Face Recognition Project - Emotion in Video

Cecily Wang

University of Illinois at Urbana-Champaign

IL, United States

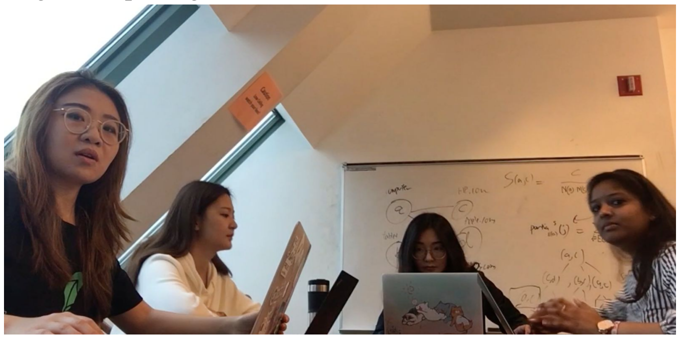
61801

***Abstract* - This Face Recognition Project aims to shows an application that analyzes faces detected in sampled video clips to interpret the emotion or mood of the subjects . It identifies faces, analyzes the emotions displayed on those faces, generates corresponding Emoji overlays on the video, and logs emotion data.**

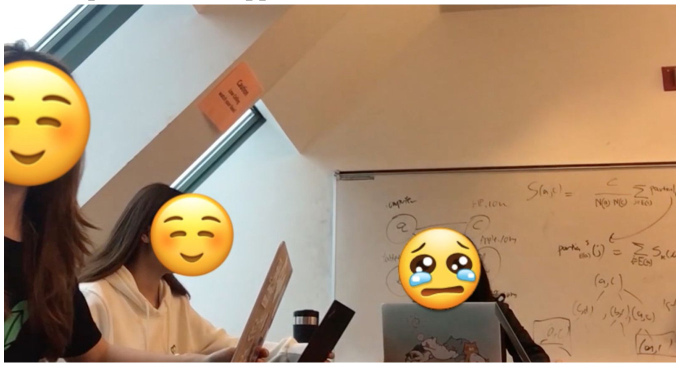
I. Introduction

The application accomplishes all of this within a serverless architecture using Amazon DeepLens,Amazon Rekognition, AWS Lambda, AWS Step Functions, and other AWS services, Aws deepLens. Also, this project combine Data Analysis and Machine Learning together.

Original Capturing Video:



The Output from the Application.



A. How it works

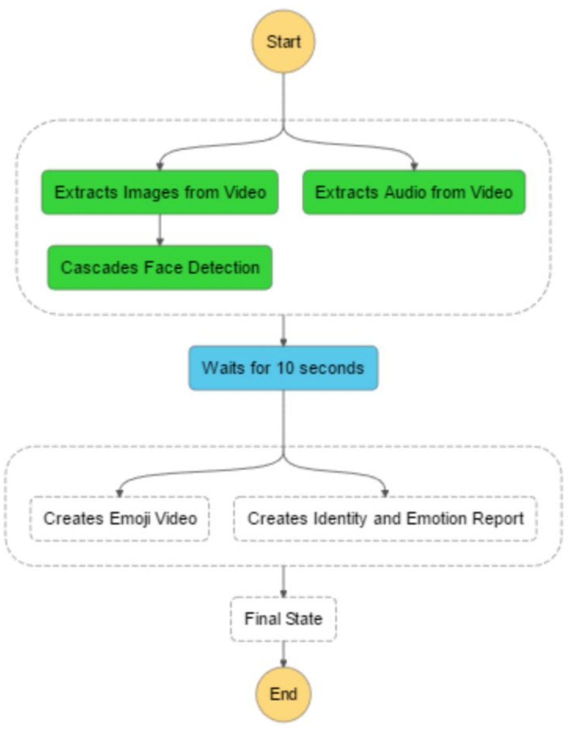
Teachers can use "FeelInVideo" to get an overall measure of a student's mood (e.g., happy, or calm, or confused) while taking attendance. The instructor can use this data to adjust his or her focus and approach to enhance the teaching experience. This research project is just beginning.

I designed a Machine Learning model in our AWS deepLens, which could detect face position properly, and then transfer the original Video to emoji Video.

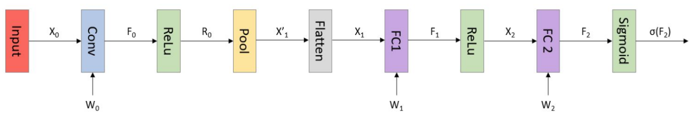
To use "FeelInVideo", a teacher sets up a basic classroom camera to take each students' attendance using face identification. The camera also captures how students feel during class. Teachers can also use "FeelInVideo" to prevent students from falsely reporting attendance.

B. Architecture and design

"FeelInVideo" is a serverless application built using AWS Lambda functions. Five of the Lambda functions are included in the "FeelInVideo" state machine. AWS Step Functions streamlines coordinating the components of distributed applications and microservices using visual workflows. This simplifies building and running multi-step applications.
The "FeelInVideo" state machine starts with the startFaceDetectionWorkFlowLambda function, which is triggered by an Amazon S3 PUT object event. startFaceDetectionWorkFlowLambda passes in the following information into the execution:


Start
Extracts Images from Video
Extracts Audio from Video
Cascades Face Detection
Waits for 10 seconds
Creates Emoji Video
Creates Identity and Emotion Report
Final State
End

In create emoji part, I applied CNN model for face position detection by using deepLens, here is the model framwork.



My Traning set.

II. Helpful Hints

C. *Behind the scenes*

Before you start using "FeelInVideo", you need to understand how it works and a few general principles.
When you need to make use of other pre-built programs such as FFmpeg, you can run another program or start a new process in Lambda:

```
let invokeLambdaPromises = keys.map(invokeLambda);
Promise.all(invokeLambdaPromises).then(() => {
        let pngKey = keys.map(key => key.split(".")[0] + ".png");
        let data = {bucket: bucket, prefix: prefix, keys: pngKey};
        console.log("involveLambdaPromises complete!");
    callback(null, data);
    }
).catch(err => {
    console.log("involveLambdaPromises failed!");
    callback(err);
});
```

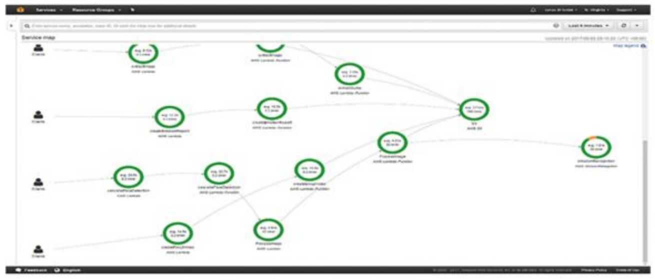
```
let invokeLambda = (key) => new Promise((resolve, reject) => {
    let data = JSON.stringify({bucket: bucket, key: prefix + "/" + key});
    let params = {
        FunctionName: process.env['ProcessImage'], /* required */
        Payload: data /* required */
    };
    lambda.invoke(params, (err, data) => {
        if (err) reject(err, err.stack); // an error occurred
        else     resolve(data);           // successful response
    });
});
```

The Cascades Face Detection step asynchronously invokes the ProcessImage Lambda function for each screen capture image nearly in parallel. Each ProcessImage function calls Amazon Rekognition for each face detected.
The following is a parallel map function which invokes the ProcessImage function for each image frame.

D. *Dependency Tree*

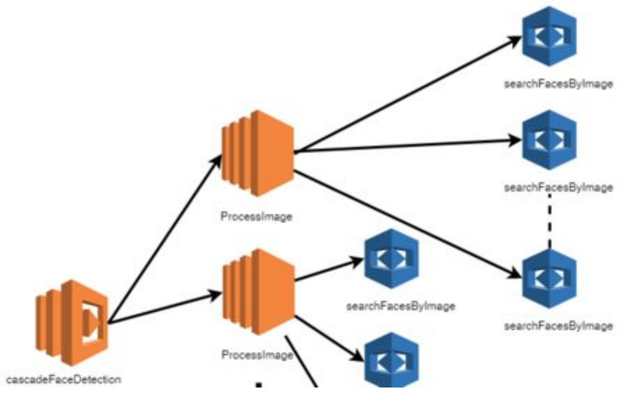
The following service map shows the dependency trees with trace data that I can use to drill into specific services or issues. This provides a view of connections between services in your application and aggregated data for each service, including average latency and failure rates.

E. The following is a latency distribution histogram for an Amazon Rekognition API call



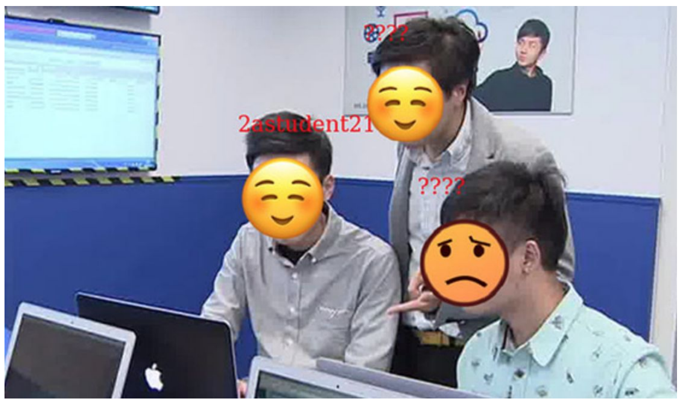
F. *Equations*

Latency is the amount of time between the start of a request and when it completes. A histogram shows a distribution of latencies. This latency distribution histogram shows duration on the x-axis, and the percentage of requests that match each duration on the y-axis.

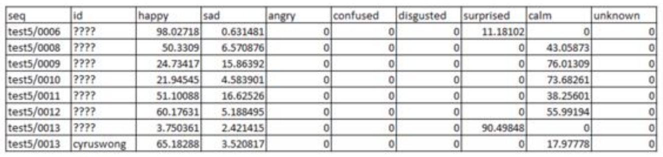


E. The following Lambda cascading timeline shows how processing operates in a highly parallel manner:

III. The result



Our Output Sample



seq	id	happy	sad	angry	confused	disgusted	surprised	calm	unknown
test5/0006	????	98.02718	0.631481	0	0	0	11.18102	0	0
test5/0008	????	50.3309	6.570876	0	0	0	0	43.05873	0
test5/0009	????	24.73417	15.86392	0	0	0	0	76.01309	0
test5/0010	????	21.94545	4.583901	0	0	0	0	73.68261	0
test5/0011	????	51.10088	16.62526	0	0	0	0	38.25601	0
test5/0012	????	60.17631	5.188495	0	0	0	0	55.99194	0
test5/0013	????	3.750361	2.421415	0	0	0	90.49848	0	0
test5/0013	cyruswong	65.18288	3.520817	0	0	0	0	17.97778	0

I don’t index images of my classmates’ faces within the video. Instead, they are each allocated an unknown faceId. With this report, you can easily aggregate data on overall student information.

SELECT Report.id AS Student, Count(Report.seq) AS Attended, Sum(Report.happy) AS SumOfhappy,

IV. My Machine Learning Algorithsm

This part shows how a Convolutional Neural Network (CNN) can apply the style of a painting to my surroundings as it's streamed with AWS DeepLens device. The project uses a pretrained optimized model that is ready to be deployed to your AWS DeepLens device. After deploying it, we can watch the stylized video stream.

I optimised CNN recognition method in deepLens’ model. The following part give enough details about how to asychonised finding ones’ face position.

A. Extracting image features

In order to extract image features, feature scaling should be firstly applied. The goal of feature scaling is to allow the range of features to be comparable. In this case, it normalizes the pixel value domain of each image from 0-255 to 0-1.

It is beneficial as it helps the algorithms work better. For instance, each feature should be normalized in the K-nearest neighbor algorithm since the classifier mainly calculates the Euclidean distance between two points. The distance will be dominated by the feature value that has a larger range value than other features.

Feature scaling can also speed up the gradient convergence. For example, it improves the convergence speed of the stochastic gradient descent algorithm.

B. Training and testing data

Mini-Batch Gradient Descent will be used for training the CNN since it gives a more stable convergence. The steps are shown below under function *batchify*:

(a) Shuffle the samples randomly and uniformly in training set.
(b) Separate samples into equally sized 16 batches.
(c) Backpropagation on each batch and update parameters accordingly.
(d) Go through all the batches in training set which completes one epoch.

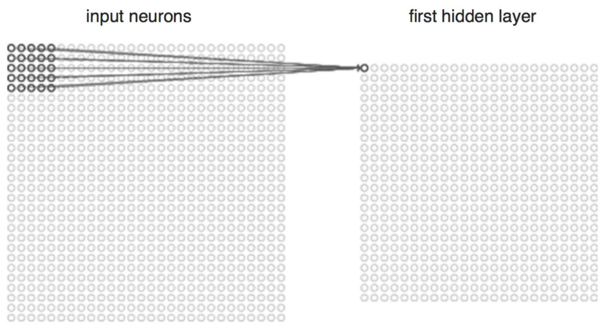


Figure 2. Input neurons and first hidden layer

In our preprocessing function:

First, I load all images using *imread* from a given folder, which is a list containing the names of the entries in the directory given by path. The stacked array has one more dimension than the input arrays.

I then read in images and create a list for each set in the form: *images, labels*. *images* is an numpy array with dimensions *N* * *img_height* * *img width* * *num_channels*. *labels* is also an numpy array with dimension [N * 1].
elephant = 0, *lionfish* = 1. As for the return value, *train_set* is the list of *train_images* and *train_labels* and *al_set* represents the list of *val_images and val_labels.*

As for *batchify*, our inputs include:

- *train_set*: list containing images and labels.
- *batch_size*: the desired size of each batch.

Out returns include:

- *image_batches*: list of shuffled training image batches, each with size *batch_size.* For each *batch_number*, we append *shuffled_image* from *shuffled_image* to *shuffled_image* + *batch_size* in order to get *image_batches*.
- *label_batches*: list of shuffled training label batches. For each *batch_number,* we append *shuffled_label* to *shuffled_label* + *batch_size*. We have the need to randomly upset an array, which randomly scrambles the sample during training by using *batch_size*. *numpy.random.permutation().*
- *batch_number*: initialise it with *total_number* divided by *batch_size*.

C. Implement forward pass of CNN

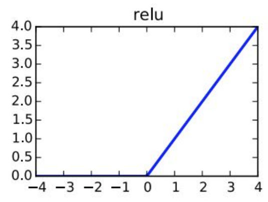
In order to test the correctness of the forward pass, pre-trained weights are provided in *weights.npz*. A loss of **0.232** and an accuracy of **91.75%** on the test set should be given by applying forward pass to the network.

Activation Functions:

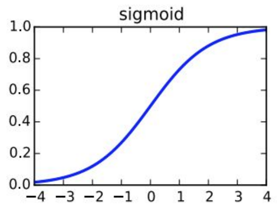
(1) relu
Inputs include *x*: multi-dimensional array with size N along the first axis. Returns include *out*: multi-dimensional array with same size of *x*. The ReLU layer applies the ReLU activation function over each feature map returned by the conv layer. It is called using the relu function.

(2) sigmoid
Inputs include *x*: multi-dimensional array with size N along the first axis. Returns include *out*: multi-dimensional array with same size of *x*.

(3) unit_step
Inputs include *x*: multi-dimensional array with size N along the first axis. Returns include *out*: Multi-dimensional array with same size of *x*.



$$f(x) = \max\{x, 0\}$$



$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

Figure 3. Graph of relu and sigmoid function

Layer Functions:

(1) convolve2D
We have two input, X with N * height * width * num_channels and filters with num_filters * filter_height * filter_width * num_input_channels.
As for our output, we have Xc with output array by convolving X and filters with N x output_height x output_width x num_filters. The size of the filters bank is specified by the above zero array but not the actual values of the filters. It is possible to override such values as follows to detect vertical and horizontal edges.

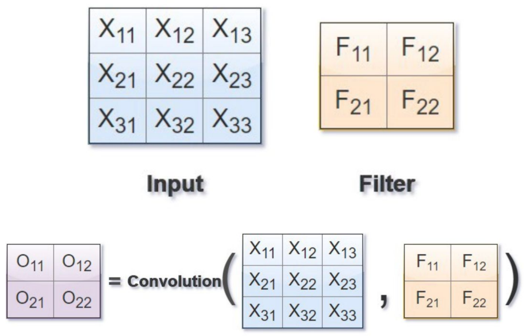
(2) maxPool
Inputs include R0: N * height * width * num_channels. And mp_len: size of max pool window, also the stride for this MP. We return p_out: output of pooling R0, which can be calculated as N * output_height * output_width * num_channels. R0_mask: A binary mask with the same size as R0. Indicates which index was chosen to be the max for each max pool window. This will be used for backpropagation. The max pooling layer accepts the output of

the ReLU layer and applies the max pooling operation. The function accepts three inputs which are the output of the ReLU layer, pooling mask size, and stride. It simply creates an empty array, as previous, that holds the output of such layer. The size of such array is specified according to the size and stride arguments.The function accepts three inputs which are the output of the ReLU layer, pooling mask size, and stride. It simply creates an empty array, as previous, that holds the output of such layer. The size of such array is specified according to the size and stride arguments.Up to this point, the CNN architecture with conv, ReLU, and max pooling layers is complete. There might be some other layers to be stacked in addition to the previous ones

(3) fc
In the fully convolutional layers, our Inputs contains X: N * num_input_features. And W: num_input_features * num_fc_nodes.
Our returns contains out: Linear combination of X and W , which can be calculated by N * num_fc_nodes.
Weights, which we load from "*weights.npz*"

(4) cnn_fwd
Given pretrained weights to test correctness. Test provided in notebook.

Method Explanation:
Convolution:
The convolution operation can be written as described in the figure below:



$$O_{11} = F_{11}X_{11} + F_{12}X_{12} + F_{21}X_{21} + F_{22}X_{22}$$
$$O_{12} = F_{11}X_{12} + F_{12}X_{13} + F_{21}X_{22} + F_{22}X_{23}$$
$$O_{21} = F_{11}X_{21} + F_{12}X_{22} + F_{21}X_{31} + F_{22}X_{32}$$
$$O_{22} = F_{11}X_{22} + F_{12}X_{23} + F_{21}X_{32} + F_{22}X_{33}$$

Figure 4. Convolution operation (ref. Sujit Rai)

Then, we conduct visualization and calculate the gradients of filter with the expect to the error,

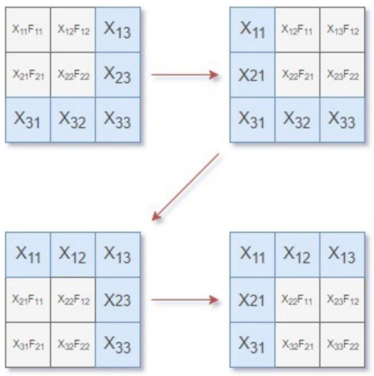


Figure 5. Convolution operation (ref. Sujit Rai)

Which evaluates to:

$$\frac{\partial E}{\partial F_{11}} = \frac{\partial E}{\partial O_{11}}X_{11} + \frac{\partial E}{\partial O_{12}}X_{12} + \frac{\partial E}{\partial O_{21}}X_{21} + \frac{\partial E}{\partial O_{22}}X_{22}$$
$$\frac{\partial E}{\partial F_{12}} = \frac{\partial E}{\partial O_{11}}X_{12} + \frac{\partial E}{\partial O_{12}}X_{13} + \frac{\partial E}{\partial O_{21}}X_{22} + \frac{\partial E}{\partial O_{22}}X_{23}$$
$$\frac{\partial E}{\partial F_{21}} = \frac{\partial E}{\partial O_{11}}X_{21} + \frac{\partial E}{\partial O_{12}}X_{22} + \frac{\partial E}{\partial O_{21}}X_{31} + \frac{\partial E}{\partial O_{22}}X_{32}$$
$$\frac{\partial E}{\partial F_{22}} = \frac{\partial E}{\partial O_{11}}X_{22} + \frac{\partial E}{\partial O_{12}}X_{23} + \frac{\partial E}{\partial O_{21}}X_{32} + \frac{\partial E}{\partial O_{22}}X_{33}$$

The above computation can be obtained by a different type of convolution operation known as full convolution.

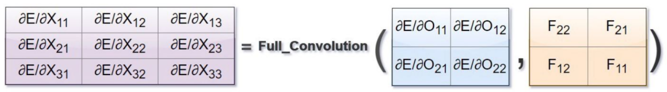


Figure 6. Full Convolution (ref. Sujit Rai)

Our input of *cnn_fwd* includes: *X0, W0, W1, W2, mp_len*, Then we complete convolve2D , reluctant , maxPool, flatten as F0, R0 ,X1p, R0_mask, X1 as our output. And do fc relu in our FC layers and then output use sigmoid activation.

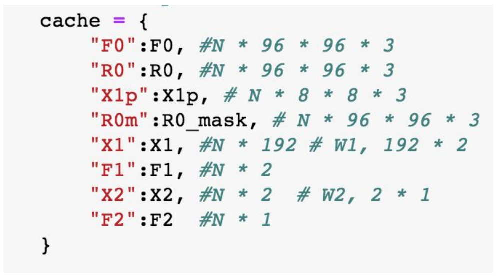
Chain Rule:

$$\frac{\partial a_i[n_1, n_2, k]}{\partial u[m_1, m_2, j, k]} = x_i[n_1 - m_1, n_2 - m_2, j]$$

$$a_i[n_1, n_2, k] = \sum_{m_1}\sum_{m_2} u[m_1, m_2, j, k]x_i[n_1 - m_1, n_2 - m_2, j]$$

$$\frac{\partial E_i}{\partial u[m_1, m_2, j, k]} = \sum_{n_1}\sum_{n_2}\left(\frac{\partial E_i}{\partial a_i[n_1, n_2, k]}\right)\left(\frac{\partial a_i[n_1, n_2, k]}{\partial u[m_1, m_2, j, k]}\right)$$

Save outputs of functions for backward pass, the cache is shown below:



```
cache = {
    "F0":F0, #N * 96 * 96 * 3
    "R0":R0, #N * 96 * 96 * 3
    "X1p":X1p, # N * 8 * 8 * 3
    "R0m":R0_mask, # N * 96 * 96 * 3
    "X1":X1, #N * 192 # W1, 192 * 2
    "F1":F1, #N * 2
    "X2":X2, #N * 2  # W2, 2 * 1
    "F2":F2  #N * 1
}
```

As for loss function, we used cross entropy to calculate the loss
Our input includes :
sig: vector containing the CNN output for each sample. [N * 1].
Y: vector containing the ground truth label for each sample. [N * 1].

D. Implement backward pass of CNN

We design and implement the code for mini-batch gradient descent. In classification problem, the error criterion/loss function used is cross-entropy. In the binary case, it is defined as follows:

$$L = \frac{1}{N}\sum_{i=0}^{N} -y_i \log(\sigma(f_{2i})) - (1-y_i)\log(1-\sigma(f_{2i}))$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial F_2} \times \frac{\partial F_2}{\partial W_2}$$

Here are the procedure of getting gradient:

$$\frac{\partial L}{\partial F_{2i}} = \frac{1}{N} \times \frac{\partial}{\partial F_{2i}} \times [-y_i \log(\delta(F_{2i}) - (1-\mathrm{y_i})\log(1-(\delta(F_{2i})))]$$

$$= \frac{1}{N}[-y_i(1-\delta(F_{2i}) + (1-y_i)\delta(F_{2i})]$$

$$= \frac{1}{N}(\delta(F_{2i}) - y_i)$$

$$\frac{\partial L}{\partial F_{2_{N\times 1}}} = \frac{1}{\mathrm{N}}[\delta(F_{2i}) - y_i]$$

$$F_2 = X_2 \times W_2$$

$$\frac{\partial L}{\partial W_{2i}} = \sum_j \frac{\partial L}{\partial F_{2j}}\frac{\partial F_{2j}}{\partial W_{2j}} = \sum_j \left(\frac{\partial L}{\partial F_2}\right)_j X_{2ji} = X_2^T \times \frac{\partial L}{\partial F_2}$$

$$F_1 = X_1 \times W_1$$

$$\frac{\partial L}{\partial W_1} = X_1^T \times \frac{\partial L}{\partial F_1} = X_1^T \times \left(\frac{\partial L}{\partial F_2} \times \frac{\partial F_2}{\partial X_2} \times \frac{\partial X_2}{\partial F_1}\right) = X_1^T \times [\frac{1}{N}(\delta(F_{2i}) - y_i) \times W_2^T \times unitstep(F_1)]$$

$$\frac{\partial L}{\partial F_0} = \frac{\partial L}{\partial R_0} \times \frac{\partial R_0}{\partial F_0} = pool^{-1}\left(unflatten\left(\frac{\partial L}{\partial F_1} \times \frac{\partial F_1}{\partial X_1}\right)\right) \times \frac{\partial R_0}{\partial F_0}$$
$$= pool^{-1}\left(unflatten\left(\frac{\partial L}{\partial F_1} \times \frac{\partial F_1}{\partial X_1}\right)\right) unitstep(F_0)$$
$$= pool^{-1}\left(unflatten\left(\frac{\partial L}{\partial F_1} @ N_1^T\right)\right) unitstep(F_0)$$

Our chain rule for convolution layer is:

$$\frac{\partial E_i}{\partial u[m_1, m_2, j, k]} = \sum_{n1}\sum_{n2} \delta_i[n_1, n_2, k] x_i[n_1 - m_1, n_2 - m_2, j]$$

Which should result in:

$$\delta_i[m_1, m_2, k] * x_i[-m_1, -m_2, j]$$

Where we've now defined the back-prop error term as:

$$\delta_i[n_1, n_2, k] = \frac{\partial E_i}{\partial a_i[n_1, n_2, k]}$$

$$\begin{aligned}\delta_i[n_1, n_2, k] &= \frac{\partial E_i}{\partial a_i[n_1, n_2, k]} \\ &= \sum_\ell \sum_{o_1} \sum_{o_2} \left(\frac{\partial E_i}{\partial b_{\ell i}}\right)\left(\frac{\partial b_{\ell i}}{\partial y_i[o_1, o_2, k]}\right)\left(\frac{\partial y_i[o_1, o_2, k]}{\partial a_i[n_1, n_2, k]}\right) \\ &= \begin{cases} \sum_\ell \epsilon_{\ell i} v_{\ell, kN_1N_2 + o_1N_2 + o_2} \\ \quad \text{if } (n_1, n_2) = \text{argmax}_{(p_1, p_2) \in \mathcal{A}(o_1, o_2)}\, a_i[p_1, p_2, k] \\ 0 \\ \quad \text{otherwise} \end{cases}\end{aligned}$$

That last condition just says that we back-prop only to the hidden nodes a[n1, n2, k] that survive max-pooling, not to any others. And to make the notation easier to remember, we can write:

$$\vec{\delta}_i = V^T \vec{\epsilon}_i|_{(n_1, n_2)}$$

when we put them together:

$$\frac{\partial E_i}{\partial u[m_1, m_2, j, k]} = \quad \delta_i[m_1, m_2, k] * x_i[-m_1, -m_2, j]$$

note that:

$$\epsilon_{\ell i} = \frac{\partial E_i}{\partial b_{\ell i}}$$

$$\delta_i[n_1, n_2, k] = V^T \vec{\epsilon}_i|_{(n_1, n_2)}$$

In our interface function, given the feature matrix [time, features], output the log probability of observing that under this model.

III. Results and Discussion

A. Train the Model

(1) Config
– Learning rate of 0.1
– Initialize weights with gaussian (0, 0.05^2)
(2) 30 seconds per epoch
(3) Record test accuracy at the end of every epoch and graph results
(4) Restart training process if accuracy is stuck at exactly 0.5 for 3 epochs or more.

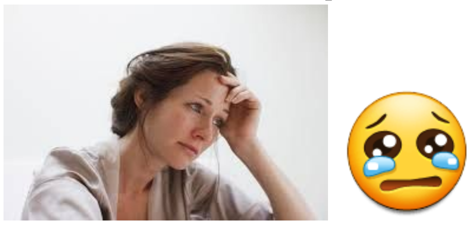
In this phase, the model is trained using training data and expected output for this data.

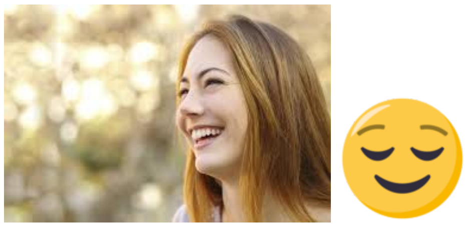
model.fit(training_data, expected_output(label))

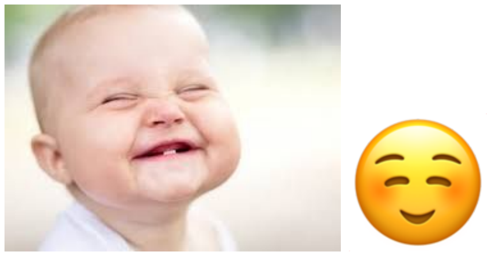
At the end of the process, the final accuracy of the model will show.
Once the model has been trained it is possible to carry out model testing. During this phase a second set of data is loaded. "Loaded 100 training images. Loaded 80 validation images" This data set has never been seen by the model and therefore it's true accuracy will be verified.
After the model training is complete, and it is understood that the model shows the right result, it can be saved by: model.save("train_loss_and_acc").

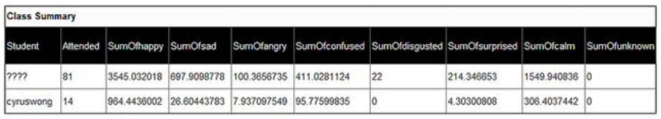
Finally, we print out val_acc which is **0.8735**.

As for our test data, our result picture are shown below:









Class Summary									
Student	Attended	SumOfhappy	SumOfsad	SumOfangry	SumOfconfused	SumOfdisgusted	SumOfsurprised	SumOfcalm	SumOfunknown
????	81	3545.032018	697.9098778	100.3656735	411.0281124	22	214.346653	1549.940836	0
cyruswong	14	964.4436002	26.60443783	7.937097549	95.77599835	0	4.30300808	306.4037442	0

Figure 7. Correctly detected results from Video

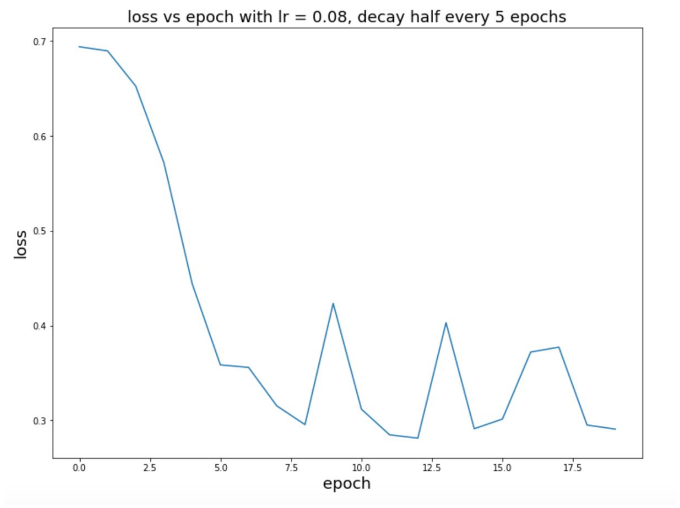


Figure 8. Loss vs epoch with lr = 0.08, decay half every 5 epochs

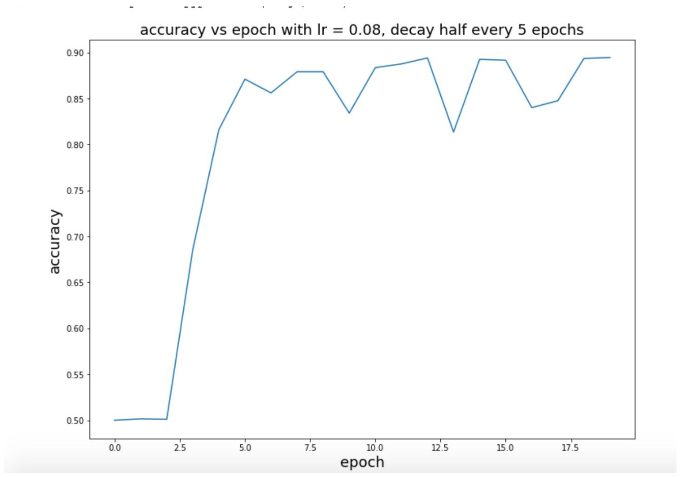


Figure 9. Accuracy vs epoch with lr = 0.08, decay half every 5 epochs

The accuracy increased from about 0.5 to over 0.9 as the model converges, however, it fluctuate a lot. This is because of the fixed learning rate as well as the behavior of mini-batch training. Mini-batch training helps to prevent the model stuck at the local minimum, at the same time results in oscillating accuracy curve.

This project can help us understand the emotions of all of the people captured in a video. It has a variety of applications, including education, training, rehabilitative care, and customer interactions. Deployment is simple with an AWS CloudFormation template. Just take a video with your smart phone and upload it to an S3 bucket. In a few minutes, you'll get the emotional analytic report!

Acknowledgment

Thanks for the remote help from my boyfriend.

References

[1] Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. CoRR, abs/1604.06174.

[2] Ganin, Y. and Lempitsky, V. (2014). Unsupervised Domain Adaptation by Backpropagation. ArXiv e-prints.

[3] Bengio, Yoshua and Glorot, Xavier. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of AISTATS 2010, volume 9, pp. 249– 256, May 2010.

[4] Kratzert, Frederik. (2016,Feb, 12). Understanding the backward pass through Batch Normalization Layer. Retrieved from https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html