Task 1 - Exploring Hyperparameters

Batch Size

Approach

Firstly I checked again what the batch size of a CNN is. It is the number of data-samples that will be put through together in one interation (not epoch) of the network."After each iteration the weights in CNN will be updated. If every data-sample was put through the network, a epoch is finished. So the batch-size on one way determines how 'long' a epoch will be.

The minimum batch-size for every CNN would be 1 and the maximum batch size the number of data-samples available.Minimum and maximum values should be definitly tested.

I selected this six values to represent a good range of results:

- 1
- 8
- 32
- 128
- 256
- 512 (=max of the samples)

Assumptions

I think the higher the batch size, the longer it will take to train the model for each iteration and also the lower the prediction accuracy will be. This could be because if you put all the data-samples in one go through the CNN, the weights will only update once (because there is only one iteration per epoch).

In [7]:

```
import cv2
import json
from matplotlib import pyplot as plt
import numpy as np
import os
import random

# import a lot of things from keras:
# sequential model
from keras.models import Sequential

# layers
from keras.layers import Input, Dense, Dropout, Flatten, Conv2D, MaxPooling2D

# loss function
from keras.metrics import categorical_crossentropy

# callback functions
from keras.callbacks import ReduceLROnPlateau, EarlyStopping

# convert data to categorial vector representation
from keras.utils import to_categorical

# nice progress bar for loading data
from tqdm.notebook import tqdm

# helper function for train/test split
from sklearn.model_selection import train_test_split

# import confusion matrix helper function
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# import pre-trained model
from keras.applications.vgg16 import VGG16

# include only those gestures
CONDITIONS = ['like', 'stop']

# image size
IMG_SIZE = 64
SIZE = (IMG_SIZE, IMG_SIZE)

# number of color channels we want to use
# set to 1 to convert to grayscale
# set to 3 to use color images
COLOR_CHANNELS = 3
```

In [8]:

```
annotations = dict()

for condition in CONDITIONS:
    with open(f'_annotations/{condition}.json') as f:
        annotations[condition] = json.load(f)
```

In [9]:

```
def preprocess_image(img):
    if COLOR_CHANNELS == 1:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_resized = cv2.resize(img, SIZE)
    return img_resized
```

In [10]:

```
# load images and annotations
```

In [11]:

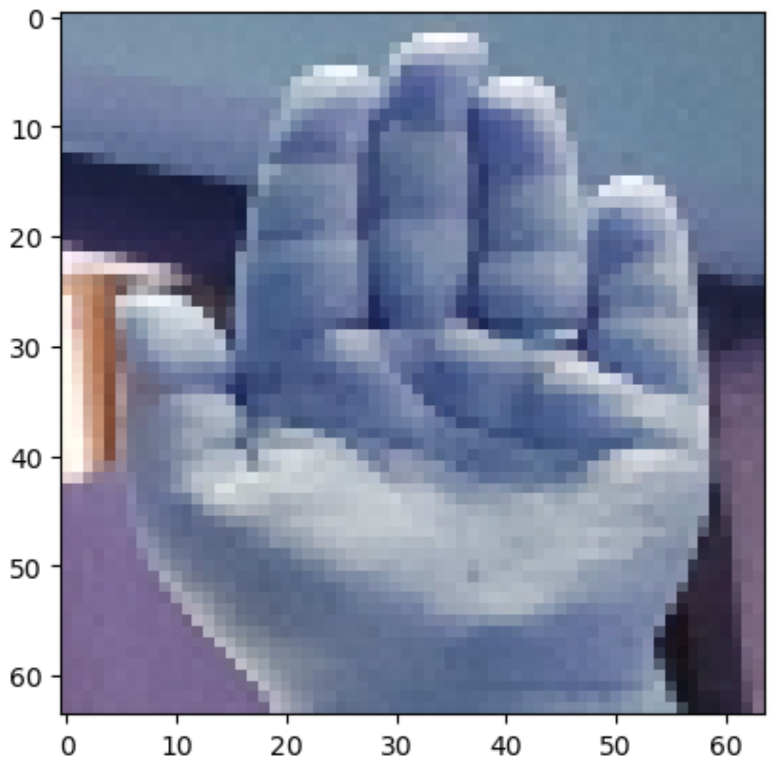
```python
images = [] # stores actual image data
labels = [] # stores labels (as integer - because this is what our network ne
label_names = [] # maps label ints to their actual categories so we can under

# loop over all conditions
# loop over all files in the condition's directory
# read the image and corresponding annotation
# crop image to the region of interest
# preprocess image
# store preprocessed image and label in corresponding lists
for condition in CONDITIONS:
    for filename in tqdm(os.listdir(condition)):
        # extract unique ID from file name
        UID = filename.split('.')[0]
        img = cv2.imread(f'{condition}/{filename}')

        # get annotation from the dict we loaded earlier
        try:
            annotation = annotations[condition][UID]
        except Exception as e:
            print(e)
            continue

        # iterate over all hands annotated in the image
        for i, bbox in enumerate(annotation['bboxes']):
            # annotated bounding boxes are in the range from 0 to 1
            # therefore we have to scale them to the image size
            x1 = int(bbox[0] * img.shape[1])
            y1 = int(bbox[1] * img.shape[0])
            w = int(bbox[2] * img.shape[1])
            h = int(bbox[3] * img.shape[0])
            x2 = x1 + w
            y2 = y1 + h

            # crop image to the bounding box and apply pre-processing
            crop = img[y1:y2, x1:x2]
            preprocessed = preprocess_image(crop)

            # get the annotated hand's label
            # if we have not seen this label yet, add it to the list of label
            label = annotation['labels'][i]
            if label not in label_names:
                label_names.append(label)

            label_index = label_names.index(label)

            images.append(preprocessed)
            labels.append(label_index)
```

```
  0%|          | 0/250 [00:00<?, ?it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

In [12]:

```python
plt.imshow(random.sample(images, 1)[0])
```

Out[12]: <matplotlib.image.AxesImage at 0x168b14c3d30>



In [13]:
```
# split data set into train and test
```

In [14]:
```
X_train, X_test, y_train, y_test = train_test_split(images, labels, test_size

print(len(X_train))
print(len(X_test))
print(len(y_train))
print(len(y_test))
```

```
512
128
512
128
```

In [15]:
```
X_train = np.array(X_train).astype('float32')
X_train = X_train / 255.

X_test = np.array(X_test).astype('float32')
X_test = X_test / 255.

y_train_one_hot = to_categorical(y_train)
y_test_one_hot = to_categorical(y_test)

train_label = y_train_one_hot
test_label = y_test_one_hot

X_train = X_train.reshape(-1, IMG_SIZE, IMG_SIZE, COLOR_CHANNELS)
X_test = X_test.reshape(-1, IMG_SIZE, IMG_SIZE, COLOR_CHANNELS)

print(X_train.shape, X_test.shape, train_label.shape, test_label.shape)
```

```
(512, 64, 64, 3) (128, 64, 64, 3) (512, 3) (128, 3)
```

batch-size = 1

In [56]:

```python
# variables for hyperparameters
batch_size = 1
epochs = 50
num_classes = len(label_names)
activation = 'relu'
activation_conv = 'LeakyReLU'  # LeakyReLU
layer_count = 2
num_neurons = 64

# define model structure
# with keras, we can use a model's add() function to add layers to the networ
model = Sequential()

# data augmentation (this can also be done beforehand - but don't augment the
model.add(RandomFlip('horizontal'))
model.add(RandomContrast(0.1))
#model.add(RandomBrightness(0.1))
#model.add(RandomRotation(0.2))

# first, we add some convolution layers followed by max pooling
model.add(Conv2D(64, kernel_size=(9, 9), activation=activation_conv, input_sh
model.add(MaxPooling2D(pool_size=(4, 4), padding='same'))

model.add(Conv2D(32, (5, 5), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), padding='same'))

model.add(Conv2D(32, (3, 3), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# dropout layers can drop part of the data during each epoch - this prevents
model.add(Dropout(0.2))

# after the convolution layers, we have to flatten the data so it can be fed
model.add(Flatten())

# add some fully connected layers ("Dense")
for i in range(layer_count - 1):
    model.add(Dense(num_neurons, activation=activation))

model.add(Dense(num_neurons, activation=activation))

# for classification, the last layer has to use the softmax activation functi
model.add(Dense(num_classes, activation='softmax'))

# specify loss function, optimizer and evaluation metrics
# for classification, categorial crossentropy is used as a loss function
# use the adam optimizer unless you have a good reason not to
model.compile(loss=categorical_crossentropy, optimizer="adam", metrics=['accu

# define callback functions that react to the model's behavior during trainin
# in this example, we reduce the learning rate once we get stuck and early st
# to cancel the training if there are no improvements for a certain amount of
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2, min
stop_early = EarlyStopping(monitor='val_loss', patience=3)
```

In [57]:

```
history = model.fit(
    X_train,
    train_label,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(X_test, test_label),
    callbacks=[reduce_lr, stop_early]
)
```

```
Epoch 1/50
512/512 [==============================] - 4s 7ms/step - loss: 1.1177 - accur
acy: 0.4180 - val_loss: 1.1733 - val_accuracy: 0.3516 - lr: 0.0010
Epoch 2/50
512/512 [==============================] - 4s 7ms/step - loss: 1.1199 - accur
acy: 0.4062 - val_loss: 1.0001 - val_accuracy: 0.4141 - lr: 0.0010
Epoch 3/50
512/512 [==============================] - 4s 7ms/step - loss: 1.0224 - accur
acy: 0.4395 - val_loss: 0.9439 - val_accuracy: 0.5234 - lr: 0.0010
Epoch 4/50
512/512 [==============================] - 4s 7ms/step - loss: 0.9143 - accur
acy: 0.5371 - val_loss: 0.7684 - val_accuracy: 0.6094 - lr: 0.0010
Epoch 5/50
512/512 [==============================] - 4s 7ms/step - loss: 0.7321 - accur
acy: 0.6797 - val_loss: 0.5233 - val_accuracy: 0.7734 - lr: 0.0010
Epoch 6/50
512/512 [==============================] - 4s 7ms/step - loss: 0.6254 - accur
acy: 0.7480 - val_loss: 0.3798 - val_accuracy: 0.8516 - lr: 0.0010
Epoch 7/50
512/512 [==============================] - 4s 7ms/step - loss: 0.3778 - accur
acy: 0.8691 - val_loss: 0.2257 - val_accuracy: 0.9062 - lr: 0.0010
Epoch 8/50
512/512 [==============================] - 4s 7ms/step - loss: 0.4100 - accur
acy: 0.8770 - val_loss: 0.2874 - val_accuracy: 0.9062 - lr: 0.0010
Epoch 9/50
512/512 [==============================] - 4s 7ms/step - loss: 0.2520 - accur
acy: 0.9258 - val_loss: 0.2081 - val_accuracy: 0.9375 - lr: 0.0010
Epoch 10/50
512/512 [==============================] - 4s 7ms/step - loss: 0.2670 - accur
acy: 0.9180 - val_loss: 0.2162 - val_accuracy: 0.9141 - lr: 0.0010
Epoch 11/50
512/512 [==============================] - 4s 7ms/step - loss: 0.2577 - accur
acy: 0.9102 - val_loss: 0.2674 - val_accuracy: 0.8984 - lr: 0.0010
Epoch 12/50
512/512 [==============================] - 4s 7ms/step - loss: 0.1194 - accur
acy: 0.9609 - val_loss: 0.2019 - val_accuracy: 0.9297 - lr: 2.0000e-04
Epoch 13/50
512/512 [==============================] - 4s 7ms/step - loss: 0.1080 - accur
acy: 0.9668 - val_loss: 0.1371 - val_accuracy: 0.9609 - lr: 2.0000e-04
Epoch 14/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0676 - accur
acy: 0.9688 - val_loss: 0.1368 - val_accuracy: 0.9609 - lr: 2.0000e-04
Epoch 15/50
512/512 [==============================] - 4s 7ms/step - loss: 0.1089 - accur
acy: 0.9766 - val_loss: 0.1755 - val_accuracy: 0.9453 - lr: 2.0000e-04
Epoch 16/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0696 - accur
acy: 0.9766 - val_loss: 0.1497 - val_accuracy: 0.9609 - lr: 2.0000e-04
Epoch 17/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0456 - accur
acy: 0.9844 - val_loss: 0.1305 - val_accuracy: 0.9609 - lr: 1.0000e-04
Epoch 18/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0525 - accur
acy: 0.9883 - val_loss: 0.1360 - val_accuracy: 0.9609 - lr: 1.0000e-04
Epoch 19/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0450 - accur
acy: 0.9863 - val_loss: 0.1203 - val_accuracy: 0.9609 - lr: 1.0000e-04
Epoch 20/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0362 - accur
acy: 0.9922 - val_loss: 0.1547 - val_accuracy: 0.9609 - lr: 1.0000e-04
Epoch 21/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0411 - accur
acy: 0.9844 - val_loss: 0.1301 - val_accuracy: 0.9688 - lr: 1.0000e-04
Epoch 22/50
512/512 [==============================] - 4s 7ms/step - loss: 0.0392 - accur
acy: 0.9883 - val_loss: 0.1394 - val_accuracy: 0.9688 - lr: 1.0000e-04
```

In [59]:

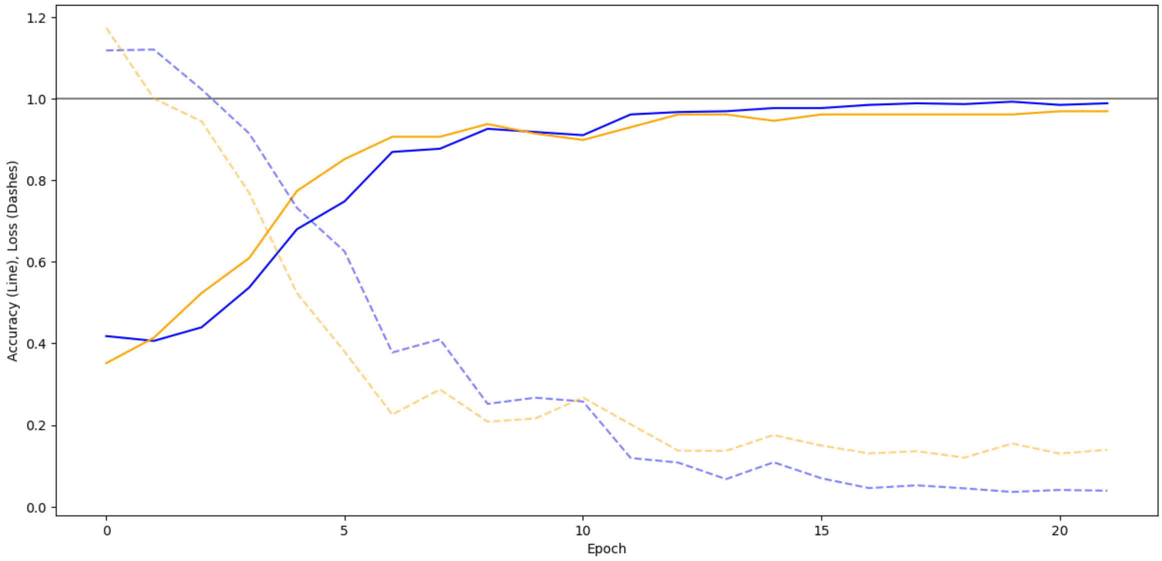
```
# plot accuracy and loss of the training process
```

In [60]:

```
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

fig = plt.figure(figsize=(15, 7))
ax = plt.gca()

ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy (Line), Loss (Dashes)')

ax.axhline(1, color='gray')

plt.plot(accuracy, color='blue')
plt.plot(val_accuracy, color='orange')
plt.plot(loss, '--', color='blue', alpha=0.5)
plt.plot(val_loss, '--', color='orange', alpha=0.5)
```

Out[60]:

```
[<matplotlib.lines.Line2D at 0x168cafab0a0>]
```



batch-size = 8

In [65]:

```
# variables for hyperparameters
batch_size = 8
epochs = 50
num_classes = len(label_names)
activation = 'relu'
activation_conv = 'LeakyReLU'  # LeakyReLU
layer_count = 2
num_neurons = 64

# define model structure
# with keras, we can use a model's add() function to add layers to the networ
model = Sequential()

# data augmentation (this can also be done beforehand - but don't augment the
model.add(RandomFlip('horizontal'))
model.add(RandomContrast(0.1))
#model.add(RandomBrightness(0.1))
#model.add(RandomRotation(0.2))

# first, we add some convolution layers followed by max pooling
model.add(Conv2D(64, kernel_size=(9, 9), activation=activation_conv, input_sh
model.add(MaxPooling2D(pool_size=(4, 4), padding='same'))

model.add(Conv2D(32, (5, 5), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), padding='same'))

model.add(Conv2D(32, (3, 3), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# dropout layers can drop part of the data during each epoch - this prevents
model.add(Dropout(0.2))

# after the convolution layers, we have to flatten the data so it can be fed
model.add(Flatten())

# add some fully connected layers ("Dense")
for i in range(layer_count - 1):
    model.add(Dense(num_neurons, activation=activation))

model.add(Dense(num_neurons, activation=activation))

# for classification, the last layer has to use the softmax activation functi
model.add(Dense(num_classes, activation='softmax'))

# specify loss function, optimizer and evaluation metrics
# for classification, categorial crossentropy is used as a loss function
# use the adam optimizer unless you have a good reason not to
model.compile(loss=categorical_crossentropy, optimizer="adam", metrics=['accu

# define callback functions that react to the model's behavior during trainin
# in this example, we reduce the learning rate once we get stuck and early st
# to cancel the training if there are no improvements for a certain amount of
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2, min
stop_early = EarlyStopping(monitor='val_loss', patience=3)
```

In [66]:

```
history = model.fit(
    X_train,
    train_label,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(X_test, test_label),
    callbacks=[reduce_lr, stop_early]
)
```

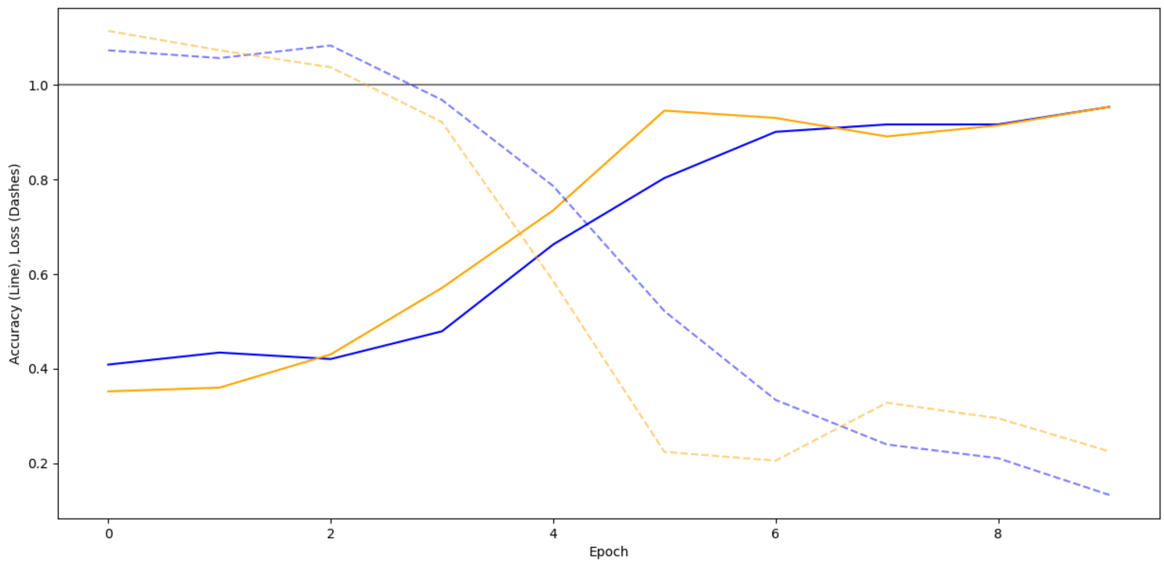
```
Epoch 1/50
64/64 [==============================] - 2s 26ms/step - loss: 1.0727 - accura
cy: 0.4082 - val_loss: 1.1138 - val_accuracy: 0.3516 - lr: 0.0010
Epoch 2/50
64/64 [==============================] - 2s 25ms/step - loss: 1.0564 - accura
cy: 0.4336 - val_loss: 1.0729 - val_accuracy: 0.3594 - lr: 0.0010
Epoch 3/50
64/64 [==============================] - 2s 26ms/step - loss: 1.0829 - accura
cy: 0.4199 - val_loss: 1.0369 - val_accuracy: 0.4297 - lr: 0.0010
Epoch 4/50
64/64 [==============================] - 2s 25ms/step - loss: 0.9679 - accura
cy: 0.4785 - val_loss: 0.9212 - val_accuracy: 0.5703 - lr: 0.0010
Epoch 5/50
64/64 [==============================] - 2s 25ms/step - loss: 0.7858 - accura
cy: 0.6621 - val_loss: 0.5848 - val_accuracy: 0.7344 - lr: 0.0010
Epoch 6/50
64/64 [==============================] - 2s 26ms/step - loss: 0.5212 - accura
cy: 0.8027 - val_loss: 0.2236 - val_accuracy: 0.9453 - lr: 0.0010
Epoch 7/50
64/64 [==============================] - 2s 25ms/step - loss: 0.3334 - accura
cy: 0.9004 - val_loss: 0.2051 - val_accuracy: 0.9297 - lr: 0.0010
Epoch 8/50
64/64 [==============================] - 2s 25ms/step - loss: 0.2391 - accura
cy: 0.9160 - val_loss: 0.3276 - val_accuracy: 0.8906 - lr: 0.0010
Epoch 9/50
64/64 [==============================] - 2s 25ms/step - loss: 0.2101 - accura
cy: 0.9160 - val_loss: 0.2947 - val_accuracy: 0.9141 - lr: 0.0010
Epoch 10/50
64/64 [==============================] - 2s 24ms/step - loss: 0.1325 - accura
cy: 0.9531 - val_loss: 0.2246 - val_accuracy: 0.9531 - lr: 2.0000e-04
```

In [68]:

```
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

fig = plt.figure(figsize=(15, 7))
ax = plt.gca()

ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy (Line), Loss (Dashes)')

ax.axhline(1, color='gray')

plt.plot(accuracy, color='blue')
plt.plot(val_accuracy, color='orange')
plt.plot(loss, '--', color='blue', alpha=0.5)
plt.plot(val_loss, '--', color='orange', alpha=0.5)
```

Out[68]:

```
[<matplotlib.lines.Line2D at 0x168cb994d00>]
```



batch-size=8

In [71]:

```
# variables for hyperparameters
batch_size = 32
epochs = 50
num_classes = len(label_names)
activation = 'relu'
activation_conv = 'LeakyReLU'  # LeakyReLU
layer_count = 2
num_neurons = 64

# define model structure
# with keras, we can use a model's add() function to add layers to the networ
model = Sequential()

# data augmentation (this can also be done beforehand - but don't augment the
model.add(RandomFlip('horizontal'))
model.add(RandomContrast(0.1))
#model.add(RandomBrightness(0.1))
#model.add(RandomRotation(0.2))

# first, we add some convolution layers followed by max pooling
model.add(Conv2D(64, kernel_size=(9, 9), activation=activation_conv, input_sh
model.add(MaxPooling2D(pool_size=(4, 4), padding='same'))

model.add(Conv2D(32, (5, 5), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), padding='same'))

model.add(Conv2D(32, (3, 3), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# dropout layers can drop part of the data during each epoch - this prevents
model.add(Dropout(0.2))

# after the convolution layers, we have to flatten the data so it can be fed
model.add(Flatten())

# add some fully connected layers ("Dense")
for i in range(layer_count - 1):
    model.add(Dense(num_neurons, activation=activation))

model.add(Dense(num_neurons, activation=activation))

# for classification, the last layer has to use the softmax activation functi
model.add(Dense(num_classes, activation='softmax'))

# specify loss function, optimizer and evaluation metrics
# for classification, categorial crossentropy is used as a loss function
# use the adam optimizer unless you have a good reason not to
model.compile(loss=categorical_crossentropy, optimizer="adam", metrics=['accu

# define callback functions that react to the model's behavior during trainin
# in this example, we reduce the learning rate once we get stuck and early st
# to cancel the training if there are no improvements for a certain amount of
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2, min
stop_early = EarlyStopping(monitor='val_loss', patience=3)
```

In [72]:
```
history = model.fit(
    X_train,
    train_label,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(X_test, test_label),
    callbacks=[reduce_lr, stop_early]
)
```

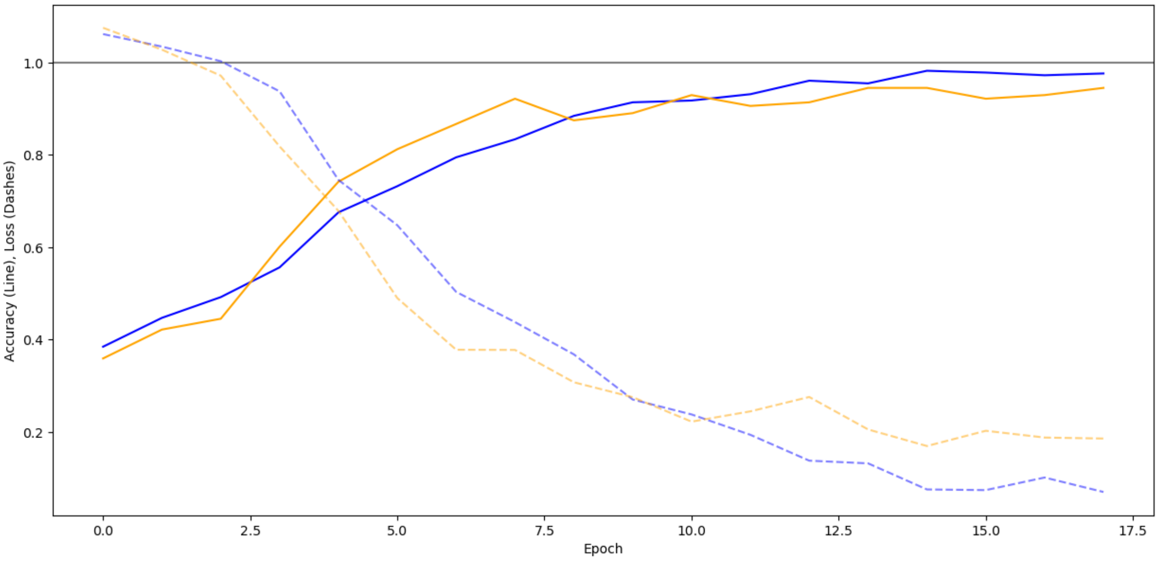
```
Epoch 1/50
16/16 [==============================] - 2s 87ms/step - loss: 1.0619 - accura
cy: 0.3848 - val_loss: 1.0754 - val_accuracy: 0.3594 - lr: 0.0010
Epoch 2/50
16/16 [==============================] - 1s 82ms/step - loss: 1.0347 - accura
cy: 0.4473 - val_loss: 1.0281 - val_accuracy: 0.4219 - lr: 0.0010
Epoch 3/50
16/16 [==============================] - 1s 81ms/step - loss: 1.0031 - accura
cy: 0.4922 - val_loss: 0.9718 - val_accuracy: 0.4453 - lr: 0.0010
Epoch 4/50
16/16 [==============================] - 1s 78ms/step - loss: 0.9378 - accura
cy: 0.5566 - val_loss: 0.8185 - val_accuracy: 0.6016 - lr: 0.0010
Epoch 5/50
16/16 [==============================] - 1s 79ms/step - loss: 0.7459 - accura
cy: 0.6758 - val_loss: 0.6788 - val_accuracy: 0.7422 - lr: 0.0010
Epoch 6/50
16/16 [==============================] - 1s 78ms/step - loss: 0.6478 - accura
cy: 0.7324 - val_loss: 0.4900 - val_accuracy: 0.8125 - lr: 0.0010
Epoch 7/50
16/16 [==============================] - 1s 78ms/step - loss: 0.5035 - accura
cy: 0.7949 - val_loss: 0.3782 - val_accuracy: 0.8672 - lr: 0.0010
Epoch 8/50
16/16 [==============================] - 1s 78ms/step - loss: 0.4380 - accura
cy: 0.8340 - val_loss: 0.3778 - val_accuracy: 0.9219 - lr: 0.0010
Epoch 9/50
16/16 [==============================] - 1s 78ms/step - loss: 0.3682 - accura
cy: 0.8848 - val_loss: 0.3077 - val_accuracy: 0.8750 - lr: 0.0010
Epoch 10/50
16/16 [==============================] - 1s 78ms/step - loss: 0.2703 - accura
cy: 0.9141 - val_loss: 0.2754 - val_accuracy: 0.8906 - lr: 0.0010
Epoch 11/50
16/16 [==============================] - 1s 79ms/step - loss: 0.2381 - accura
cy: 0.9180 - val_loss: 0.2225 - val_accuracy: 0.9297 - lr: 0.0010
Epoch 12/50
16/16 [==============================] - 1s 79ms/step - loss: 0.1939 - accura
cy: 0.9316 - val_loss: 0.2448 - val_accuracy: 0.9062 - lr: 0.0010
Epoch 13/50
16/16 [==============================] - 1s 81ms/step - loss: 0.1379 - accura
cy: 0.9609 - val_loss: 0.2759 - val_accuracy: 0.9141 - lr: 0.0010
Epoch 14/50
16/16 [==============================] - 1s 81ms/step - loss: 0.1321 - accura
cy: 0.9551 - val_loss: 0.2056 - val_accuracy: 0.9453 - lr: 2.0000e-04
Epoch 15/50
16/16 [==============================] - 1s 81ms/step - loss: 0.0756 - accura
cy: 0.9824 - val_loss: 0.1696 - val_accuracy: 0.9453 - lr: 2.0000e-04
Epoch 16/50
16/16 [==============================] - 1s 84ms/step - loss: 0.0742 - accura
cy: 0.9785 - val_loss: 0.2027 - val_accuracy: 0.9219 - lr: 2.0000e-04
Epoch 17/50
16/16 [==============================] - 1s 79ms/step - loss: 0.1015 - accura
cy: 0.9727 - val_loss: 0.1880 - val_accuracy: 0.9297 - lr: 2.0000e-04
Epoch 18/50
16/16 [==============================] - 1s 80ms/step - loss: 0.0701 - accura
cy: 0.9766 - val_loss: 0.1859 - val_accuracy: 0.9453 - lr: 1.0000e-04
```

In [74]:

```
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

fig = plt.figure(figsize=(15, 7))
ax = plt.gca()

ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy (Line), Loss (Dashes)')

ax.axhline(1, color='gray')

plt.plot(accuracy, color='blue')
plt.plot(val_accuracy, color='orange')
plt.plot(loss, '--', color='blue', alpha=0.5)
plt.plot(val_loss, '--', color='orange', alpha=0.5)
```

Out[74]:

```
[<matplotlib.lines.Line2D at 0x168ca5a6320>]
```



batch-size = 128

In [77]:

```
# variables for hyperparameters
batch_size = 128
epochs = 50
num_classes = len(label_names)
activation = 'relu'
activation_conv = 'LeakyReLU'  # LeakyReLU
layer_count = 2
num_neurons = 64

# define model structure
# with keras, we can use a model's add() function to add layers to the networ
model = Sequential()

# data augmentation (this can also be done beforehand - but don't augment the
model.add(RandomFlip('horizontal'))
model.add(RandomContrast(0.1))
#model.add(RandomBrightness(0.1))
#model.add(RandomRotation(0.2))

# first, we add some convolution layers followed by max pooling
model.add(Conv2D(64, kernel_size=(9, 9), activation=activation_conv, input_sh
model.add(MaxPooling2D(pool_size=(4, 4), padding='same'))

model.add(Conv2D(32, (5, 5), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), padding='same'))

model.add(Conv2D(32, (3, 3), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# dropout layers can drop part of the data during each epoch - this prevents
model.add(Dropout(0.2))

# after the convolution layers, we have to flatten the data so it can be fed
model.add(Flatten())

# add some fully connected layers ("Dense")
for i in range(layer_count - 1):
    model.add(Dense(num_neurons, activation=activation))

model.add(Dense(num_neurons, activation=activation))

# for classification, the last layer has to use the softmax activation functi
model.add(Dense(num_classes, activation='softmax'))

# specify loss function, optimizer and evaluation metrics
# for classification, categorial crossentropy is used as a loss function
# use the adam optimizer unless you have a good reason not to
model.compile(loss=categorical_crossentropy, optimizer="adam", metrics=['accu

# define callback functions that react to the model's behavior during trainin
# in this example, we reduce the learning rate once we get stuck and early st
# to cancel the training if there are no improvements for a certain amount of
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2, min
stop_early = EarlyStopping(monitor='val_loss', patience=3)
```

In [78]:

```
history = model.fit(
    X_train,
    train_label,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(X_test, test_label),
    callbacks=[reduce_lr, stop_early]
)
```

```
Epoch 1/50
4/4 [==============================] - 2s 324ms/step - loss: 1.0611 - accurac
y: 0.4160 - val_loss: 1.0422 - val_accuracy: 0.5234 - lr: 0.0010
Epoch 2/50
4/4 [==============================] - 1s 277ms/step - loss: 1.0064 - accurac
y: 0.4746 - val_loss: 1.0062 - val_accuracy: 0.5156 - lr: 0.0010
Epoch 3/50
4/4 [==============================] - 1s 277ms/step - loss: 0.9903 - accurac
y: 0.4824 - val_loss: 0.9555 - val_accuracy: 0.5000 - lr: 0.0010
Epoch 4/50
4/4 [==============================] - 1s 276ms/step - loss: 0.9413 - accurac
y: 0.5234 - val_loss: 0.9055 - val_accuracy: 0.5391 - lr: 0.0010
Epoch 5/50
4/4 [==============================] - 1s 276ms/step - loss: 0.9171 - accurac
y: 0.5645 - val_loss: 0.8883 - val_accuracy: 0.5938 - lr: 0.0010
Epoch 6/50
4/4 [==============================] - 1s 276ms/step - loss: 0.8493 - accurac
y: 0.5957 - val_loss: 0.8773 - val_accuracy: 0.5625 - lr: 0.0010
Epoch 7/50
4/4 [==============================] - 1s 277ms/step - loss: 0.7904 - accurac
y: 0.6465 - val_loss: 0.7092 - val_accuracy: 0.7344 - lr: 0.0010
Epoch 8/50
4/4 [==============================] - 1s 275ms/step - loss: 0.7176 - accurac
y: 0.6992 - val_loss: 0.6834 - val_accuracy: 0.7031 - lr: 0.0010
Epoch 9/50
4/4 [==============================] - 1s 278ms/step - loss: 0.6759 - accurac
y: 0.7305 - val_loss: 0.5685 - val_accuracy: 0.8203 - lr: 0.0010
Epoch 10/50
4/4 [==============================] - 1s 297ms/step - loss: 0.5691 - accurac
y: 0.7949 - val_loss: 0.5091 - val_accuracy: 0.7969 - lr: 0.0010
Epoch 11/50
4/4 [==============================] - 1s 292ms/step - loss: 0.4958 - accurac
y: 0.8125 - val_loss: 0.4762 - val_accuracy: 0.8125 - lr: 0.0010
Epoch 12/50
4/4 [==============================] - 1s 294ms/step - loss: 0.4584 - accurac
y: 0.8262 - val_loss: 0.4772 - val_accuracy: 0.8125 - lr: 0.0010
Epoch 13/50
4/4 [==============================] - 1s 298ms/step - loss: 0.5046 - accurac
y: 0.8184 - val_loss: 0.3908 - val_accuracy: 0.8438 - lr: 0.0010
Epoch 14/50
4/4 [==============================] - 1s 297ms/step - loss: 0.4406 - accurac
y: 0.8633 - val_loss: 0.3915 - val_accuracy: 0.8594 - lr: 0.0010
Epoch 15/50
4/4 [==============================] - 1s 303ms/step - loss: 0.3880 - accurac
y: 0.8711 - val_loss: 0.3907 - val_accuracy: 0.8672 - lr: 0.0010
Epoch 16/50
4/4 [==============================] - 1s 290ms/step - loss: 0.3822 - accurac
y: 0.8652 - val_loss: 0.3224 - val_accuracy: 0.8828 - lr: 0.0010
Epoch 17/50
4/4 [==============================] - 1s 282ms/step - loss: 0.3076 - accurac
y: 0.9102 - val_loss: 0.3211 - val_accuracy: 0.9141 - lr: 0.0010
Epoch 18/50
4/4 [==============================] - 1s 291ms/step - loss: 0.2823 - accurac
y: 0.9082 - val_loss: 0.3156 - val_accuracy: 0.8594 - lr: 0.0010
Epoch 19/50
4/4 [==============================] - 1s 285ms/step - loss: 0.2505 - accurac
y: 0.9121 - val_loss: 0.3068 - val_accuracy: 0.8984 - lr: 0.0010
Epoch 20/50
4/4 [==============================] - 1s 285ms/step - loss: 0.2283 - accurac
y: 0.9297 - val_loss: 0.2786 - val_accuracy: 0.8906 - lr: 0.0010
Epoch 21/50
4/4 [==============================] - 1s 282ms/step - loss: 0.2167 - accurac
y: 0.9316 - val_loss: 0.2866 - val_accuracy: 0.9219 - lr: 0.0010
Epoch 22/50
4/4 [==============================] - 1s 283ms/step - loss: 0.2159 - accurac
y: 0.9258 - val_loss: 0.2332 - val_accuracy: 0.9297 - lr: 0.0010
```

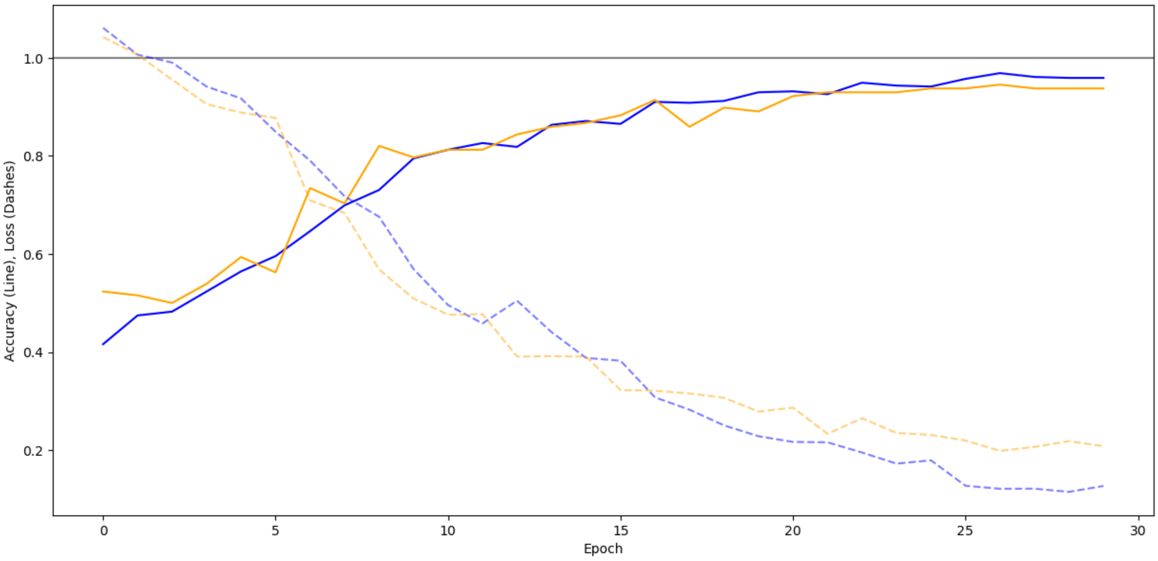
```
Epoch 23/50
4/4 [==============================] - 1s 282ms/step - loss: 0.1950 - accurac
y: 0.9492 - val_loss: 0.2649 - val_accuracy: 0.9297 - lr: 0.0010
Epoch 24/50
4/4 [==============================] - 1s 289ms/step - loss: 0.1724 - accurac
y: 0.9434 - val_loss: 0.2351 - val_accuracy: 0.9297 - lr: 0.0010
Epoch 25/50
4/4 [==============================] - 1s 289ms/step - loss: 0.1791 - accurac
y: 0.9414 - val_loss: 0.2310 - val_accuracy: 0.9375 - lr: 2.0000e-04
Epoch 26/50
4/4 [==============================] - 1s 286ms/step - loss: 0.1273 - accurac
y: 0.9570 - val_loss: 0.2196 - val_accuracy: 0.9375 - lr: 2.0000e-04
Epoch 27/50
4/4 [==============================] - 1s 280ms/step - loss: 0.1211 - accurac
y: 0.9688 - val_loss: 0.1984 - val_accuracy: 0.9453 - lr: 2.0000e-04
Epoch 28/50
4/4 [==============================] - 1s 281ms/step - loss: 0.1213 - accurac
y: 0.9609 - val_loss: 0.2066 - val_accuracy: 0.9375 - lr: 2.0000e-04
Epoch 29/50
4/4 [==============================] - 1s 290ms/step - loss: 0.1148 - accurac
y: 0.9590 - val_loss: 0.2182 - val_accuracy: 0.9375 - lr: 2.0000e-04
Epoch 30/50
4/4 [==============================] - 1s 275ms/step - loss: 0.1269 - accurac
y: 0.9590 - val_loss: 0.2081 - val_accuracy: 0.9375 - lr: 1.0000e-04
```

In [80]:

```python
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

fig = plt.figure(figsize=(15, 7))
ax = plt.gca()

ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy (Line), Loss (Dashes)')

ax.axhline(1, color='gray')

plt.plot(accuracy, color='blue')
plt.plot(val_accuracy, color='orange')
plt.plot(loss, '--', color='blue', alpha=0.5)
plt.plot(val_loss, '--', color='orange', alpha=0.5)
```

Out[80]:

```
[<matplotlib.lines.Line2D at 0x168caf54460>]
```



batch-size = 256

In [83]:

```
# variables for hyperparameters
batch_size = 256
epochs = 50
num_classes = len(label_names)
activation = 'relu'
activation_conv = 'LeakyReLU'  # LeakyReLU
layer_count = 2
num_neurons = 64

# define model structure
# with keras, we can use a model's add() function to add layers to the networ
model = Sequential()

# data augmentation (this can also be done beforehand - but don't augment the
model.add(RandomFlip('horizontal'))
model.add(RandomContrast(0.1))
#model.add(RandomBrightness(0.1))
#model.add(RandomRotation(0.2))

# first, we add some convolution layers followed by max pooling
model.add(Conv2D(64, kernel_size=(9, 9), activation=activation_conv, input_sh
model.add(MaxPooling2D(pool_size=(4, 4), padding='same'))

model.add(Conv2D(32, (5, 5), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), padding='same'))

model.add(Conv2D(32, (3, 3), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# dropout layers can drop part of the data during each epoch - this prevents
model.add(Dropout(0.2))

# after the convolution layers, we have to flatten the data so it can be fed
model.add(Flatten())

# add some fully connected layers ("Dense")
for i in range(layer_count - 1):
    model.add(Dense(num_neurons, activation=activation))

model.add(Dense(num_neurons, activation=activation))

# for classification, the last layer has to use the softmax activation functi
model.add(Dense(num_classes, activation='softmax'))

# specify loss function, optimizer and evaluation metrics
# for classification, categorial crossentropy is used as a loss function
# use the adam optimizer unless you have a good reason not to
model.compile(loss=categorical_crossentropy, optimizer="adam", metrics=['accu

# define callback functions that react to the model's behavior during trainin
# in this example, we reduce the learning rate once we get stuck and early st
# to cancel the training if there are no improvements for a certain amount of
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2, min
stop_early = EarlyStopping(monitor='val_loss', patience=3)
```

In [84]:

```
history = model.fit(
    X_train,
    train_label,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(X_test, test_label),
    callbacks=[reduce_lr, stop_early]
)
```

```
Epoch 1/50
2/2 [==============================] - 2s 736ms/step - loss: 1.0877 - accurac
y: 0.3984 - val_loss: 1.0569 - val_accuracy: 0.4219 - lr: 0.0010
Epoch 2/50
2/2 [==============================] - 1s 549ms/step - loss: 1.0488 - accurac
y: 0.3828 - val_loss: 1.0552 - val_accuracy: 0.5312 - lr: 0.0010
Epoch 3/50
2/2 [==============================] - 1s 542ms/step - loss: 1.0426 - accurac
y: 0.4219 - val_loss: 1.0488 - val_accuracy: 0.4531 - lr: 0.0010
Epoch 4/50
2/2 [==============================] - 1s 554ms/step - loss: 1.0274 - accurac
y: 0.4375 - val_loss: 1.0381 - val_accuracy: 0.4375 - lr: 0.0010
Epoch 5/50
2/2 [==============================] - 1s 556ms/step - loss: 1.0182 - accurac
y: 0.4805 - val_loss: 1.0049 - val_accuracy: 0.4922 - lr: 0.0010
Epoch 6/50
2/2 [==============================] - 1s 553ms/step - loss: 0.9867 - accurac
y: 0.5215 - val_loss: 0.9655 - val_accuracy: 0.5703 - lr: 0.0010
Epoch 7/50
2/2 [==============================] - 1s 548ms/step - loss: 0.9659 - accurac
y: 0.5391 - val_loss: 0.9676 - val_accuracy: 0.4922 - lr: 0.0010
Epoch 8/50
2/2 [==============================] - 1s 545ms/step - loss: 0.9538 - accurac
y: 0.5273 - val_loss: 0.8852 - val_accuracy: 0.5859 - lr: 0.0010
Epoch 9/50
2/2 [==============================] - 1s 554ms/step - loss: 0.9080 - accurac
y: 0.5684 - val_loss: 0.8534 - val_accuracy: 0.5859 - lr: 0.0010
Epoch 10/50
2/2 [==============================] - 1s 546ms/step - loss: 0.8657 - accurac
y: 0.5879 - val_loss: 0.7954 - val_accuracy: 0.6250 - lr: 0.0010
Epoch 11/50
2/2 [==============================] - 1s 544ms/step - loss: 0.7792 - accurac
y: 0.6445 - val_loss: 0.7571 - val_accuracy: 0.6641 - lr: 0.0010
Epoch 12/50
2/2 [==============================] - 1s 540ms/step - loss: 0.7998 - accurac
y: 0.6348 - val_loss: 0.7311 - val_accuracy: 0.6875 - lr: 0.0010
Epoch 13/50
2/2 [==============================] - 1s 567ms/step - loss: 0.7537 - accurac
y: 0.6680 - val_loss: 0.7021 - val_accuracy: 0.6641 - lr: 0.0010
Epoch 14/50
2/2 [==============================] - 1s 576ms/step - loss: 0.7128 - accurac
y: 0.7188 - val_loss: 0.7068 - val_accuracy: 0.7188 - lr: 0.0010
Epoch 15/50
2/2 [==============================] - 1s 576ms/step - loss: 0.6907 - accurac
y: 0.6836 - val_loss: 0.6329 - val_accuracy: 0.7109 - lr: 0.0010
Epoch 16/50
2/2 [==============================] - 1s 560ms/step - loss: 0.6478 - accurac
y: 0.7285 - val_loss: 0.5769 - val_accuracy: 0.7812 - lr: 0.0010
Epoch 17/50
2/2 [==============================] - 1s 584ms/step - loss: 0.6127 - accurac
y: 0.7539 - val_loss: 0.5277 - val_accuracy: 0.8125 - lr: 0.0010
Epoch 18/50
2/2 [==============================] - 1s 577ms/step - loss: 0.5834 - accurac
y: 0.7812 - val_loss: 0.4883 - val_accuracy: 0.8359 - lr: 0.0010
Epoch 19/50
2/2 [==============================] - 1s 548ms/step - loss: 0.5061 - accurac
y: 0.8223 - val_loss: 0.4686 - val_accuracy: 0.8281 - lr: 0.0010
Epoch 20/50
2/2 [==============================] - 1s 543ms/step - loss: 0.5025 - accurac
y: 0.8086 - val_loss: 0.4145 - val_accuracy: 0.8438 - lr: 0.0010
Epoch 21/50
2/2 [==============================] - 1s 573ms/step - loss: 0.4456 - accurac
y: 0.8340 - val_loss: 0.3953 - val_accuracy: 0.8594 - lr: 0.0010
Epoch 22/50
2/2 [==============================] - 1s 527ms/step - loss: 0.4441 - accurac
y: 0.8320 - val_loss: 0.3661 - val_accuracy: 0.8672 - lr: 0.0010
```

```
Epoch 23/50
2/2 [==============================] - 1s 535ms/step - loss: 0.4255 - accurac
y: 0.8496 - val_loss: 0.3752 - val_accuracy: 0.8594 - lr: 0.0010
Epoch 24/50
2/2 [==============================] - 1s 587ms/step - loss: 0.3920 - accurac
y: 0.8555 - val_loss: 0.3274 - val_accuracy: 0.9062 - lr: 0.0010
Epoch 25/50
2/2 [==============================] - 1s 598ms/step - loss: 0.3702 - accurac
y: 0.8828 - val_loss: 0.3610 - val_accuracy: 0.8750 - lr: 0.0010
Epoch 26/50
2/2 [==============================] - 1s 570ms/step - loss: 0.3485 - accurac
y: 0.8926 - val_loss: 0.2923 - val_accuracy: 0.9062 - lr: 0.0010
Epoch 27/50
2/2 [==============================] - 1s 556ms/step - loss: 0.3363 - accurac
y: 0.8887 - val_loss: 0.3468 - val_accuracy: 0.8750 - lr: 0.0010
Epoch 28/50
2/2 [==============================] - 1s 585ms/step - loss: 0.3261 - accurac
y: 0.8926 - val_loss: 0.2894 - val_accuracy: 0.9062 - lr: 0.0010
Epoch 29/50
2/2 [==============================] - 1s 568ms/step - loss: 0.2866 - accurac
y: 0.9062 - val_loss: 0.2973 - val_accuracy: 0.9062 - lr: 0.0010
Epoch 30/50
2/2 [==============================] - 1s 567ms/step - loss: 0.3072 - accurac
y: 0.8965 - val_loss: 0.3249 - val_accuracy: 0.8828 - lr: 0.0010
Epoch 31/50
2/2 [==============================] - 1s 567ms/step - loss: 0.3032 - accurac
y: 0.8789 - val_loss: 0.2821 - val_accuracy: 0.9141 - lr: 2.0000e-04
Epoch 32/50
2/2 [==============================] - 1s 555ms/step - loss: 0.2513 - accurac
y: 0.9043 - val_loss: 0.2557 - val_accuracy: 0.9375 - lr: 2.0000e-04
Epoch 33/50
2/2 [==============================] - 1s 567ms/step - loss: 0.2593 - accurac
y: 0.9082 - val_loss: 0.2917 - val_accuracy: 0.9141 - lr: 2.0000e-04
Epoch 34/50
2/2 [==============================] - 1s 571ms/step - loss: 0.2722 - accurac
y: 0.9180 - val_loss: 0.2593 - val_accuracy: 0.9297 - lr: 2.0000e-04
Epoch 35/50
2/2 [==============================] - 1s 544ms/step - loss: 0.2092 - accurac
y: 0.9414 - val_loss: 0.2520 - val_accuracy: 0.9141 - lr: 1.0000e-04
Epoch 36/50
2/2 [==============================] - 1s 543ms/step - loss: 0.2255 - accurac
y: 0.9297 - val_loss: 0.2538 - val_accuracy: 0.9141 - lr: 1.0000e-04
Epoch 37/50
2/2 [==============================] - 1s 544ms/step - loss: 0.2475 - accurac
y: 0.9062 - val_loss: 0.2573 - val_accuracy: 0.9219 - lr: 1.0000e-04
Epoch 38/50
2/2 [==============================] - 1s 545ms/step - loss: 0.2058 - accurac
y: 0.9375 - val_loss: 0.2519 - val_accuracy: 0.9219 - lr: 1.0000e-04
Epoch 39/50
2/2 [==============================] - 1s 625ms/step - loss: 0.2225 - accurac
y: 0.9258 - val_loss: 0.2477 - val_accuracy: 0.9219 - lr: 1.0000e-04
Epoch 40/50
2/2 [==============================] - 1s 593ms/step - loss: 0.2200 - accurac
y: 0.9355 - val_loss: 0.2422 - val_accuracy: 0.9297 - lr: 1.0000e-04
Epoch 41/50
2/2 [==============================] - 1s 607ms/step - loss: 0.1999 - accurac
y: 0.9277 - val_loss: 0.2378 - val_accuracy: 0.9375 - lr: 1.0000e-04
Epoch 42/50
2/2 [==============================] - 1s 622ms/step - loss: 0.2028 - accurac
y: 0.9336 - val_loss: 0.2352 - val_accuracy: 0.9375 - lr: 1.0000e-04
Epoch 43/50
2/2 [==============================] - 1s 570ms/step - loss: 0.2120 - accurac
y: 0.9355 - val_loss: 0.2332 - val_accuracy: 0.9609 - lr: 1.0000e-04
Epoch 44/50
2/2 [==============================] - 1s 630ms/step - loss: 0.2118 - accurac
y: 0.9297 - val_loss: 0.2339 - val_accuracy: 0.9453 - lr: 1.0000e-04
```

```
Epoch 45/50
2/2 [==============================] - 1s 680ms/step - loss: 0.2214 - accurac
y: 0.9316 - val_loss: 0.2336 - val_accuracy: 0.9375 - lr: 1.0000e-04
Epoch 46/50
2/2 [==============================] - 1s 591ms/step - loss: 0.2040 - accurac
y: 0.9492 - val_loss: 0.2316 - val_accuracy: 0.9531 - lr: 1.0000e-04
Epoch 47/50
2/2 [==============================] - 1s 576ms/step - loss: 0.1991 - accurac
y: 0.9453 - val_loss: 0.2312 - val_accuracy: 0.9531 - lr: 1.0000e-04
Epoch 48/50
2/2 [==============================] - 1s 584ms/step - loss: 0.1921 - accurac
y: 0.9414 - val_loss: 0.2317 - val_accuracy: 0.9453 - lr: 1.0000e-04
Epoch 49/50
2/2 [==============================] - 1s 604ms/step - loss: 0.1790 - accurac
y: 0.9473 - val_loss: 0.2338 - val_accuracy: 0.9453 - lr: 1.0000e-04
Epoch 50/50
2/2 [==============================] - 1s 650ms/step - loss: 0.2017 - accurac
y: 0.9316 - val_loss: 0.2289 - val_accuracy: 0.9531 - lr: 1.0000e-04
```

In [85]:

```
model.summary()
```

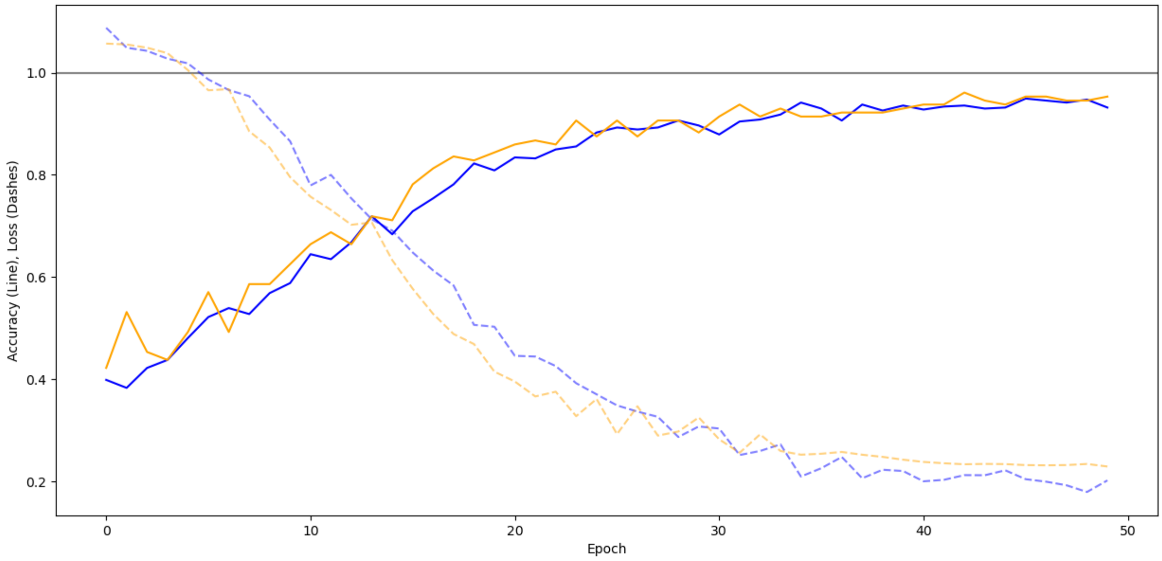
```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 random_flip_15 (RandomFlip)  (None, 64, 64, 3)        0         
                                                                 
 random_contrast_15 (RandomC  (None, 64, 64, 3)        0         
 ontrast)                                                        
                                                                 
 conv2d_45 (Conv2D)          (None, 64, 64, 64)        15616     
                                                                 
 max_pooling2d_45 (MaxPoolin  (None, 16, 16, 64)       0         
 g2D)                                                            
                                                                 
 conv2d_46 (Conv2D)          (None, 16, 16, 32)        51232     
                                                                 
 max_pooling2d_46 (MaxPoolin  (None, 6, 6, 32)         0         
 g2D)                                                            
                                                                 
 conv2d_47 (Conv2D)          (None, 6, 6, 32)          9248      
                                                                 
 max_pooling2d_47 (MaxPoolin  (None, 3, 3, 32)         0         
 g2D)                                                            
                                                                 
 dropout_15 (Dropout)        (None, 3, 3, 32)          0         
                                                                 
 flatten_15 (Flatten)        (None, 288)               0         
                                                                 
 dense_45 (Dense)            (None, 64)                18496     
                                                                 
 dense_46 (Dense)            (None, 64)                4160      
                                                                 
 dense_47 (Dense)            (None, 3)                 195       
                                                                 
=================================================================
Total params: 98,947
Trainable params: 98,947
Non-trainable params: 0
_________________________________________________________________
```

In [86]:

```
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

fig = plt.figure(figsize=(15, 7))
ax = plt.gca()

ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy (Line), Loss (Dashes)')

ax.axhline(1, color='gray')

plt.plot(accuracy, color='blue')
plt.plot(val_accuracy, color='orange')
plt.plot(loss, '--', color='blue', alpha=0.5)
plt.plot(val_loss, '--', color='orange', alpha=0.5)
```

Out[86]:

```
[<matplotlib.lines.Line2D at 0x168ca2eaf50>]
```



batch-size = 512 (max)

In [89]:

```
# variables for hyperparameters
batch_size = 512
epochs = 50
num_classes = len(label_names)
activation = 'relu'
activation_conv = 'LeakyReLU'  # LeakyReLU
layer_count = 2
num_neurons = 64

# define model structure
# with keras, we can use a model's add() function to add layers to the networ
model = Sequential()

# data augmentation (this can also be done beforehand - but don't augment the
model.add(RandomFlip('horizontal'))
model.add(RandomContrast(0.1))
#model.add(RandomBrightness(0.1))
#model.add(RandomRotation(0.2))

# first, we add some convolution layers followed by max pooling
model.add(Conv2D(64, kernel_size=(9, 9), activation=activation_conv, input_sh
model.add(MaxPooling2D(pool_size=(4, 4), padding='same'))

model.add(Conv2D(32, (5, 5), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(3, 3), padding='same'))

model.add(Conv2D(32, (3, 3), activation=activation_conv, padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# dropout layers can drop part of the data during each epoch - this prevents
model.add(Dropout(0.2))

# after the convolution layers, we have to flatten the data so it can be fed
model.add(Flatten())

# add some fully connected layers ("Dense")
for i in range(layer_count - 1):
    model.add(Dense(num_neurons, activation=activation))

model.add(Dense(num_neurons, activation=activation))

# for classification, the last layer has to use the softmax activation functi
model.add(Dense(num_classes, activation='softmax'))

# specify loss function, optimizer and evaluation metrics
# for classification, categorial crossentropy is used as a loss function
# use the adam optimizer unless you have a good reason not to
model.compile(loss=categorical_crossentropy, optimizer="adam", metrics=['accu

# define callback functions that react to the model's behavior during trainin
# in this example, we reduce the learning rate once we get stuck and early st
# to cancel the training if there are no improvements for a certain amount of
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2, min
stop_early = EarlyStopping(monitor='val_loss', patience=3)
```

In [90]:

```
history = model.fit(
    X_train,
    train_label,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(X_test, test_label),
    callbacks=[reduce_lr, stop_early]
)
```

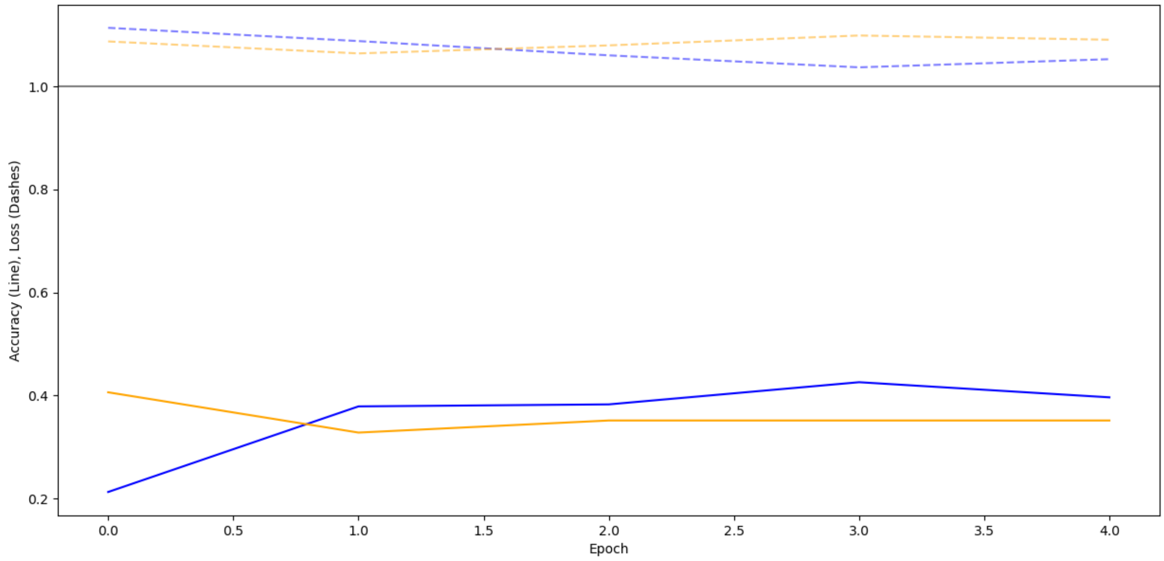
```
Epoch 1/50
1/1 [==============================] - 2s 2s/step - loss: 1.1132 - accuracy:
0.2129 - val_loss: 1.0868 - val_accuracy: 0.4062 - lr: 0.0010
Epoch 2/50
1/1 [==============================] - 1s 1s/step - loss: 1.0875 - accuracy:
0.3789 - val_loss: 1.0636 - val_accuracy: 0.3281 - lr: 0.0010
Epoch 3/50
1/1 [==============================] - 1s 1s/step - loss: 1.0598 - accuracy:
0.3828 - val_loss: 1.0792 - val_accuracy: 0.3516 - lr: 0.0010
Epoch 4/50
1/1 [==============================] - 1s 1s/step - loss: 1.0366 - accuracy:
0.4258 - val_loss: 1.0984 - val_accuracy: 0.3516 - lr: 0.0010
Epoch 5/50
1/1 [==============================] - 1s 1s/step - loss: 1.0524 - accuracy:
0.3965 - val_loss: 1.0901 - val_accuracy: 0.3516 - lr: 2.0000e-04
```

In [92]:

```
loss = history.history['loss']
val_loss = history.history['val_loss']
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

fig = plt.figure(figsize=(15, 7))
ax = plt.gca()

ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy (Line), Loss (Dashes)')

ax.axhline(1, color='gray')

plt.plot(accuracy, color='blue')
plt.plot(val_accuracy, color='orange')
plt.plot(loss, '--', color='blue', alpha=0.5)
plt.plot(val_loss, '--', color='orange', alpha=0.5)
```

Out[92]:

```
[<matplotlib.lines.Line2D at 0x168cdc83190>]
```



Results

batch size	accuracy	val_accuracy	time per step	time per epoch	epochs needed
1	0.9883	0.9688	~ 7 ms	~ 4 s	22
8	0.9531	0.9531	~ 24 ms	~ 2 s	10
32	0.9766	0.9453	~ 80 ms	~ 1 s	18
128	0.9590	0.9375	~ 285 ms	~ 1 s	30
256	0.9316	0.9531	~ 600 ms	~ 1 s	50
512	0.3965	0.3516	~ 1 s	~ 1 s	5

As you can see the worst accuracy is with bactch size 512 (0.3965 / val: 0.3416). Already after 5 epochs the accuracy couldn't get better, but with ~5 seconds training time it was the fastest.

The best accuracy can be seen with batch size 1 (0.9883 / val: 0.9688), but it took around 88 seconds for the training.

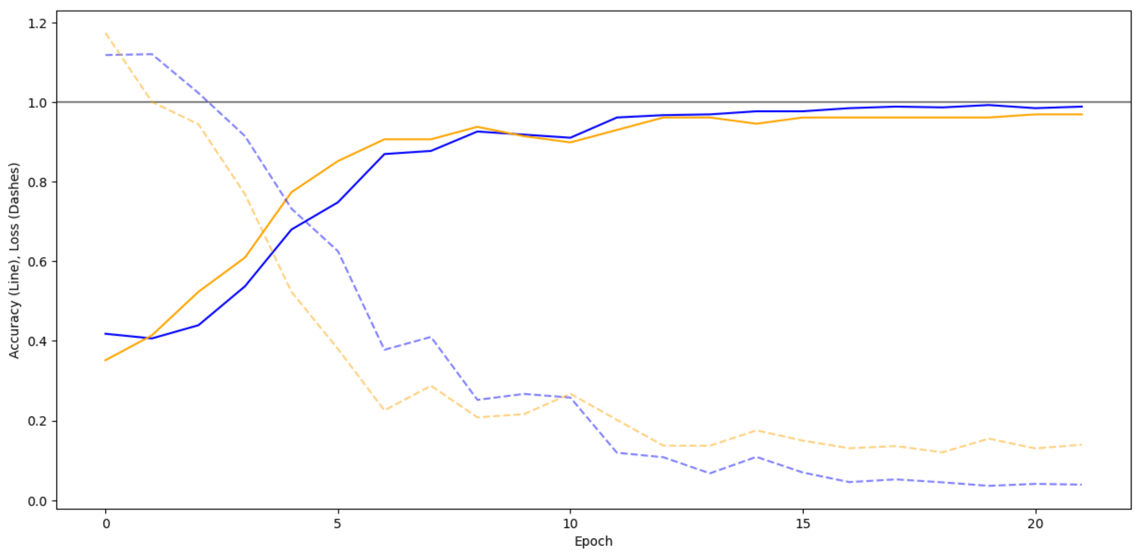
As we can see: the higher the batch size, the lower the accuracy. Except for the batch size = 1, the time to train the model got longer with a higher batch size. But it seems that there is a point, where the model prediciton gets so bad, the model will cancel the training after few epochs, because no better results are shown (batch size 512).

The plots of accuracy and loss of the training process also shows this visually.

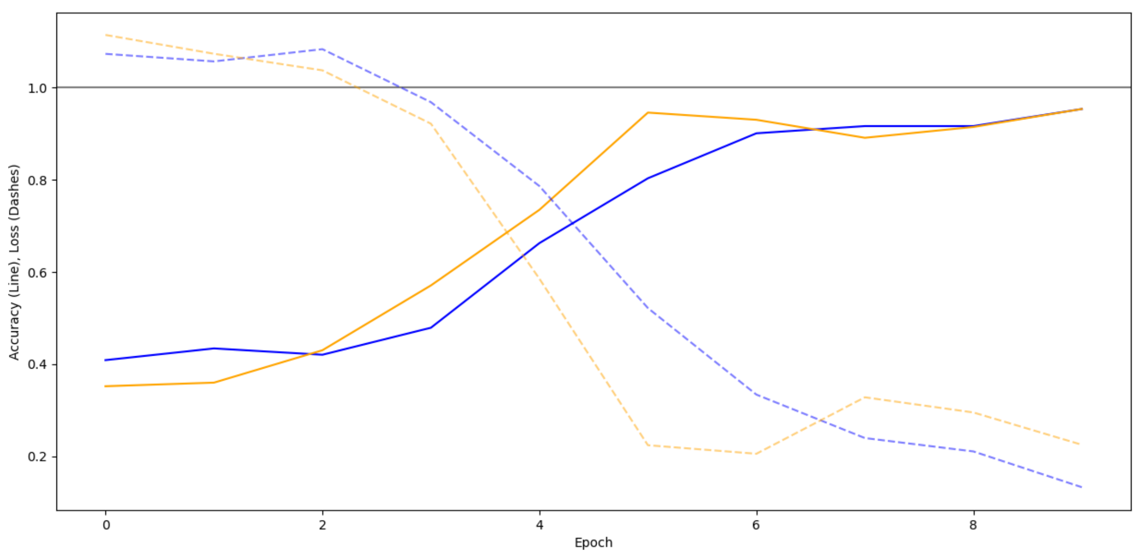
The batch size = 1 was very quick for a rather good prediction, in the last ~8 epochs not much progress in better accuracy was made. The plot with the maximum batch size shows that after one epoch accuracy couldn't approve.

Plots of accuracy and loss of the training process

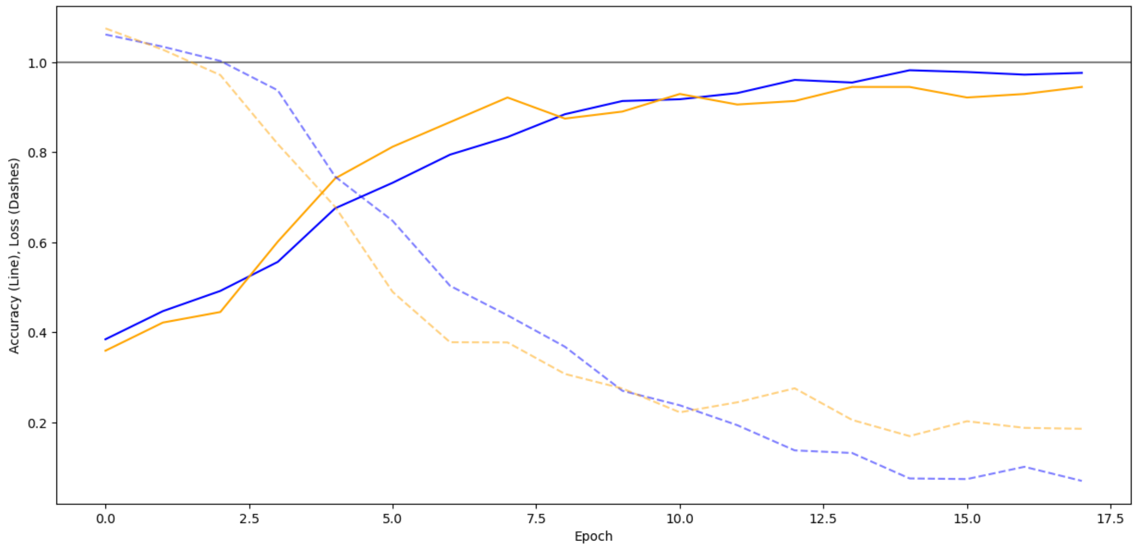
batch size = 1



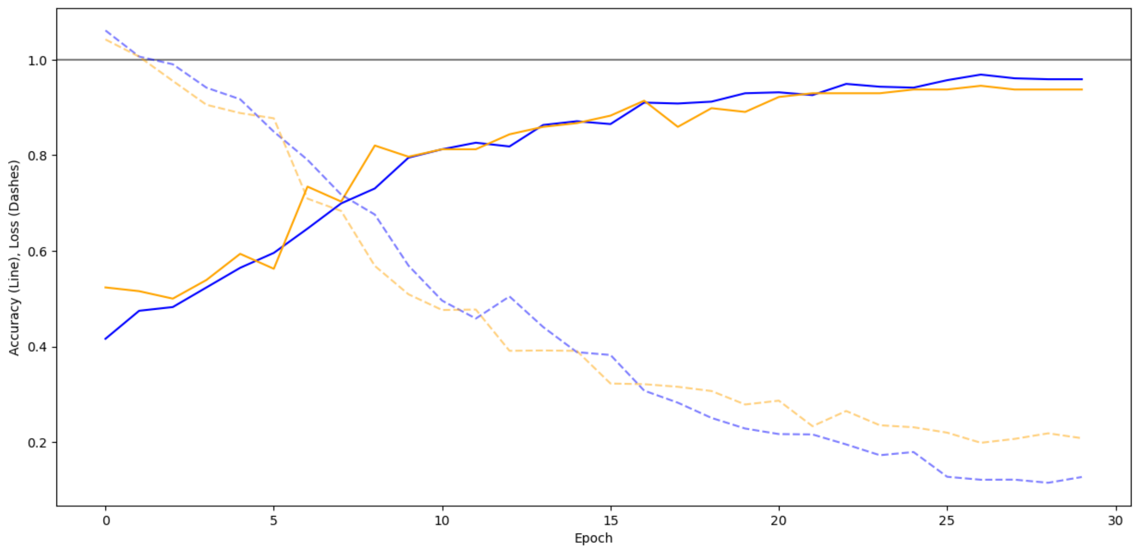
batch size = 8



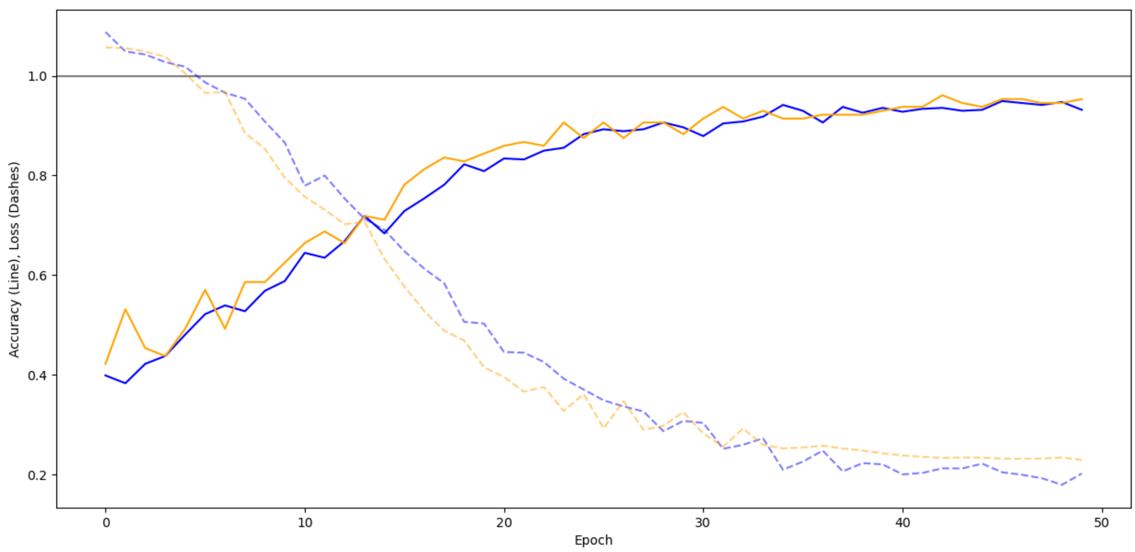
batch size = 32



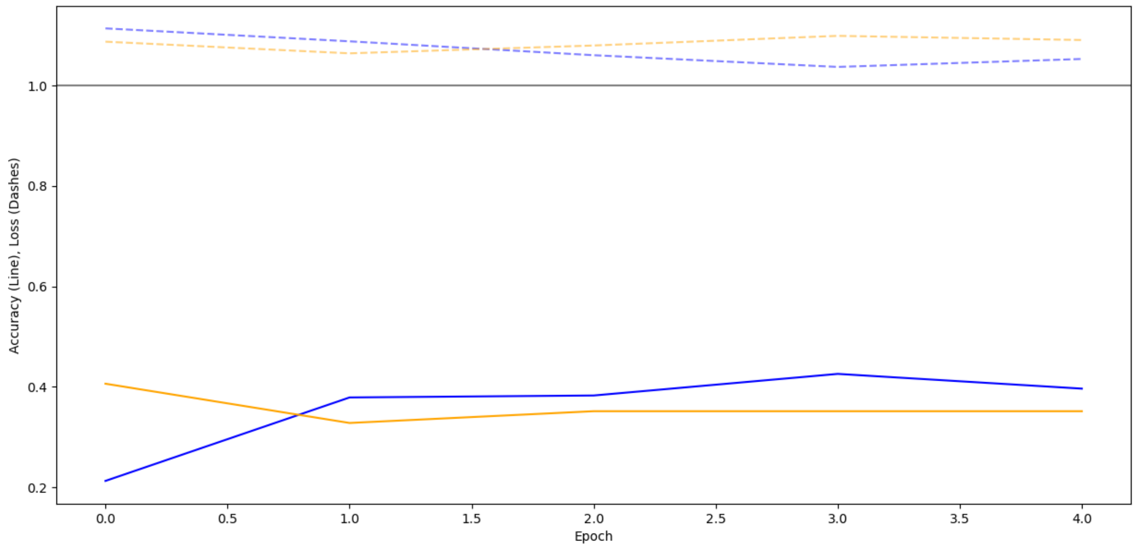
batch size = 128



batch size = 256



batch size = 512




```
In [ ]:
```