CS F425: Deep Learning

Project Report

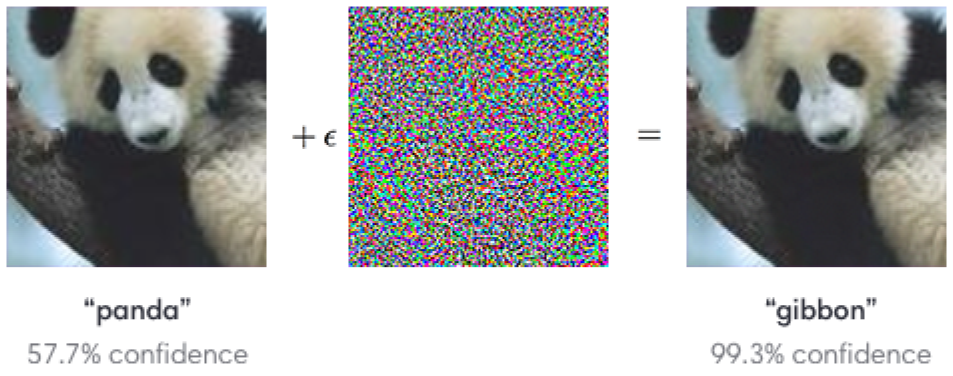
Asmita Limaye (2018B5A70881G)

Varun Singh (2018B5A70869G)

Neural Networks with Recurrent Generative Feedback

Key idea of the paper

Neural networks are vulnerable to input perturbations such as additive noise and adversarial attacks. In contrast, human perception is much more robust to such perturbations.



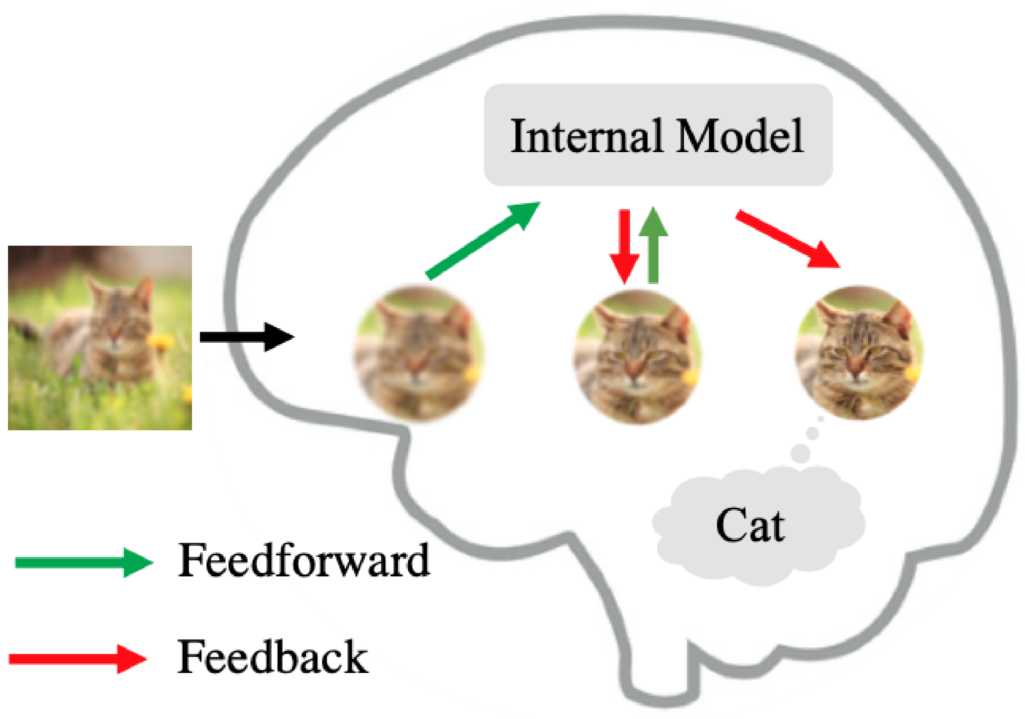
Adding carefully chosen noise (with, say, a projected gradient descent attack) can fool a CNN based classifier into thinking this is a gibbon. To human eyes, though, this is still clearly a panda.

The authors take inspiration from the idea of predictive coding in neuroscience: feedback connections from a higher level of the visual cortical area in the brain carry predictions of lower-level neural activities.

They propose a framework, termed Convolutional Neural Networks with Feedback (CNN-F), with modifications to how layers work to support both **feedforward** and **feedback** passes (the latter should not be confused with backpropagation).

And an adversarial training scheme with loss functions based on outputs of intermediate layers. Their motivation is to train the intermediate layer outputs *for noisy input images* to be the same as those *for clean input images* – making the model robust against (adversarial) noise.

During training, an image is fed forward into a CNN along with its noisy counterpart, and the classification output for the clean image is saved. Then the noisy image is sent **back and forth** through the model for a few cycles. This should get rid of the noise and get a self-consistent classification output, i.e., the classifier output for the data across cycles should not change.



Consider the process of classifying a blurry cat. The blurry input is first processed by feedforward pathways. A label of “cat” is assigned by the internal model. After several cycles of feedforward and feedback, this leads to generation of a sharp image of a cat through the generative pathway. The reason is that the sharp image is the one that is maximally consistent with the input image and the internal generative model.

Adapting layers to work in reverse

Every layer must support “feedback” in addition to standard feedforward. We feed outputs of layers back into them to recover input.

Linear

Applies a linear transform over the input. In the feedback step, a transposed linear transform is applied to the input with the transposed weights of the linear transform.

In the implementation, we have two linear layers with tied weights.

Conv2d

Applies a 2D convolution over the input. In the feedback step, a transposed 2D convolution is applied to the input with the same weights as the 2D convolution.

In the implementation, we achieve this rather conveniently with nn.ConvTranspose2d. Weights are shared by Conv2d and ConvTranspose2d layers.

Initial values of the weights are initialized by sampling from a Gaussian distribution with Mean 0, stddev sqrt (2/num_weights).

InsNorm

Normalize the the input by the norm of each channel. This layer cannot be adapted to work in reverse. (Cannot “unnormalize” outputs fed back to it)

In the feedback pass, we use it as is, but call it on (backward direction) outputs of convolutional/linear layers. As opposed to calling it on the outputs of the layer that comes after it in the feedforward pass.

Unlike batchnorm, instance normalization behaves the same during both training and testing.

ReLU

Applies ReLU activation over the input. In the feedback step, the input is pointwise multiplied through the units that were activated in the forward step — which are saved during the forward step.

In the implementation, to remember which units were not activated in feedforward (and not activate them in feedback) every ReLU unit has a latent variable that notes whether it was activated or not. Their value is reset after every feedforward.

Dropout

Performs dropout regularization to the input. In the feedback step, the same dropout transformation is applied in the backward step.

MaxPool2d

The pixel governing a grid (part of the image being pooled) is the one with the maximum value. This layer behaves thus:

- In the feedforward pass, if the pixel in g that governs this grid is > 0, does a maxpool on this grid.
- In the feedforward pass, if the pixel in g that governs this grid is < 0, do min pool on this grid
- In the feedback pass, if the pixel comes from a max value, put it back to the position of the max value
- In the feedback pass, if the pixel comes from a min value, put it back to the position of the min value

The "put it back to the position of max/min value" is implemented by maintaining hidden variables storing indices of governing pixels identified in feedforward. These are passed to (the very convenient) nn.MaxUnpool2d layer.

MaxPooling cannot be perfectly reversed since the non-governing values are lost in feedforward. In the implementation, latent variables ensure feedback unpools same indices that the immediately preceding feedforward picked out.

AvgPool2d

Similar to MaxPool. Doesn't require storing indices of any governing pixels. Feedforward is a straightforward (heh) windowed average. Feedback uses 2D nearest neighbour upsamping (implemented with nn.UpsamplingNearest2d).

Bias

Adds a bias to the input. In the feedback step, bias is subtracted from the input (i.e. the output being fed back in)

Training methodology and Results

We train the CNNF with pairs of clean and noisy images.

The model is divided into two blocks (block 1 and block 2). Each block consists of convolution, instance normalization, relu, and maxpool in that order for the forward pass.

During feedback, data is sent back through (convolution of the previous layer or input followed by) instance normalization, maxpool, relu, and then convolution. The order is not exact reverse of feedforward because instance normalization cannot be perfectly reversed. So the current order is more sensible.

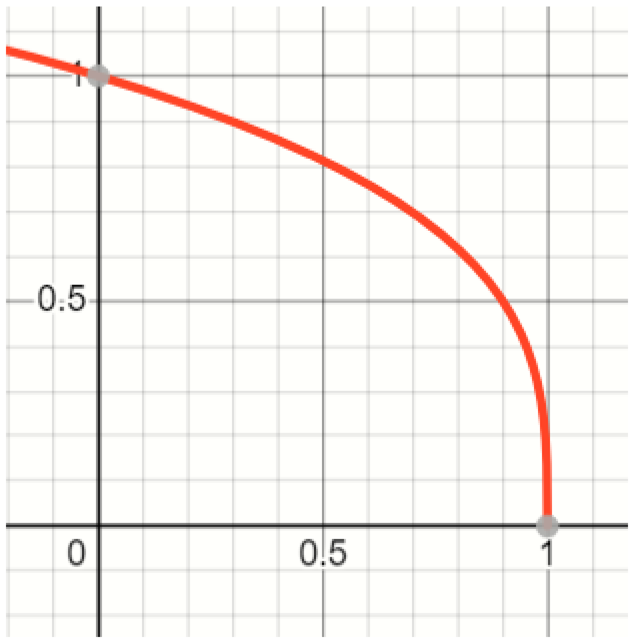
The loss function is calculated by taking five things into consideration:

1. The cross entropy loss between the logits generated by the forward pass of the model and the actual logits of the image.
2. The MSE loss between the reconstructed features of adversarial images (h' for the adversarial images, generated by the backward pass) and the original features of the image (h of the clean images, present before the last forward pass).
3. The MSE loss between the block 1 features of the adversarial images (generated by the backward pass) and the block 1 features of the clean images (generated by the forward pass).
4. The MSE loss between the block 2 features of the adversarial images (generated by the backward pass) and the block 2 features of the clean images (generated by the forward pass).
5. The cross entropy loss between the logits generated by the backward pass of the model and the actual logits of the image.

The block 1 and block 2 are outputs of the intermediate part of the backwards pass: the idea is that the clean and dirty (noisy or adversarial) images should have the same outputs at the intermediate stages.

The optimizer we use is Stochastic Gradient Descent with a polynomial equation scheduling the learning rate. The equation is as follows:

```
base_lr * ((1 - float(iter) / max_iter) ** (power))
```



An illustrative graph of the learning rate schedule. Y-axis is the factor base learning rate is multiplied by, X-axis is the fraction of training steps completed..

The two most relevant hyperparameters for the recurrent generative feedback are: number of cycles for which the data is sent back and forth through the model and the residual parameter ratio. We performed a grid search for 9 pairs of values, our results are tabulated below.

The other hyperparameters (batch size, epochs) as well as the model architecture was chosen on the basis of Google Colab’s restriction on GPU, memory, and session length.

Grid search (We looked at accuracy on test set)

resparam | cycles	1	2	3
0	0.8420	0.8739	0.8397
0.1	0.8779	0.8793	0.8227
0.3	0.8718	0.8807	**0.8939**

After selecting the best values of cycles and the residual update parameter, we proceeded to plot the training loss curves of both CNN and CNN-F to observe that the loss was in fact decreasing.

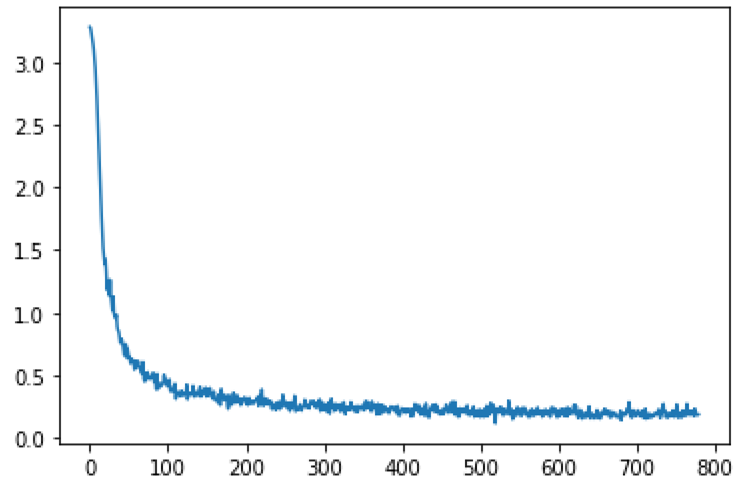


Fig 1. The training loss curve of the vanilla CNN for each batch (5 epochs)

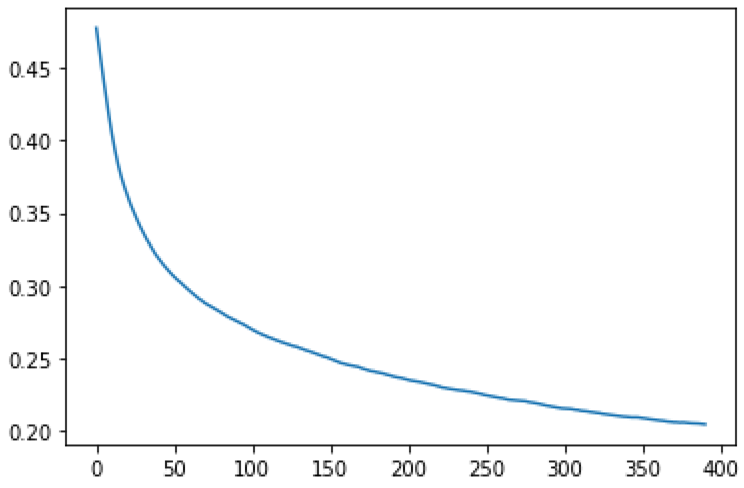


Fig 2. The moving averages of the training loss. The smoothened version shows a clear downward trend.

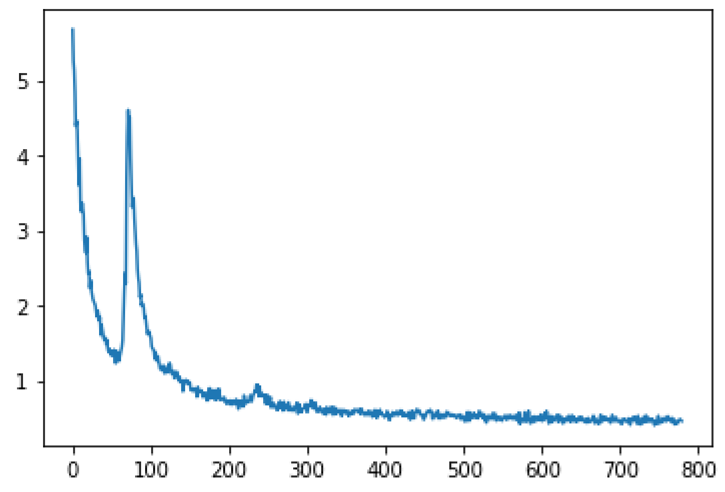


Fig 3. The training loss curve of the CNN-F for each batch (5 epochs)

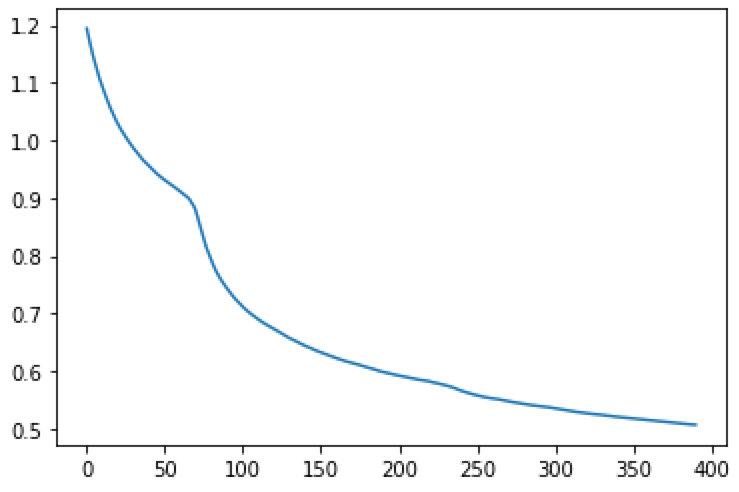


Fig 4. The smoothened training loss curve of the CNN-F. Shows a clear downward trend.

We also found the accuracies of the CNN and CNN-F on clean images and noisy images.

CNN on clean images	0.923
CNN on noisy images	0.839
CNN-F on clean images	0.780
CNN-F on noisy images	0.888

Confusion matrices:

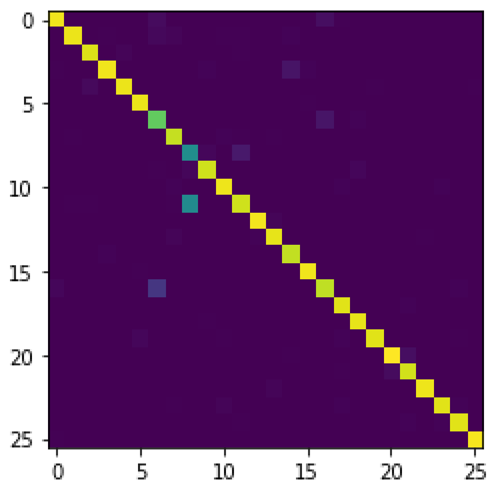


Fig 5. The confusion matrix of the CNN on clean images.

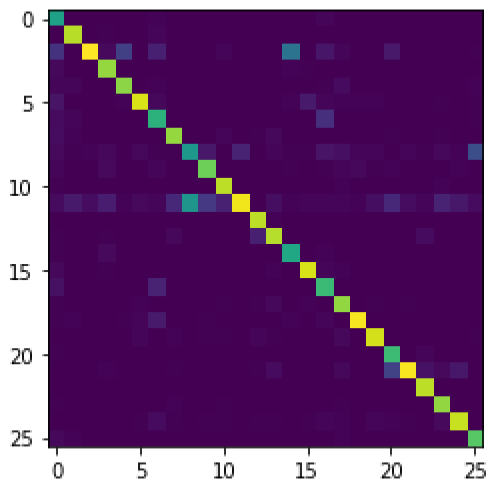


Fig 6. The confusion matrix of the CNN-F on clean images

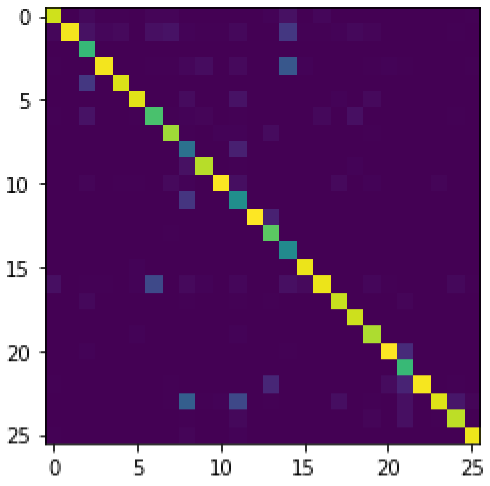


Fig 7. Confusion matrix of the CNN on noisy images

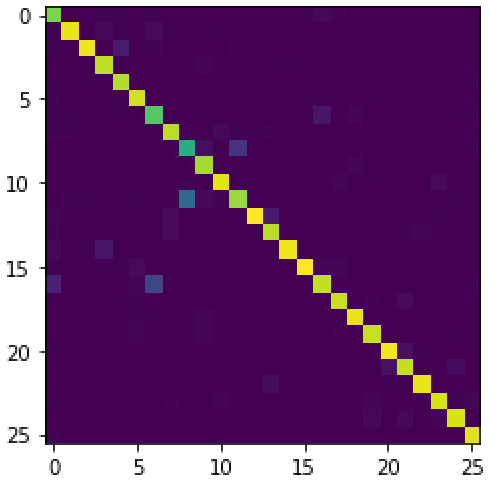


Fig 8. Confusion matrix of the CNN-F on noisy images

The CNN performs better than the CNNF on clean images. This is not surprising since the CNNF needs more training to reach comparable performance (Google Colab session lengths are too limited).

On noisy images, the CNNF outperforms the CNN. Fig 8. has clearer hot diagonal than Fig 7.

The paper shows that the CNNF is also much more robust against adversarial attacks of increasing strengths compared to a CNN. We were unable to verify this due to constraints.

Our Contributions

- We implemented a simple version of the model architecture (CNNF: Convolutional Neural Network with recurrent generative feedback) presented in the paper, with implementations of layers supporting feedforward and feedback.
- We implemented a standard CNN model with the exact same architecture as the CNNF in notebook (but no feedback) to compare it with. This lets us remark on whether the whole scheme makes a significant difference.
- We implemented the training framework explained in the paper with noisy images (random noise). We use noisy images instead of adversarial examples because generating them in the train loop makes training too costly for Google Colab.
- We do all this with a new dataset we chose during midsems EMNIST Letters.We preprocessed it to make it equivalent to the FashionMNIST dataset used in the paper.
- We demonstrated an untargeted adversarial attack on CNN [Midsem Submission]. Fast gradient sign method
- We demonstrated a targeted adversarial attack on CNN. [Midsem Submission] Projected Gradient Descent.
- We compared CNN vs CNNF on clean and noisy images and have furnished accuracy tabulations, loss curves, and confusion matrices.