Inhalt

Einführung

Der Kartoffelanbau ist ein zentraler Pfeiler der Landwirtschaft, der jedoch kontinuierlich durch Pflanzenkrankheiten bedroht wird. Eine der größten Gefahren stellt die Kraut- und Knollenfäule (Late Blight) dar. Diese Krankheit ist hochinfektiös und kann sich unter günstigen Witterungsbedingungen extrem schnell ausbreiten, was oft zum Totalverlust ganzer Felder führt. Auch die Alternaria-Dürrfleckenkrankheit (Early Blight) beeinträchtigt die Pflanzengesundheit und reduziert den Ertrag erheblich.

Die traditionelle Feldüberwachung erfolgt manuell durch Landwirte oder Agronomen. Dieser Prozess ist zeitaufwendig, teuer und oft nicht schnell genug, um auf einen aggressiven Ausbruch wie Late Blight rechtzeitig zu reagieren.

Hier setzt unser Projekt an: Das Ziel ist die Entwicklung eines automatisierten Monitoring-Systems auf Basis von Computer Vision. Durch die Analyse von Bilddaten der Kartoffelblätter soll das System in der Lage sein, einen Befall durch Early Blight oder Late Blight frühzeitig und zuverlässig zu erkennen.

Der Mehrwert dieser Lösung liegt in der direkten Unterstützung des landwirtschaftlichen Managements. Wir definieren den Erfolg dieses Projekts anhand der folgenden drei Kriterien:

1. Zuverlässige Erkennung zur Ertragssicherung: Das System muss eine hohe Genauigkeit bei der Identifizierung der Krankheiten aufweisen. Eine frühzeitige Warnung ermöglicht es dem Landwirt, sofortige und gezielte Gegenmaßnahmen einzuleiten und so den Ertrag zu sichern.

2. Minimierung unnötiger Kosten: Durch die präzise Lokalisierung von Krankheitsherden kann der Einsatz von Pflanzenschutzmitteln (Fungiziden) optimiert werden. Anstatt ganze Felder präventiv zu behandeln, erlaubt das System eine bedarfsgerechte Anwendung.

Dies senkt die Betriebskosten und schont die Umwelt.
3. Nachweis der Wirksamkeit der Maßnahme: Im finalen Schritt muss messbar nachgewiesen werden, dass der Einsatz des automatisierten Systems zu einer effektiveren Krankheitskontrolle und damit zu einer messbaren Reduzierung von Ernteausfällen im Vergleich zu traditionellen Methoden führt.

Dataset

1. Quelle und Übersicht

Der für dieses Projekt verwendete Datensatz ist eine Sammlung von Bildern, die Blätter von Kartoffelpflanzen zeigen. Er wurde von der Online-Plattform Kaggle bezogen und ist öffentlich für Forschungs- und Entwicklungszwecke im Bereich Computer Vision verfügbar.

```
Quelle: https://www.kaggle.com/datasets/faysalmiah1721758/potato-dataset
```

2. Inhalt und Klassen

Der Datensatz dient der Klassifizierung von drei verschiedenen Zuständen eines Kartoffelblattes. Jedes Bild ist einer der folgenden drei Kategorien zugeordnet:

- Potato Early blight: Bilder von Blättern, die Symptome der Dürrfleckenkrankheit aufweisen.
- Potato Late blight: Bilder von Blättern, die von der Kraut- und Knollenfäule befallen sind.
- Potato healthy: Bilder von gesunden Blättern ohne sichtbare Krankheitsanzeichen.

3. Datenverteilung

Der Datensatz besteht aus insgesamt 2152 Bildern. Die Aufteilung auf die einzelnen Klassen ist wie folgt:

Klasse	Anzahl
Potato Early blight (Alternaria-Dürrfleckenkrankheit)	1000

Potato Late blight (Kraut- und Knollenfäule)	1000
Potato healthy (gesunde Blätter)	152
Gesamt	**2152**

4. Beispielbilder

Healthy

Early Blight

Late Blight

5. Anmerkung zum Klassenungleichgewicht

Wie aus der Tabelle ersichtlich ist, liegt ein signifikantes Klassenungleichgewicht (Class Imbalance) vor. Die beiden Krankheitsklassen sind mit jeweils 1000 Bildern stark vertreten, während die Klasse der gesunden Blätter mit 152 Bildern deutlich unterrepräsentiert ist.

Dieses Ungleichgewicht muss beim Training von Machine-Learning-Modellen berücksichtigt werden, um eine Verzerrung (Bias) des Modells zugunsten der Mehrheitsklassen zu vermeiden. Mögliche Gegenmaßnahmen sind beispielsweise:

- Datenerweiterung (Data Augmentation) speziell für die unterrepräsentierte Klasse.
- Oversampling der "healthy"-Klasse (z. B. mittels SMOTE).
- Undersampling der "blight"-Klassen.

- Verwendung einer gewichteten Verlustfunktion (Weighted Loss Function) während des Trainings.

Code

Die Code Zellen können nacheinander einzeln ausgeführt werden. Die Zellen für die visualisierung der Feature Maps sind für das Training nicht notwendig und können übersprungen werden.

Es werden alle notwendigen Bibliotheken imiportiert:

- kagglehub: Zum Herunterladen des Datensatzes von Kaggle.
- tensorflow & keras: Das Haupt-Framework für den Aufbau und das Training des Neuronalen Netzes.
- ImageDataGenerator: Ein Keras-Werkzeug, um Bilder von der Festplatte zu laden, sie "on-the-fly" zu augmentieren (z.B. drehen, zoomen) und in Batches an das Modell zu übergeben.
- regularizers: Wird benötigt, um Regularisierungstechniken (wie L2) auf die Schichten des Modells anzuwenden, was Overfitting reduziert.
- compute_class_weight: Eine Scikit-learn-Funktion, die hilft, das Modell-Training bei unausgewogenen Datensätzen (ungleiche Anzahl von Bildern pro Klasse) zu balancieren.
- numpy: Standardbibliothek für numerische Operationen.
- matplotlib.pyplot: Wird verwendet, um die Trainingsergebnisse zu visualisieren.
- pathlib.Path: Bietet eine moderne, objektorientierte Schnittstelle zur Handhabung von Dateipfaden.

In [ ]:
```
import os
os.environ["TF_ENABLE_ONEDNN_OPTS"] = "0"

import kagglehub
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, regularizers
from sklearn.utils.class_weight import compute_class_weight
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
```

Diese Zelle lädt den "Potato Dataset" mithilfe der `kagglehub`-Bibliothek herunter. Der zurückgegebene Pfad (`path`) zeigt auf den lokalen Speicherort des Datensatzes. Anschließend wird der Datensatz mit `pathlib` durchsucht. Die Schleife dient als "Sanity Check": Sie iteriert durch alle Unterordner, gibt deren Namen aus (die typischerweise den Klassennamen entsprechen, z.B. `Potato___Early_blight`) und zählt die Anzahl der Bilddateien in jedem Ordner.

In [ ]:
```
path = kagglehub.dataset_download("faysalmiah1721758/potato-dataset")
print(f"Dataset Pfad: {path}")

dataset_path = Path(path)
for item in dataset_path.rglob("*"):
    if item.is_dir():
        print(f"Ordner Name: {item.name}")
        image_count = len(list(item.glob("*.png"))) + len(list(item.glob("*.jpg"))) + len(list(item.glob("*.jpeg"))
        if image_count > 0:
            print(f"{image_count} Bilder gefunden")
        else:
            print("Keine Bilder gefunden")
```

Hier werden die grundlegenden Hyperparameter für das Training festgelegt.

- IMG_SIZE = 224: Definiert die Zielgröße (224x224 Pixel), auf die alle Bilder skaliert werden. CNNs benötigen eine einheitliche Eingabegröße. 224x224 ist ein gängiger Standard, der von vielen bekannten Architekturen (wie VGG) verwendet wird.
- BATCH_SIZE = 32: Legt fest, wie viele Bilder das Modell auf einmal verarbeitet, bevor es seine Gewichte aktualisiert. 32 ist ein üblicher Kompromiss zwischen Recheneffizienz, Speicherbedarf und Stabilität des Trainings.
- train_dir: Speichert den Pfad zum Hauptverzeichnis des Datensatzes für späteren Zugriff.

In [ ]:
```
IMG_SIZE = 224
BATCH_SIZE = 32

train_dir = dataset_path #Trainingsordner

print(f"Trainingsorder ist {train_dir}")
print(f"Bildgroesse ist {IMG_SIZE} X {IMG_SIZE}")
print(f"Es werden {BATCH_SIZE} Bilder pro Batch (auf einmal) betrachtet")
```

Diese Zelle bereitet die Bilddaten für das Training vor. Der `ImageDataGenerator` wird mit drei Hauptaufgaben konfiguriert:

1. Normierung (`rescale=1/255`): Dies ist ein essenzieller Schritt. Alle Pixelwerte (ursprünglich [0, 255]) werden auf den Bereich [0, 1] skaliert. Neuronale Netze lernen mit diesen kleineren Werten wesentlich effizienter und stabiler.
2. Data Augmentation (z.B. `rotation_range` , `zoom_range` ...): Das Modell wird mit künstlich veränderten Versionen der Trainingsbilder trainiert (zufällig gedreht, gezoomt, gespiegelt etc.). Dies vergrößert den Datensatz virtuell und ist eine sehr effektive Methode, um Overfitting zu reduzieren, da das Modell lernt, robust gegenüber leichten Variationen zu sein.
3. Validation Split (`validation_split=0.2`): 20% der Daten werden automatisch für die Validierung reserviert.

In [ ]:
```
train_datagen = ImageDataGenerator(
    rescale=1/255, #RGB Werte werden normiert
    rotation_range=20, #Max 20° rotation
    width_shift_range=0.2, #Verschiebung links-recht (Relativ zur Breite)
    height_shift_range=0.2, #Verschiebung unten-oben (Relativ zur Höhe)
    shear_range=0.2, #Verzerrung
    zoom_range=0.2,#Zufälliger Zoom 0,8 bis 1,2
    horizontal_flip=True,
    validation_split=0.2, #20% werden zur Validierung verwendet
)

# Generator für die Trainingsdaten.
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode="categorical",
    subset="training"
)

# Generator für die Validierungsdaten.
val_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    subset='validation'
```

```
)

print(f"Gefundene Klassen: {train_generator.class_indices}")
num_classes = len(train_generator.class_indices)
```

Diese Zelle definiert die Architektur des Convolutional Neural Network (CNN) als Funktion. Das Modell ist ein `Sequential`-Stack von Schichten.

Faltungs-Blöcke (Merkmalsextraktion): Das Modell besteht aus drei Faltungs-Blöcken. Jeder Block folgt einem gängigen Muster:

1. `Conv2D`: Wendet Filter (z.B. 32, dann 64, dann 128) an, um Muster zu erkennen (Kanten, Texturen). `padding='same'` sorgt dafür, dass die Bildgröße durch die Faltung nicht schrumpft.
2. `BatchNormalization`: Stabilisiert und beschleunigt das Training durch Normalisierung der Aktivierungen zwischen den Schichten.
3. `Activation('relu')`: Führt Nichtlinearität ein (`f(x) = max(0, x)`), damit das Netz komplexe Zusammenhänge lernen kann.
4. `MaxPooling2D`: Reduziert die räumliche Auflösung (Downsampling). Dies spart Rechenleistung und macht das Modell robuster gegenüber kleinen Verschiebungen im Bild.

Klassifikations-Blöcke (Entscheidungsfindung):

1. `Flatten`: Wandelt die 3D-Feature-Maps (Höhe x Breite x Kanäle) aus dem letzten Faltungsblock in einen langen 1D-Vektor um.
2. `Dense(256)`: Eine voll verbundene Schicht, die die gelernten Merkmale kombiniert.
3. `kernel_regularizer=regularizers.l2(0.001)`: Wendet L2-Regularisierung an (eine "Strafe" für große Gewichte), um Overfitting zu reduzieren.
4. `Dropout(0.4)`: Deaktiviert während des Trainings zufällig 40% der Neuronen in dieser Schicht. Dies zwingt das Netz, redundante Informationen zu lernen und ist eine sehr effektive Methode gegen Overfitting.
5. `Dense(num_classes, activation='softmax')`: Die finale Output-Schicht. Sie hat so viele Neuronen, wie es Klassen gibt (`num_classes`). Die `softmax`-Aktivierung wandelt die Ausgabe in eine Wahrscheinlichkeitsverteilung um (alle Ausgaben summieren sich zu 1), die angibt, wie sicher sich das Modell bei jeder Klasse ist.

`summary()` gibt eine textuelle Zusammenfassung der Modellarchitektur aus, die die Schichten, ihre Ausgabe-Formen und die Anzahl der Parameter zeigt.

In [ ]:
```
def create_small_cnn(num_classes):
    model = keras.Sequential([
```

```
        # Block 1
        layers.Conv2D(32, (3, 3), padding='same', input_shape=(IMG_SIZE, IMG_SIZE, 3)),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPooling2D((2, 2)),

        # Block 2
        layers.Conv2D(64, (3, 3), padding='same'),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPooling2D((2, 2)),

        # Block 3
        layers.Conv2D(128, (3, 3), padding='same'),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPooling2D((2, 2)),

        # Klassifikations-Teil
        layers.Flatten(),
        layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
        layers.Dropout(0.4),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

# Erstellen des Modells
model = create_small_cnn(num_classes)
model.summary()
```

Diese Zelle definiert eine komplexe Hilfsfunktion, `save_feature_maps_comparison`, um die internen Vorgänge des CNNs zu visualisieren.

Was sind Feature Maps? Eine Feature Map ist die Ausgabe eines Filters in einer `Conv2D`-Schicht. Sie zeigt, welche Bereiche des Bildes diesen spezifischen Filter "aktiviert" haben. Die Visualisierung hilft zu verstehen, worauf das Netz "achtet" (z.B. Kanten, Texturen).

Was macht die Funktion?

1. Sie identifiziert alle `Conv2D`-Schichten im Modell.

2. Sie erstellt ein neues "Sub-Modell" (`feature_map_model`), das denselben Input wie das Originalmodell nimmt, aber als Output die Aktivierungen (die Feature Maps) *jeder* `Conv2D` -Schicht zurückgibt.
3. Sie führt ein Testbild (`img_array`) durch dieses Sub-Modell, um die Feature Maps zu berechnen.
4. Sie iteriert durch die Feature Maps jeder Schicht und speichert für jede Schicht ein Bild. Dieses Bild zeigt das Originalbild links und rechts daneben drei Beispiel-Feature-Maps (die erste, die mittlere und die letzte) aus dieser Schicht.
5. Die Bilder werden im Ordner `feature_maps` gespeichert.

In [ ]:

```python
def save_feature_maps_comparison(model, img_array, save_dir='feature_maps'):
    """
    Speichert Original-Bild und jeweils 3 Feature Maps pro Conv-Block
    """

    Path(save_dir).mkdir(exist_ok=True)


    conv_layer_names = [layer.name for layer in model.layers
                        if 'conv' in layer.name.lower()]

    # Feature Map Modell erstellen
    layer_outputs = [model.get_layer(name).output for name in conv_layer_names]
    feature_map_model = keras.Model(
        inputs=model.layers[0].input,
        outputs=layer_outputs
    )

    # Feature Maps berechnen
    feature_maps = feature_map_model.predict(img_array, verbose=0)

    print(f"\nSpeichere Feature Maps in Ordner: {save_dir}/\n")

    # Für jeden Conv-Block: Original + 3 Feature Maps speichern
    for block_idx, (layer_name, feature_map) in enumerate(zip(conv_layer_names, feature_maps), 1):
        num_features = feature_map.shape[-1]

        # Figure mit 4 Subplots erstellen
        fig, axes = plt.subplots(1, 4, figsize=(20, 5))
        fig.suptitle(f'Block {block_idx}: {layer_name} ({num_features} Feature Maps gesamt)',
                     fontsize=16, fontweight='bold')
```

```
        # Original-Bild
        axes[0].imshow(img_array[0])
        axes[0].set_title('Original-Bild', fontsize=12)
        axes[0].axis('off')

        # 3 Feature Maps zeigen
        feature_indices = [0, num_features//2, num_features-1] # Erste, mittlere, letzte

        for i, feat_idx in enumerate(feature_indices, 1):
            if feat_idx < num_features:
                channel_image = feature_map[0, :, :, feat_idx]

                # Werte für Visualisierung
                vmin, vmax = channel_image.min(), channel_image.max()

                im = axes[i].imshow(channel_image, cmap='viridis')
                axes[i].set_title(f'Feature Map {feat_idx}\n'
                                  f'Range: [{vmin:.2f}, {vmax:.2f}]',
                                  fontsize=10)
                axes[i].axis('off')

                # Colorbar hinzufügen
                plt.colorbar(im, ax=axes[i], fraction=0.046, pad=0.04)

        plt.tight_layout()

        # Speichern
        filename = f'{save_dir}/block_{block_idx}_{layer_name}.png'
        plt.savefig(filename, dpi=150, bbox_inches='tight')
        plt.close() # Figure schließen um Speicher zu sparen

        print(f"[OK] Gespeichert: {filename}")

        # Info ausgeben
        print(f"  Output Shape: {feature_map.shape}")
        print(f"  Anzahl Feature Maps: {num_features}")
        print(f"  Spatial Groesse: {feature_map.shape[1]} x {feature_map.shape[2]}")
        print(f"  Wertebereich: [{feature_map.min():.3f}, {feature_map.max():.3f}]\n")

    print(f"\n{'='*60}")
```

```
    print(f"Alle Feature Maps wurden erfolgreich gespeichert!")
    print(f"Ordner: {save_dir}/")
    print(f"{'='*60}\n")
```

In dieser Zelle wird die in der vorherigen Zelle definierte Visualisierungsfunktion aufgerufen.

1. `next(train_generator)` : Holt einen einzelnen Batch (32 Bilder) aus dem Trainingsdatensatz.
2. `test_img = test_images[0:1]` : Wählt das erste Bild aus diesem Batch aus. (Der Slice `[0:1]` statt nur `[0]` wird verwendet, um die Batch-Dimension `(1, 224, 224, 3)` zu erhalten, die das Modell erwartet.)
3. `save_feature_maps_comparison(...)` : Ruft die Funktion auf, um die Aktivierungen dieses Testbildes zu berechnen und die Diagramme im Ordner `feature_maps` zu speichern.

Wichtiger Hinweis: Das Modell ist zu diesem Zeitpunkt noch **untrainiert**. Die Gewichte sind zufällig, und die Feature Maps werden daher noch keine sinnvollen Muster zeigen. Diese Zelle ist am nützlichsten, wenn man sie *nach* dem erfolgreichen Training des Modells erneut ausführt.

Der `print` -Block am Ende gibt eine Interpretation, was man typischerweise in den Feature Maps *nach* dem Training sieht.

In [ ]:
```
#Ein Testbild holen
test_images, test_labels = next(train_generator)
test_img = test_images[0:1] # Erstes Bild nehmen

print("\n" + "="*60)
print("FEATURE MAPS VISUALISIERUNG - Speichere Bilder...")
print("="*60 + "\n")

# Feature Maps speichern
save_feature_maps_comparison(model, test_img, save_dir='feature_maps')

print("""
ERKLAERUNG DER GESPEICHERTEN BILDER:

Block 1 (32 Maps):
  - Einfache Muster wie Kanten, Farbverlaeufe
  - Hellere Bereiche = staerkere Aktivierung
  - Jede Map reagiert auf unterschiedliche Muster

Block 2 (64 Maps):
```

```
  - Kombiniert die 32 Maps von Block 1
  - Erkennt komplexere Formen (z.B. Flecken-Raender)
  - Kleinere Aufloesung durch MaxPooling

Block 3 & 4:
  - Noch abstraktere Muster
  - Erkennen Kombinationen von Block 2
  - Sehr kleine Aufloesung, aber semantisch reich

Die Grauwerte sind KONTINUIERLICH, nicht binaer!
Das ermoeglicht nuancierte Informationen fuer spaetere Blocks.

Schau dir die Bilder im Ordner 'feature_maps' an!
""")
```

Der `model.compile()`-Schritt konfiguriert das Modell für den Trainingsprozess. Hier wird festgelegt, *wie* das Modell lernen soll:

1. `optimizer=keras.optimizers.Adam(learning_rate=initial_lr)`: Wählt den Optimierungsalgorithmus. `Adam` ist ein sehr beliebter und effektiver Optimizer, der die Lernrate (`initial_lr = 0.0005`) während des Trainings intelligent anpasst, um schnell einen guten "Loss" (Fehlerwert) zu erreichen.

2. `loss="categorical_crossentropy"`: Definiert die Verlustfunktion. Diese Funktion misst, wie "falsch" die Vorhersagen des Modells sind. Da wir ein Mehrklassen-Klassifikationsproblem haben (mehr als 2 Klassen) und unsere Labels im One-Hot-Format vorliegen (`class_mode='categorical'`), ist dies die Standard-Loss-Funktion. Das Ziel des Trainings ist es, diesen Wert zu minimieren.

3. `metrics=["accuracy"]`: Gibt an, welche Metriken wir während des Trainings überwachen möchten. `accuracy` (Genauigkeit) ist der Prozentsatz der korrekt klassifizierten Bilder und für uns Menschen leichter zu interpretieren als der abstrakte Loss-Wert.

In [ ]:
```python
# Optimizer & Compile
initial_lr = 0.0005
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=initial_lr),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)
```

Diese Zelle adressiert das Problem der **Klassen-Imbalance**. Wenn der Datensatz unausgewogen ist (z.B. 1000 Bilder von "gesunden" Kartoffeln, aber nur 100 von "kranken"), könnte das Modell lernen, einfach immer "gesund" zu raten, um eine hohe Genauigkeit zu erzielen.

`compute_class_weight` von Scikit-learn analysiert die Verteilung der Labels (`train_generator.classes`).

- `class_weight='balanced'` : Weist Klassen mit *weniger* Beispielen automatisch ein *höheres* Gewicht zu.
- Das Ergebnis wird in ein Dictionary `class_weights` umgewandelt (z.B. `{0: 0.8 (häufig), 1: 2.5 (selten)}`).

Wenn dieses Dictionary später an `model.fit()` übergeben wird, wird der Loss (Fehler) bei einer falschen Vorhersage für eine seltene Klasse (Klasse 1) mit einem höheren Faktor (2.5) multipliziert. Das Modell wird also stärker "bestraft" und gezwungen, diese selteneren Klassen ernster zu nehmen.

In [ ]:
```
# Class Weights berechnen
labels = train_generator.classes
class_weights_array = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(labels),
    y=labels
)
class_weights = dict(enumerate(class_weights_array))
```

Callbacks sind Funktionen, die Keras an bestimmten Punkten im Training aufruft (z.B. am Ende jeder Epoche).

Hier wird `EarlyStopping` konfiguriert, eine extrem wichtige Technik, um Overfitting zu bekämpfen und Zeit zu sparen:

- `monitor="val_loss"` : Der Callback überwacht den Fehler (Loss) auf dem **Validierungsdatensatz** (den ungesehenen Daten).
- `patience=3` : Das Training wird abgebrochen, wenn sich der `val_loss` für 3 aufeinanderfolgende Epochen nicht signifikant (`min_delta=0.001`) verbessert hat.
- `restore_best_weights=True` : Dies ist der wichtigste Parameter. Wenn das Training z.B. in Epoche 15 abbricht, weil Overfitting begann, stellt dieser Befehl sicher, dass das Modell automatisch auf den Zustand von Epoche 12 (dem Punkt mit dem niedrigsten `val_loss`) zurückgesetzt wird. Man erhält somit das Modell mit der besten Generalisierungsfähigkeit.

In [ ]:
```
# Callbacks
callbacks = [
    keras.callbacks.EarlyStopping(
```

```
        monitor="val_loss",
        patience=3,
        min_delta=0.001,
        restore_best_weights=True
    )

]
```

Dies ist der Befehl, der das eigentliche Training startet.

- `EPOCHS = 30` : Definiert die *maximale* Anzahl an Durchläufen durch den gesamten Trainingsdatensatz. Dank `EarlyStopping` (aus den `callbacks`) wird das Training wahrscheinlich schon früher abbrechen.
- `model.fit(...)` : Startet den Trainingsprozess.
- `train_generator` : Liefert die Trainingsdaten (Bilder und Labels) in Batches.
- `validation_data=val_generator` : Gibt den Generator an, der die Validierungsdaten liefert. Nach jeder Epoche wird das Modell auf diesen Daten getestet.
- `callbacks=callbacks` : Übergibt die zuvor definierte `EarlyStopping` -Funktion.
- `class_weight=class_weights` : Übergibt die berechneten Klassengewichte, um die Klassen-Imbalance auszugleichen.

Der Rückgabewert `history` ist ein Objekt, das alle Metriken (loss, accuracy, val_loss, val_accuracy) für jede abgeschlossene Epoche speichert.

In [ ]:
```
EPOCHS = 30

history = model.fit(
    train_generator,
    epochs=EPOCHS,
    validation_data=val_generator,
    callbacks=callbacks,
    class_weight=class_weights
)
```

Diese Zelle definiert und ruft eine Funktion (`plot_training_history`) auf, um die Ergebnisse des Trainingsprozesses zu visualisieren. Sie verwendet das `history` -Objekt, das von `model.fit()` zurückgegeben wurde.

Es werden zwei Diagramme erstellt:

1. Model Accuracy: Zeigt die Genauigkeit auf den Trainingsdaten (`accuracy`) und den Validierungsdaten (`val_accuracy`) über die Epochen hinweg.
2. Model Loss: Zeigt den Fehler auf den Trainingsdaten (`loss`) und den Validierungsdaten (`val_loss`) über die Epochen hinweg.

Interpretation der Graphen:

- Idealfall: Trainings- und Validierungskurven verlaufen nah beieinander und verbessern sich (Loss sinkt, Accuracy steigt).
- Overfitting: Die Trainingskurve verbessert sich weiter (z.B. `accuracy` steigt auf 100%), während die Validierungskurve stagniert oder sich verschlechtert (z.B. `val_loss` steigt an).

Das Diagramm wird auch als `training_history.png` gespeichert.

In [ ]:
```
def plot_training_history(history):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    # Accuracy Graph
    ax1.plot(history.history['accuracy'], label='Training Accuracy')
    ax1.plot(history.history['val_accuracy'], label='Validation Accuracy')
    ax1.set_title('Model Accuracy')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Accuracy')
    ax1.legend()
    ax1.grid(True)

    # Loss Graph
    ax2.plot(history.history['loss'], label='Training Loss')
    ax2.plot(history.history['val_loss'], label='Validation Loss')
    ax2.set_title('Model Loss')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Loss')
    ax2.legend()
    ax2.grid(True)

    plt.tight_layout()
    plt.savefig('training_history.png')
    plt.show()

plot_training_history(history)
```

In der letzten Zelle werden die Ergebnisse des Trainings für die spätere Verwendung gespeichert.

1. `model.save('potato_disease_model.keras')`: Speichert das gesamte trainierte Modell (Architektur, die gelernten Gewichte und die Optimizer-Konfiguration) in einer einzigen Datei im modernen `.keras`-Format. Diese Datei kann später geladen werden, um Vorhersagen auf neuen, ungesehenen Bildern zu machen (Inferenz).

2. Klassennamen speichern: Das Modell selbst gibt nur Zahlen (Indizes) als Vorhersage aus (z.B. `0`, `1`, `2`). Um diese Zahlen in für Menschen lesbare Namen (z.B. `Potato___healthy`) zu übersetzen, müssen wir die Zuordnung speichern, die der `ImageDataGenerator` erstellt hat.

    - `train_generator.class_indices` ist ein Dictionary wie `{'Healthy': 0, 'Sick': 1}`.
    - Wir drehen dieses Dictionary um (`{0: 'Healthy', 1: 'Sick'}`) und speichern es als `class_names.json`-Datei.
    - Bei der späteren Inferenz laden wir das Modell *und* diese JSON-Datei, um die numerischen Vorhersagen zu "übersetzen".

In [ ]:
```
model.save('potato_disease_model.keras')

# Klassennamen speichern
import json
class_names = {v: k for k, v in train_generator.class_indices.items()}
with open("class_names.json", "w") as f:
    json.dump(class_names, f)
```

Fazit

Analyse der Trainingsergebnisse

Die Trainings- und Validierungsgrafiken (siehe Abb. 1) zeigen ein sehr positives Ergebnis.

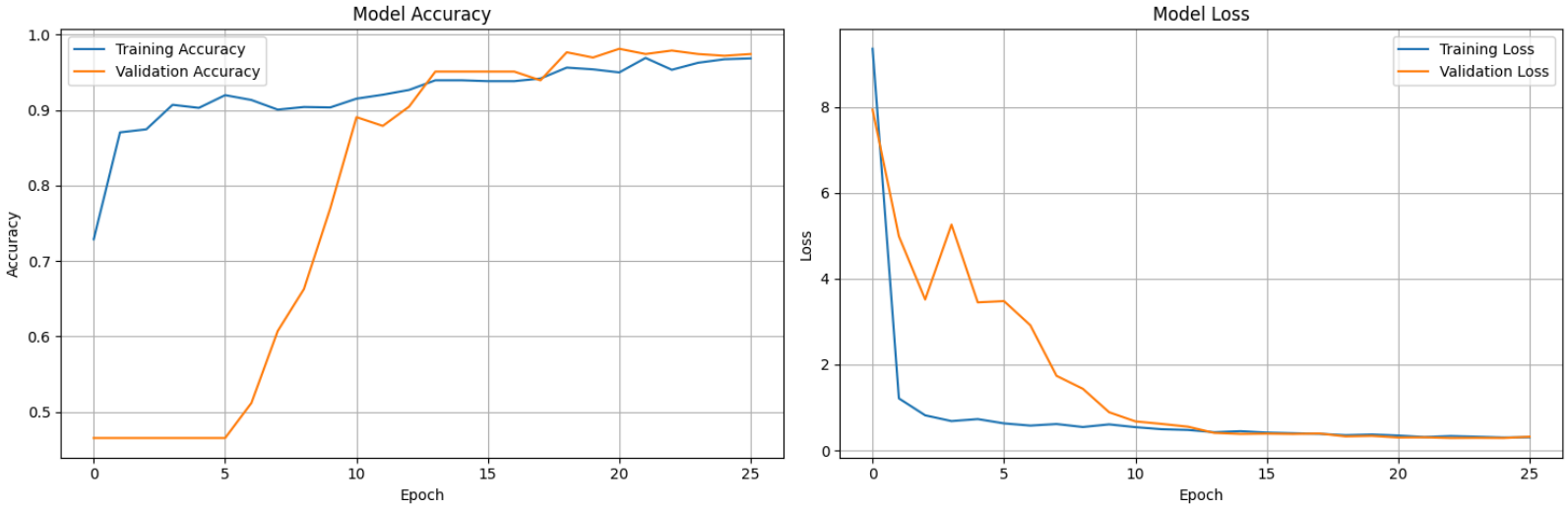

Abb. 1: Training Historie.

Nach einer initialen Phase (ca. bis Epoche 8), in der das Modell noch Schwierigkeiten hatte, die Muster der unterrepräsentierten "healthy"-Klasse zu generalisieren (erkennbar an der stagnierenden/fluktuierenden Validierungskurve), stabilisierte sich der Lernprozess deutlich.

Ab Epoche 10 konvergieren die Kurven für Training und Validierung eindrucksvoll:

- Hohe Genauigkeit (Accuracy): Sowohl die Trainings- als auch die Validierungsgenauigkeit erreichen am Ende der 25 Epochen einen Wert von ca. 98 %.
- Keine Überanpassung (Overfitting): Die Genauigkeitskurven (links) bleiben eng beieinander, und die Verlustkurven (Loss, rechts) konvergieren ebenfalls auf einen gemeinsamen, niedrigen Wert. Dies ist ein starkes Indiz dafür, dass das Modell die gelernten Muster gut auf ungesehene Daten übertragen kann und nicht nur den Trainingsdatensatz auswendig gelernt hat.

Die erfolgreiche Konvergenz ist ein direkter Beleg dafür, dass die Anwendung der Klassengewichtung (Class Weighting) zur Kompensation des Datenungleichgewichts erfolgreich war. Ohne diese Maßnahme wäre die Validierungsgenauigkeit (insbesondere für die "healthy"-Klasse) voraussichtlich deutlich schlechter ausgefallen.

Bezug zu den Erfolgskriterien

Die hohe Validierungsgenauigkeit von ~98 % ist ein entscheidender Schritt zur Erfüllung unseres primären Erfolgskriteriums: "Zuverlässige Erkennung zur Ertragssicherung".

Ein Modell, das mit dieser Präzision zwischen gesunden und kranken Blättern unterscheiden kann, legt den Grundstein für die weiteren Ziele:

- Es ermöglicht eine präzise Alarmierung und damit eine gezielte Behandlung, was die unnötigen Kosten (Erfolgskriterium 2) durch pauschales Spritzen minimiert.
- Es liefert die notwendige Datengrundlage, um die Wirksamkeit der Maßnahme (Erfolgskriterium 3) im Feldeinsatz zu messen.