# Facial Emotion Recognition

## Import Libraries

In [2]:
```
import math
import numpy as np
import pandas as pd

import scikitplot
import seaborn as sns
from matplotlib import pyplot

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report

import tensorflow as tf
from tensorflow.keras import optimizers
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Conv2D, MaxPooling2D
from tensorflow.keras.layers import Dropout, BatchNormalization, LeakyReLU, Activation
from tensorflow.keras.callbacks import Callback, EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator

from keras.utils import np_utils
```

```
Using TensorFlow backend.
```

In [3]:
```
df = pd.read_csv('../input/facial-expression-recognitionferchallenge/fer2013.csv')
print(df.shape)
df.head()
```

```
(35887, 3)
```

Out[3]:

| | emotion | pixels | Usage |
|---|---|---|---|
| **0** | 0 | 70 80 82 72 58 58 60 63 54 58 60 48 89 115 121... | Training |
| **1** | 0 | 151 150 147 155 148 133 111 140 170 174 182 15... | Training |
| **2** | 2 | 231 212 156 164 174 138 161 173 182 200 106 38... | Training |
| **3** | 4 | 24 32 36 30 32 23 19 20 30 41 21 22 32 34 21 1... | Training |
| **4** | 6 | 4 0 0 0 0 0 0 0 0 0 0 0 0 3 15 23 28 48 50 58 84... | Training |

In [4]:
```
df.emotion.unique()
```

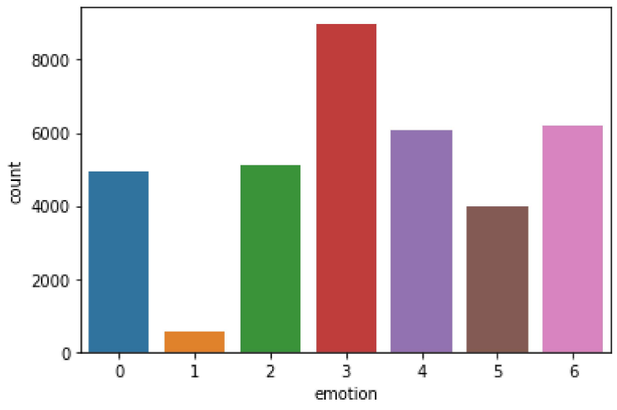
Out[4]:
```
array([0, 2, 4, 6, 3, 5, 1])
```

In [5]:
```
emotion_label_to_text = {0:'anger', 1:'disgust', 2:'fear', 3:'happiness', 4: 'sadness', 5: 'surprise', 6: 'neutral'}
```

In [6]:
```
df.emotion.value_counts()
```

Out[6]:
```
3    8989
6    6198
4    6077
2    5121
0    4953
5    4002
1     547
Name: emotion, dtype: int64
```

In [7]:
```
sns.countplot(df.emotion)
pyplot.show()
```




So majority classes belongs to 3:Happy, 4:Sad and 6:Neutral. we are intersted in these three classes only.

In [8]:
```
math.sqrt(len(df.pixels[0].split(' ')))
```

Out[8]:
```
48.0
```

In [9]:
```
fig = pyplot.figure(1, (14, 14))

k = 0
for label in sorted(df.emotion.unique()):
    for j in range(7):
        px = df[df.emotion==label].pixels.iloc[k]
        px = np.array(px.split(' ')).reshape(48, 48).astype('float32')

        k += 1
        ax = pyplot.subplot(7, 7, k)
        ax.imshow(px, cmap='gray')
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(emotion_label_to_text[label])
        pyplot.tight_layout()
```


anger
anger
anger
anger
anger
anger
anger
disgust
disgust
disgust
disgust
disgust
disgust
disgust
fear
fear
fear
fear
fear
fear
fear
happiness
happiness
happiness
happiness
happiness
happiness
happiness
sadness
sadness
sadness
sadness
sadness
sadness
sadness
surprise
surprise
surprise
surprise
surprise
surprise
surprise
neutral
neutral
neutral
neutral
neutral
neutral
neutral

In [10]:
```
INTERESTED_LABELS = [3, 4, 6]
```

In [11]:
```
df = df[df.emotion.isin(INTERESTED_LABELS)]
df.shape
```

Out[11]:
```
(21264, 3)
```

Now I will make the data compatible for neural networks.

In [12]:
```
img_array = df.pixels.apply(lambda x: np.array(x.split(' ')).reshape(48, 48, 1).astype('float32'))
img_array = np.stack(img_array, axis=0)
```

In [13]:
```
img_array.shape
```

Out[13]:
```
(21264, 48, 48, 1)
```

In [14]:
```
le = LabelEncoder()
img_labels = le.fit_transform(df.emotion)
img_labels = np_utils.to_categorical(img_labels)
img_labels.shape
```

Out[14]:
```
(21264, 3)
```

In [15]:
```
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
print(le_name_mapping)
```

```
{3: 0, 4: 1, 6: 2}
```

Splitting the data into training and validation set.

In [16]:
```
X_train, X_valid, y_train, y_valid = train_test_split(img_array, img_labels,
                                                      shuffle=True, stratify=img_labels,
                                                      test_size=0.1, random_state=42)
X_train.shape, X_valid.shape, y_train.shape, y_valid.shape
```

Out[16]:
```
((19137, 48, 48, 1), (2127, 48, 48, 1), (19137, 3), (2127, 3))
```

In [17]:
```
del df
del img_array
del img_labels
```

In [18]:
```
img_width = X_train.shape[1]
img_height = X_train.shape[2]
img_depth = X_train.shape[3]
num_classes = y_train.shape[1]
```

In [19]:
```
# Normalizing results, as neural networks are very sensitive to unnormalized data.
X_train = X_train / 255.
X_valid = X_valid / 255.
```

In [20]:
```
def build_net(optim):
    """
    This is a Deep Convolutional Neural Network (DCNN). For generalization purpose I used dropouts in regular intervals.
```

```
I used `ELU` as the activation because it avoids dying relu problem but also performed well as compared to LeakyRelu
atleast in this case. `he_normal` kernel initializer is used as it suits ELU. BatchNormalization is also used for better
results.
"""
net = Sequential(name='DCNN')

net.add(
    Conv2D(
        filters=64,
        kernel_size=(5,5),
        input_shape=(img_width, img_height, img_depth),
        activation='elu',
        padding='same',
        kernel_initializer='he_normal',
        name='conv2d_1'
    )
)
net.add(BatchNormalization(name='batchnorm_1'))
net.add(
    Conv2D(
        filters=64,
        kernel_size=(5,5),
        activation='elu',
        padding='same',
        kernel_initializer='he_normal',
        name='conv2d_2'
    )
)
net.add(BatchNormalization(name='batchnorm_2'))

net.add(MaxPooling2D(pool_size=(2,2), name='maxpool2d_1'))
net.add(Dropout(0.4, name='dropout_1'))

net.add(
    Conv2D(
        filters=128,
        kernel_size=(3,3),
        activation='elu',
        padding='same',
        kernel_initializer='he_normal',
        name='conv2d_3'
    )
)
net.add(BatchNormalization(name='batchnorm_3'))
net.add(
    Conv2D(
        filters=128,
        kernel_size=(3,3),
        activation='elu',
        padding='same',
        kernel_initializer='he_normal',
        name='conv2d_4'
    )
)
net.add(BatchNormalization(name='batchnorm_4'))

net.add(MaxPooling2D(pool_size=(2,2), name='maxpool2d_2'))
net.add(Dropout(0.4, name='dropout_2'))

net.add(
```

```
        Conv2D(
            filters=256,
            kernel_size=(3,3),
            activation='elu',
            padding='same',
            kernel_initializer='he_normal',
            name='conv2d_5'
        )
    )
    net.add(BatchNormalization(name='batchnorm_5'))
    net.add(
        Conv2D(
            filters=256,
            kernel_size=(3,3),
            activation='elu',
            padding='same',
            kernel_initializer='he_normal',
            name='conv2d_6'
        )
    )
    net.add(BatchNormalization(name='batchnorm_6'))

    net.add(MaxPooling2D(pool_size=(2,2), name='maxpool2d_3'))
    net.add(Dropout(0.5, name='dropout_3'))

    net.add(Flatten(name='flatten'))

    net.add(
        Dense(
            128,
            activation='elu',
            kernel_initializer='he_normal',
            name='dense_1'
        )
    )
    net.add(BatchNormalization(name='batchnorm_7'))

    net.add(Dropout(0.6, name='dropout_4'))

    net.add(
        Dense(
            num_classes,
            activation='softmax',
            name='out_layer'
        )
    )

    net.compile(
        loss='categorical_crossentropy',
        optimizer=optim,
        metrics=['accuracy']
    )

    net.summary()

    return net
```

In [21]:

```
"""
I used two callbacks one is `early stopping` for avoiding overfitting training data
```

```
and other `ReduceLROnPlateau` for learning rate.
"""

early_stopping = EarlyStopping(
    monitor='val_accuracy',
    min_delta=0.00005,
    patience=11,
    verbose=1,
    restore_best_weights=True,
)

lr_scheduler = ReduceLROnPlateau(
    monitor='val_accuracy',
    factor=0.5,
    patience=7,
    min_lr=1e-7,
    verbose=1,
)

callbacks = [
    early_stopping,
    lr_scheduler,
]
```

In [22]:
```
# As the data in hand is less as compared to the task so ImageDataGenerator is good to go.
train_datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    shear_range=0.15,
    zoom_range=0.15,
    horizontal_flip=True,
)
train_datagen.fit(X_train)
```

In [23]:
```
batch_size = 32 #batch size of 32 performs the best.
epochs = 100
optims = [
    optimizers.Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, name='Nadam'),
    optimizers.Adam(0.001),
]

# I tried both `Nadam` and `Adam`, the difference in results is not different but I finally went with Nadam as it is more popular.
model = build_net(optims[1])
history = model.fit_generator(
    train_datagen.flow(X_train, y_train, batch_size=batch_size),
    validation_data=(X_valid, y_valid),
    steps_per_epoch=len(X_train) / batch_size,
    epochs=epochs,
    callbacks=callbacks,
    use_multiprocessing=True
)
```

```
Model: "DCNN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 48, 48, 64)        1664
_________________________________________________________________
batchnorm_1 (BatchNormalizat (None, 48, 48, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 48, 48, 64)        102464
_________________________________________________________________
batchnorm_2 (BatchNormalizat (None, 48, 48, 64)        256
_________________________________________________________________
maxpool2d_1 (MaxPooling2D)   (None, 24, 24, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 24, 24, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 128)       73856
_________________________________________________________________
batchnorm_3 (BatchNormalizat (None, 24, 24, 128)       512
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 24, 24, 128)       147584
_________________________________________________________________
batchnorm_4 (BatchNormalizat (None, 24, 24, 128)       512
_________________________________________________________________
maxpool2d_2 (MaxPooling2D)   (None, 12, 12, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 12, 12, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 12, 12, 256)       295168
_________________________________________________________________
batchnorm_5 (BatchNormalizat (None, 12, 12, 256)       1024
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 12, 12, 256)       590080
_________________________________________________________________
batchnorm_6 (BatchNormalizat (None, 12, 12, 256)       1024
_________________________________________________________________
maxpool2d_3 (MaxPooling2D)   (None, 6, 6, 256)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 6, 6, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1179776
_________________________________________________________________
batchnorm_7 (BatchNormalizat (None, 128)               512
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
out_layer (Dense)            (None, 3)                 387
=================================================================
Total params: 2,395,075
Trainable params: 2,393,027
Non-trainable params: 2,048
_________________________________________________________________
Train for 598.03125 steps, validate on 2127 samples
Epoch 1/100
599/598 [==============================] - 14s 23ms/step - loss: 1.2925 - accuracy: 0.4078 - val_loss: 0.9978 - val_accuracy: 0.5049
Epoch 2/100
599/598 [==============================] - 10s 16ms/step - loss: 1.0121 - accuracy: 0.4953 - val_loss: 0.9018 - val_accuracy: 0.5487
Epoch 3/100
```

```
599/598 [==============================] - 10s 16ms/step - loss: 0.9009 - accuracy: 0.5671 - val_loss: 0.7757 - val_accuracy: 0.6342
Epoch 4/100
599/598 [==============================] - 10s 16ms/step - loss: 0.7957 - accuracy: 0.6245 - val_loss: 0.6632 - val_accuracy: 0.6963
Epoch 5/100
599/598 [==============================] - 10s 16ms/step - loss: 0.7401 - accuracy: 0.6585 - val_loss: 0.6749 - val_accuracy: 0.6925
Epoch 6/100
599/598 [==============================] - 10s 16ms/step - loss: 0.7182 - accuracy: 0.6732 - val_loss: 0.6091 - val_accuracy: 0.7405
Epoch 7/100
599/598 [==============================] - 10s 16ms/step - loss: 0.6845 - accuracy: 0.6982 - val_loss: 0.6155 - val_accuracy: 0.7348
Epoch 8/100
599/598 [==============================] - 10s 16ms/step - loss: 0.6656 - accuracy: 0.7032 - val_loss: 0.6107 - val_accuracy: 0.7311
Epoch 9/100
599/598 [==============================] - 10s 16ms/step - loss: 0.6448 - accuracy: 0.7200 - val_loss: 0.6025 - val_accuracy: 0.7330
Epoch 10/100
599/598 [==============================] - 10s 16ms/step - loss: 0.6348 - accuracy: 0.7261 - val_loss: 0.5476 - val_accuracy: 0.7593
Epoch 11/100
599/598 [==============================] - 10s 16ms/step - loss: 0.6205 - accuracy: 0.7333 - val_loss: 0.5438 - val_accuracy: 0.7729
Epoch 12/100
599/598 [==============================] - 10s 16ms/step - loss: 0.6130 - accuracy: 0.7331 - val_loss: 0.5132 - val_accuracy: 0.7856
Epoch 13/100
599/598 [==============================] - 9s 16ms/step - loss: 0.6021 - accuracy: 0.7434 - val_loss: 0.5208 - val_accuracy: 0.7842
Epoch 14/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5909 - accuracy: 0.7476 - val_loss: 0.5706 - val_accuracy: 0.7466
Epoch 15/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5795 - accuracy: 0.7542 - val_loss: 0.5009 - val_accuracy: 0.7955
Epoch 16/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5759 - accuracy: 0.7568 - val_loss: 0.5193 - val_accuracy: 0.7866
Epoch 17/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5683 - accuracy: 0.7593 - val_loss: 0.5587 - val_accuracy: 0.7560
Epoch 18/100
599/598 [==============================] - 10s 17ms/step - loss: 0.5597 - accuracy: 0.7611 - val_loss: 0.5042 - val_accuracy: 0.7870
Epoch 19/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5507 - accuracy: 0.7661 - val_loss: 0.5445 - val_accuracy: 0.7729
Epoch 20/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5479 - accuracy: 0.7693 - val_loss: 0.5397 - val_accuracy: 0.7781
Epoch 21/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5451 - accuracy: 0.7689 - val_loss: 0.5165 - val_accuracy: 0.7861
Epoch 22/100
598/598 [==============================>.] - ETA: 0s - loss: 0.5439 - accuracy: 0.7689
Epoch 00022: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
599/598 [==============================] - 10s 16ms/step - loss: 0.5435 - accuracy: 0.7691 - val_loss: 0.5286 - val_accuracy: 0.7819
Epoch 23/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5173 - accuracy: 0.7824 - val_loss: 0.4762 - val_accuracy: 0.8072
Epoch 24/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5069 - accuracy: 0.7894 - val_loss: 0.4661 - val_accuracy: 0.8096
Epoch 25/100
599/598 [==============================] - 10s 16ms/step - loss: 0.5032 - accuracy: 0.7888 - val_loss: 0.4578 - val_accuracy: 0.8143
Epoch 26/100
599/598 [==============================] - 9s 16ms/step - loss: 0.5003 - accuracy: 0.7898 - val_loss: 0.4559 - val_accuracy: 0.8190
Epoch 27/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4967 - accuracy: 0.7928 - val_loss: 0.4804 - val_accuracy: 0.7997
Epoch 28/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4943 - accuracy: 0.7961 - val_loss: 0.4606 - val_accuracy: 0.8105
Epoch 29/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4913 - accuracy: 0.7956 - val_loss: 0.4563 - val_accuracy: 0.8072
Epoch 30/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4886 - accuracy: 0.7978 - val_loss: 0.4706 - val_accuracy: 0.8096
Epoch 31/100
599/598 [==============================] - 9s 16ms/step - loss: 0.4854 - accuracy: 0.8004 - val_loss: 0.4545 - val_accuracy: 0.8110
Epoch 32/100
```

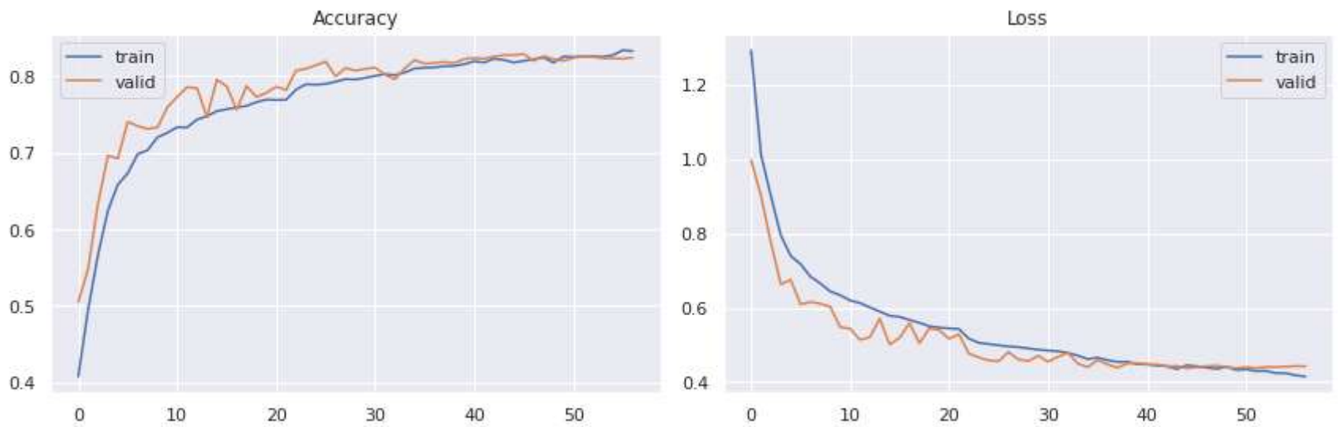
```
599/598 [==============================] - 9s 16ms/step - loss: 0.4842 - accuracy: 0.8025 - val_loss: 0.4677 - val_accuracy: 0.8016
Epoch 33/100
598/598 [============================>.] - ETA: 0s - loss: 0.4801 - accuracy: 0.8012
Epoch 00033: ReduceLROnPlateau reducing learning rate to 0.00025000000118743628.
599/598 [==============================] - 10s 16ms/step - loss: 0.4797 - accuracy: 0.8013 - val_loss: 0.4789 - val_accuracy: 0.7960
Epoch 34/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4719 - accuracy: 0.8049 - val_loss: 0.4497 - val_accuracy: 0.8105
Epoch 35/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4629 - accuracy: 0.8100 - val_loss: 0.4400 - val_accuracy: 0.8209
Epoch 36/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4661 - accuracy: 0.8108 - val_loss: 0.4586 - val_accuracy: 0.8162
Epoch 37/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4597 - accuracy: 0.8114 - val_loss: 0.4476 - val_accuracy: 0.8171
Epoch 38/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4551 - accuracy: 0.8130 - val_loss: 0.4382 - val_accuracy: 0.8185
Epoch 39/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4554 - accuracy: 0.8133 - val_loss: 0.4497 - val_accuracy: 0.8171
Epoch 40/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4486 - accuracy: 0.8156 - val_loss: 0.4508 - val_accuracy: 0.8223
Epoch 41/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4493 - accuracy: 0.8194 - val_loss: 0.4467 - val_accuracy: 0.8237
Epoch 42/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4453 - accuracy: 0.8177 - val_loss: 0.4474 - val_accuracy: 0.8228
Epoch 43/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4441 - accuracy: 0.8227 - val_loss: 0.4416 - val_accuracy: 0.8256
Epoch 44/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4358 - accuracy: 0.8210 - val_loss: 0.4430 - val_accuracy: 0.8275
Epoch 45/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4461 - accuracy: 0.8177 - val_loss: 0.4369 - val_accuracy: 0.8275
Epoch 46/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4430 - accuracy: 0.8199 - val_loss: 0.4398 - val_accuracy: 0.8289
Epoch 47/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4404 - accuracy: 0.8221 - val_loss: 0.4424 - val_accuracy: 0.8199
Epoch 48/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4358 - accuracy: 0.8243 - val_loss: 0.4438 - val_accuracy: 0.8260
Epoch 49/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4418 - accuracy: 0.8176 - val_loss: 0.4402 - val_accuracy: 0.8223
Epoch 50/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4331 - accuracy: 0.8254 - val_loss: 0.4372 - val_accuracy: 0.8199
Epoch 51/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4351 - accuracy: 0.8250 - val_loss: 0.4400 - val_accuracy: 0.8237
Epoch 52/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4311 - accuracy: 0.8258 - val_loss: 0.4376 - val_accuracy: 0.8256
Epoch 53/100
598/598 [============================>.] - ETA: 0s - loss: 0.4306 - accuracy: 0.8256
Epoch 00053: ReduceLROnPlateau reducing learning rate to 0.00012500000059371814.
599/598 [==============================] - 10s 16ms/step - loss: 0.4304 - accuracy: 0.8257 - val_loss: 0.4401 - val_accuracy: 0.8251
Epoch 54/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4253 - accuracy: 0.8251 - val_loss: 0.4402 - val_accuracy: 0.8232
Epoch 55/100
599/598 [==============================] - 10s 16ms/step - loss: 0.4243 - accuracy: 0.8273 - val_loss: 0.4413 - val_accuracy: 0.8232
Epoch 56/100
599/598 [==============================] - 10s 17ms/step - loss: 0.4190 - accuracy: 0.8338 - val_loss: 0.4426 - val_accuracy: 0.8228
Epoch 57/100
596/598 [============================>.] - ETA: 0s - loss: 0.4149 - accuracy: 0.8328Restoring model weights from the end of the best epoch.
599/598 [==============================] - 10s 16ms/step - loss: 0.4154 - accuracy: 0.8326 - val_loss: 0.4422 - val_accuracy: 0.8242
Epoch 00057: early stopping
```

In [24]:

```
model_yaml = model.to_yaml()
with open("model.yaml", "w") as yaml_file:
```

```
    yaml_file.write(model_yaml)

model.save("model.h5")
```

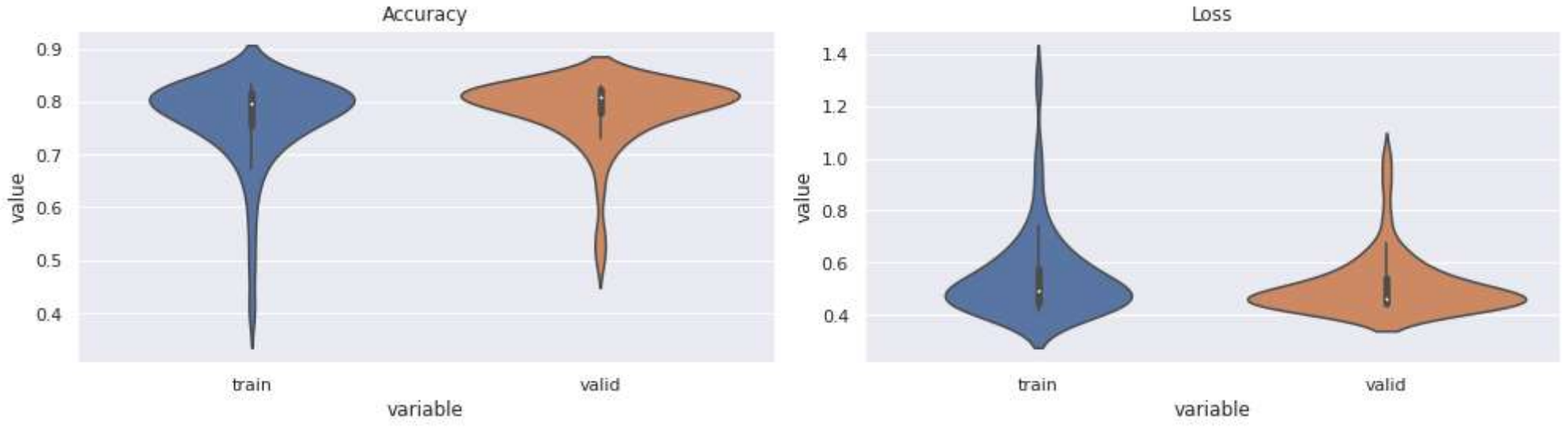
In [25]:
```
sns.set()
fig = pyplot.figure(0, (12, 4))

ax = pyplot.subplot(1, 2, 1)
sns.lineplot(history.epoch, history.history['accuracy'], label='train')
sns.lineplot(history.epoch, history.history['val_accuracy'], label='valid')
pyplot.title('Accuracy')
pyplot.tight_layout()

ax = pyplot.subplot(1, 2, 2)
sns.lineplot(history.epoch, history.history['loss'], label='train')
sns.lineplot(history.epoch, history.history['val_loss'], label='valid')
pyplot.title('Loss')
pyplot.tight_layout()

pyplot.savefig('epoch_history_dcnn.png')
pyplot.show()
```




The epochs history shows that accuracy gradually increases and achieved +83% accuracy on both training and validation set, but at the end, the model starts overfitting training data.

In [26]:
```
df_accu = pd.DataFrame({'train': history.history['accuracy'], 'valid': history.history['val_accuracy']})
df_loss = pd.DataFrame({'train': history.history['loss'], 'valid': history.history['val_loss']})

fig = pyplot.figure(0, (14, 4))
ax = pyplot.subplot(1, 2, 1)
sns.violinplot(x="variable", y="value", data=pd.melt(df_accu), showfliers=False)
pyplot.title('Accuracy')
pyplot.tight_layout()

ax = pyplot.subplot(1, 2, 2)
sns.violinplot(x="variable", y="value", data=pd.melt(df_loss), showfliers=False)
pyplot.title('Loss')
pyplot.tight_layout()
```

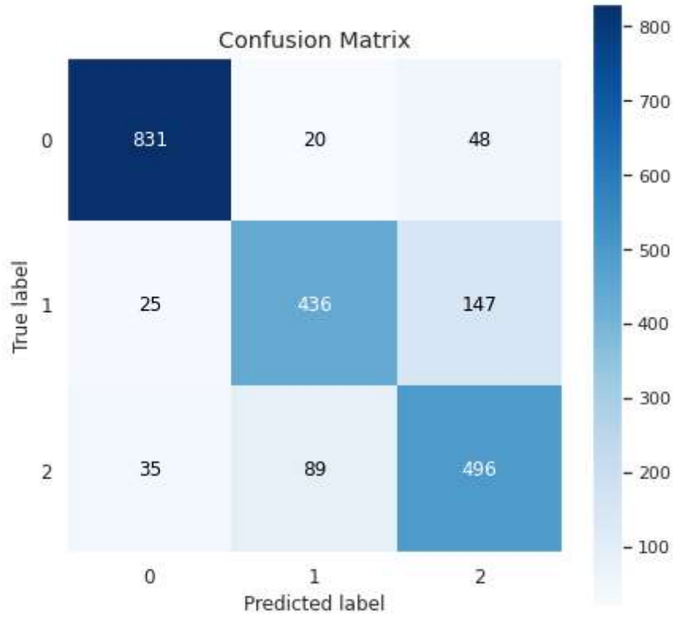
```
pyplot.savefig('performance_dist.png')
pyplot.show()
```




In [27]:
```
yhat_valid = model.predict_classes(X_valid)
scikitplot.metrics.plot_confusion_matrix(np.argmax(y_valid, axis=1), yhat_valid, figsize=(7,7))
pyplot.savefig("confusion_matrix_dcnn.png")

print(f'total wrong validation predictions: {np.sum(np.argmax(y_valid, axis=1) != yhat_valid)}\n\n')
print(classification_report(np.argmax(y_valid, axis=1), yhat_valid))
```

```
total wrong validation predictions: 364
```

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.93 | 0.92 | 0.93 | 899 |
| 1 | 0.80 | 0.72 | 0.76 | 608 |
| 2 | 0.72 | 0.80 | 0.76 | 620 |
| accuracy | | | 0.83 | 2127 |
| macro avg | 0.82 | 0.81 | 0.81 | 2127 |
| weighted avg | 0.83 | 0.83 | 0.83 | 2127 |




The confusion matrix clearly shows that our model is performing well in the 'happy' class, but its performance is lower in the other two classes. One possible reason for this could be the scarcity of data in these two classes. However, upon reviewing the images, I noticed that some from these classes are even challenging for a human to discern whether the person appears sad or neutral. Facial expressions can vary widely among individuals; for instance, some people's neutral expressions might resemble a sad expression.

In [28]:
```
mapper = {
    0: "happy",
    1: "sad",
    2: "neutral",
}
```

In [29]:
```
np.random.seed(2)
random_sad_imgs = np.random.choice(np.where(y_valid[:, 1]==1)[0], size=9)
random_neutral_imgs = np.random.choice(np.where(y_valid[:, 2]==1)[0], size=9)

fig = pyplot.figure(1, (18, 4))

for i, (sadidx, neuidx) in enumerate(zip(random_sad_imgs, random_neutral_imgs)):
        ax = pyplot.subplot(2, 9, i+1)
        sample_img = X_valid[sadidx,:,:,0]
        ax.imshow(sample_img, cmap='gray')
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(f"true:sad, pred:{mapper[model.predict_classes(sample_img.reshape(1,48,48,1))[0]]}")

        ax = pyplot.subplot(2, 9, i+10)
        sample_img = X_valid[neuidx,:,:,0]
        ax.imshow(sample_img, cmap='gray')
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(f"t:neut, p:{mapper[model.predict_classes(sample_img.reshape(1,48,48,1))[0]]}")

        pyplot.tight_layout()
```


Consider the seventh image in the first row: it appears more neutral than sad, and our model correctly predicted it as neutral. Conversely, the last image in the second row exhibits clear signs of sadness, which our model also correctly identified.