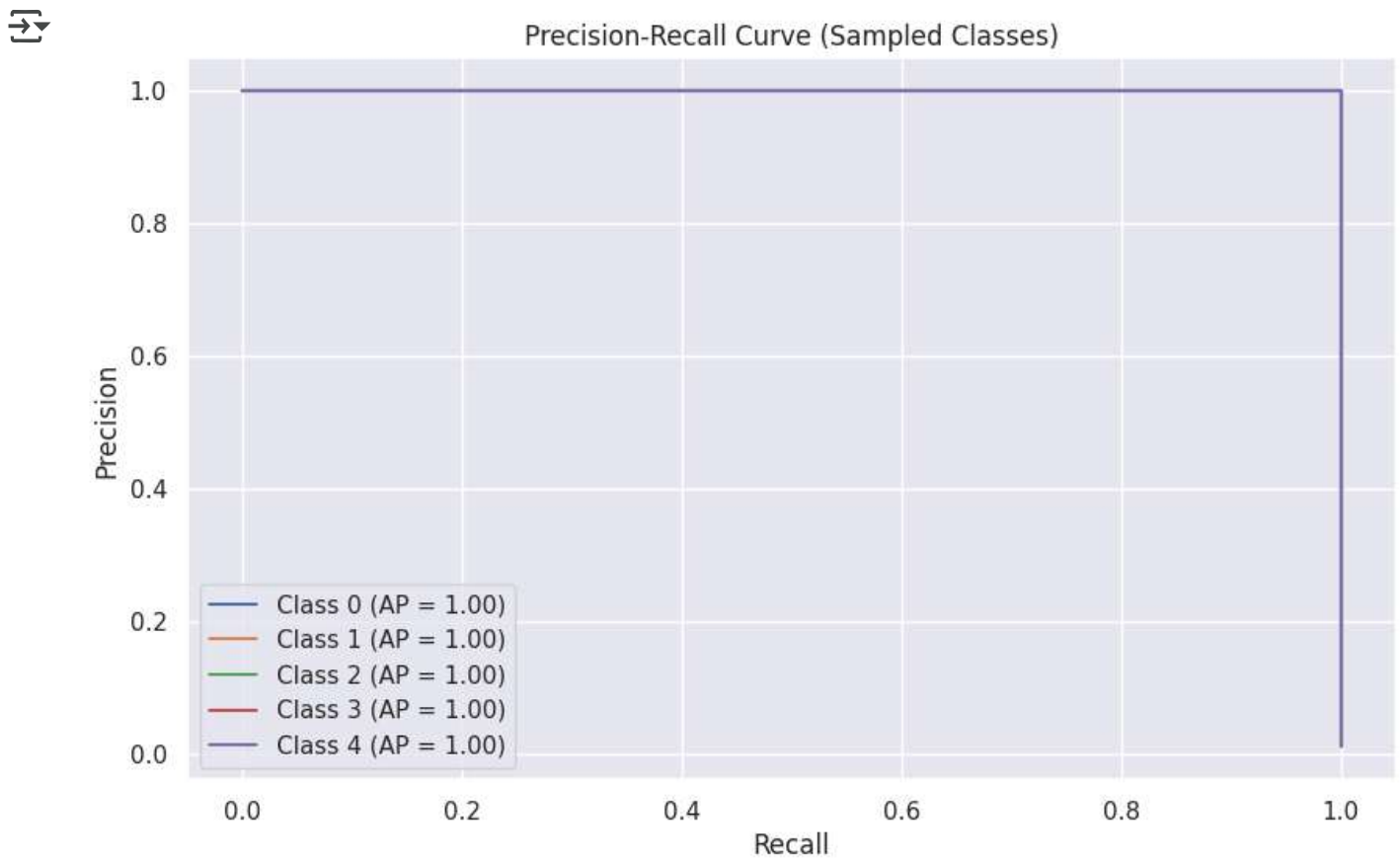
```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step
```

```
# ROC and AUC
fpr = dict()
```

```
tpr = dict()
roc_auc = dict()

for i in range(y_val.shape[1]):
    fpr[i], tpr[i], _ = roc_curve(y_val[:, i], y_val_pred_proba[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot 5 example classes
plt.figure(figsize=(10, 6))
for i in range(5):
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC = {roc_auc[i]:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.title('Multi-class ROC (Sampled Classes)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.grid(True)
plt.show()
```



```
# Precision-recall curves
precision = dict()
recall = dict()
avg_precision = dict()

for i in range(y_val.shape[1]):
    precision[i], recall[i], _ = precision_recall_curve(y_val[:, i], y_val_pred_proba[:, i])
    avg_precision[i] = average_precision_score(y_val[:, i], y_val_pred_proba[:, i])

# Plot 5 example classes
plt.figure(figsize=(10, 6))
for i in range(5):
    plt.plot(recall[i], precision[i], label=f'Class {i} (AP = {avg_precision[i]:.2f})')
plt.title('Precision-Recall Curve (Sampled Classes)')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.grid(True)
plt.show()
```



Introduction

Part 1 Code (Data Prep) - https://colab.research.google.com/drive/1dN0y6HYqIjKFhqZpgYYhlCY2ecmYSoN6?usp=sharing

Part 2 Code (NN DoE) - https://colab.research.google.com/drive/1xpimK1T2i8VQ9clx9Jx69ZKy6m0844Vu?usp=sharing

Part 3 Code (CNN Models) - https://colab.research.google.com/drive/1XCXj9PZTnriaDO9sqpb-PucG1fMsIN4B?usp=sharing

Data taken from - https://www.kaggle.com/competitions/leaf-classification/data

Ready data files

Import modules and load data

[ ] ↳ 1 cell hidden

Import images

[ ] ↳ 1 cell hidden

Export labels for each image

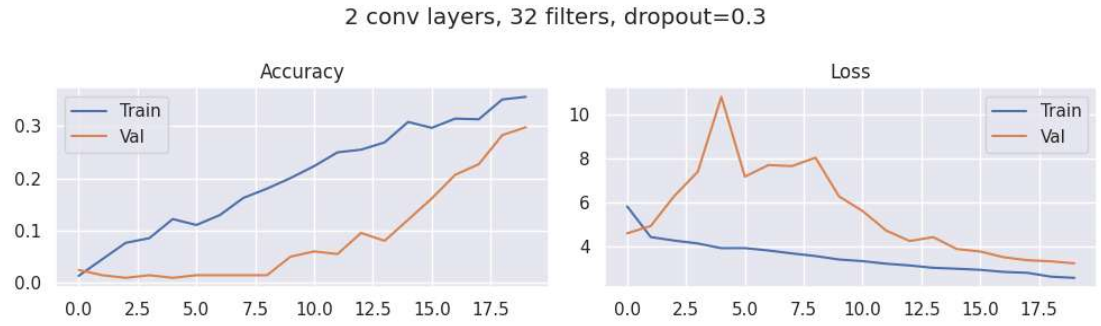
↳ 3 cells hidden

CNN Models

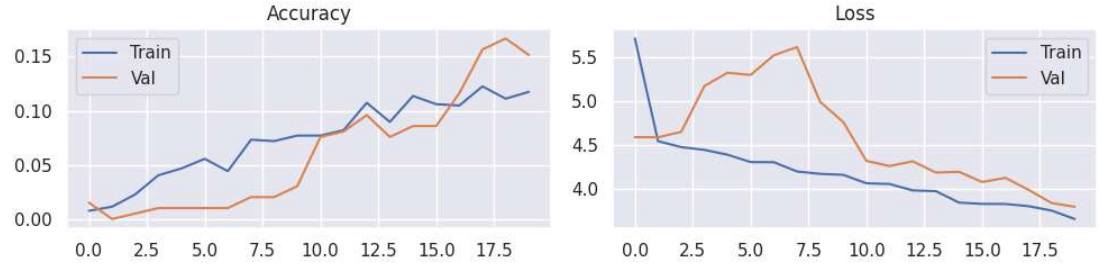
```
# Filter to training data and build image paths

df_train = df_labels[df_labels['train'] == 1].copy()
df_train['file_path'] = df_train['id'].astype(str).apply(lambda x: os.path.join("images", "images", f"{x}.jpg"))


# Encode species and split data

label_encoder = LabelEncoder()
df_train['label'] = label_encoder.fit_transform(df_train['species'])

class_names = label_encoder.classes_
num_classes = len(class_names)

df_train, df_val = train_test_split(df_train, test_size=0.2, stratify=df_train['label'], random_state=42)



IMG_SIZE = (96, 96) # started with 128 x 128 but cut down to improve learning speed
BATCH_SIZE = 32

def preprocess_image(path, label):
    image = tf.io.read_file(path)
    image = tf.image.decode_jpeg(image, channels=1) # grayscale not RGB
    image = tf.image.resize(image, IMG_SIZE)
    image = tf.cast(image, tf.float32) / 255.0 # normalized to 255, but not sure if that much detail is required
    return image, tf.one_hot(label, num_classes)

# Training dataset
train_ds = tf.data.Dataset.from_tensor_slices((df_train['file_path'].values, df_train['label'].values))
train_ds = train_ds.map(preprocess_image).shuffle(1000).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

# Validation dataset
val_ds = tf.data.Dataset.from_tensor_slices((df_val['file_path'].values, df_val['label'].values))
val_ds = val_ds.map(preprocess_image).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)



# Define hyperparameter grid
conv_layers_list = [2, 3]
filters_list = [32, 64]
dropout_rates = [0.3, 0.5]

results = []
best_val_acc = 0
best_model_path = "best_cnn_model.h5"

for num_conv in conv_layers_list:
    for filters in filters_list:
        for dropout in dropout_rates:
            print(f"Training model: {num_conv} conv layers, {filters} filters, dropout={dropout}")

            # Build model
            model = Sequential()
            model.add(tf.keras.Input(shape=(96, 96, 1)))

            for _ in range(num_conv):
                model.add(Conv2D(filters, (3, 3), activation='relu', padding='same'))
```

```
                model.add(BatchNormalization())
                model.add(MaxPooling2D((2, 2)))

            model.add(Flatten())
            model.add(Dense(128, activation='relu'))
            model.add(Dropout(dropout))
            model.add(Dense(num_classes, activation='softmax'))

            model.compile(optimizer='adam',
                          loss='categorical_crossentropy',
                          metrics=['accuracy'])

            # Train with timing
            start_time = time.time()
            history = model.fit(train_ds,
                                validation_data=val_ds,
                                epochs=20,
                                verbose=0)
            end_time = time.time()

            # Evaluate performance
            val_loss, val_acc = model.evaluate(val_ds, verbose=0)
            train_acc = history.history['accuracy'][-1]
            train_loss = history.history['loss'][-1]

            results.append({
                'conv_layers': num_conv,
                'filters': filters,
                'dropout': dropout,
                'train_accuracy': round(train_acc, 4),
                'train_loss': round(train_loss, 4),
                'val_accuracy': round(val_acc, 4),
                'val_loss': round(val_loss, 4),
                'train_time_sec': round(end_time - start_time, 2)
            })

            # Save best model
            if val_acc > best_val_acc:
                best_val_acc = val_acc
                model.save(best_model_path)

            # Plot training curves
            plt.figure(figsize=(10, 3))
            plt.subplot(1, 2, 1)
            plt.plot(history.history['accuracy'], label='Train')
            plt.plot(history.history['val_accuracy'], label='Val')
            plt.title('Accuracy')
            plt.legend()

            plt.subplot(1, 2, 2)
            plt.plot(history.history['loss'], label='Train')
            plt.plot(history.history['val_loss'], label='Val')
            plt.title('Loss')
            plt.legend()

            plt.suptitle(f"{num_conv} conv layers, {filters} filters, dropout={dropout}")
            plt.tight_layout()
            plt.show()

# Summarize all runs
df_results = pd.DataFrame(results)
print("Hyperparameter Tuning Results:")
print(df_results)

# Load best model
best_model = tf.keras.models.load_model(best_model_path)
```

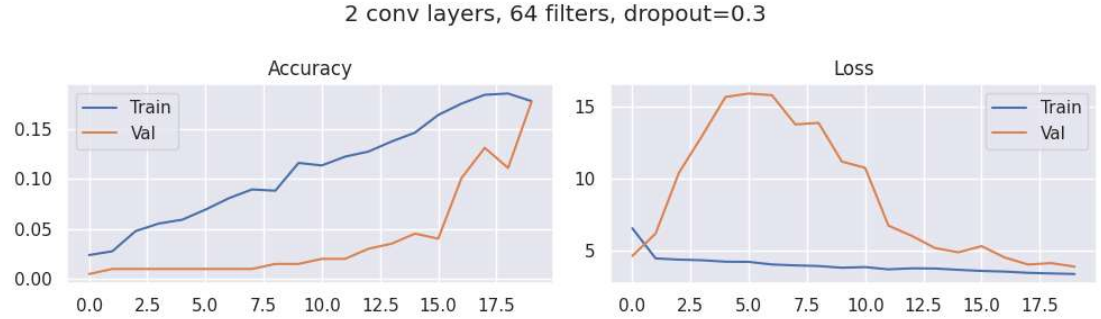
```
Training model: 2 conv layers, 32 filters, dropout=0.3
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend
```



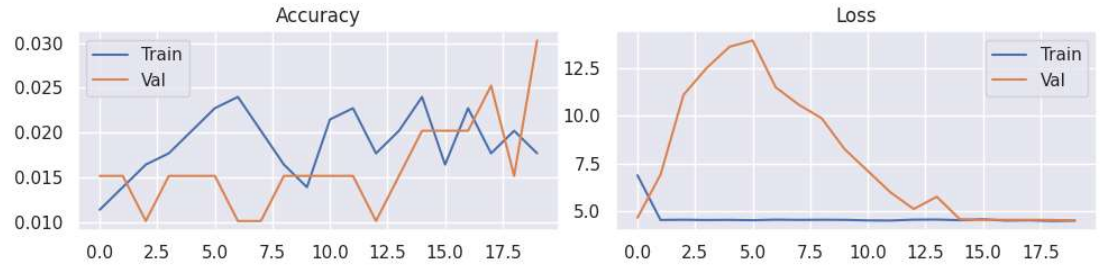

```
Training model: 2 conv layers, 32 filters, dropout=0.5
```



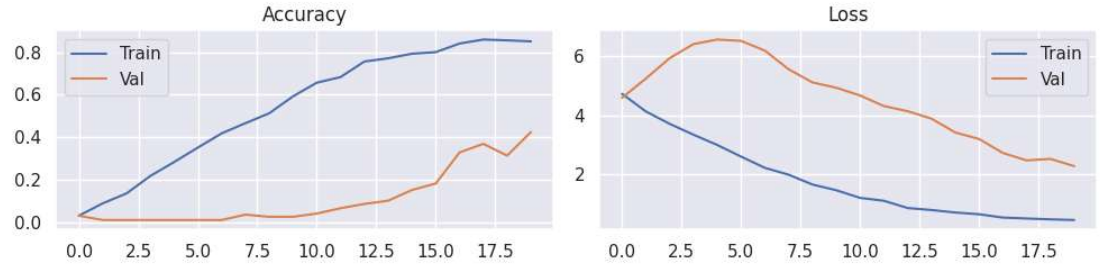

```
Training model: 2 conv layers, 64 filters, dropout=0.3
```



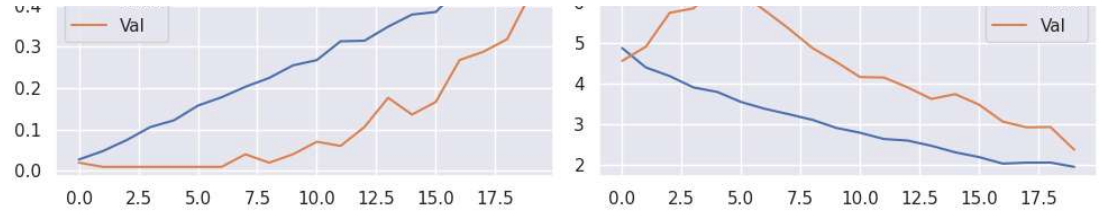

```
Training model: 2 conv layers, 64 filters, dropout=0.5
```



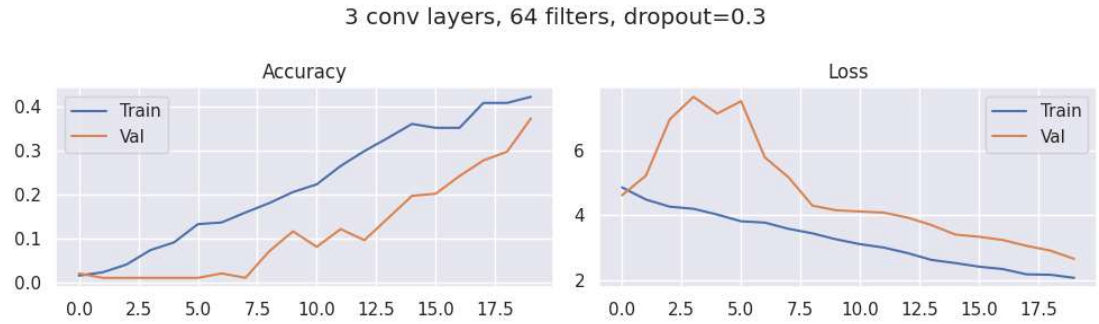

```
Training model: 3 conv layers, 32 filters, dropout=0.3
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend
```



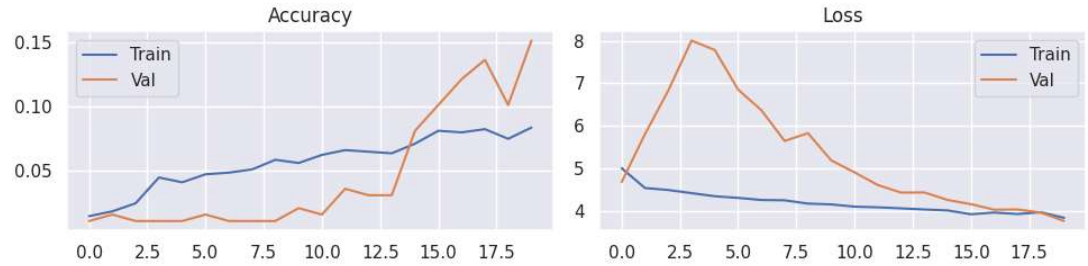

```
Training model: 3 conv layers, 32 filters, dropout=0.5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend
```




```
Training model: 3 conv layers, 64 filters, dropout=0.3
```




```
Training model: 3 conv layers, 64 filters, dropout=0.5
```




```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
Hyperparameter Tuning Results:
   conv_layers  filters  dropout  train_accuracy  train_loss  val_accuracy  \
0            2       32      0.3          0.3561      2.5515        0.2980
1            2       32      0.5          0.1174      3.6637        0.1515
2            2       64      0.3          0.1780      3.3646        0.1768
3            2       64      0.5          0.0177      4.5147        0.0303
4            3       32      0.3          0.8510      0.4587        0.4242
5            3       32      0.5          0.4470      1.9530        0.4293
6            3       64      0.3          0.4230      2.0821        0.3737
7            3       64      0.5          0.0833      3.8403        0.1515

   val_loss  train_time_sec
0    3.2136          310.14
1    3.8020          316.53
2    3.8745          657.10
3    4.5277          643.14
4    2.2768          291.72
5    2.3776          304.89
6    2.6645          653.33
7    3.7655          676.71
```

Takeaways

- Best model performance was validation accuracy of 42%
- 2-layer models consistently performed poorly, need to increase the number of convolution layers, next test should probably be 3, 4 and 5 layers
- Number of training epochs should be increased and addition of early stopping might be beneficial
- Model can be further generalized by flipping the images or zooming the images