Style Transfer with TensorFlow™

BOSTON UNIVERSITY
MACHINE INTELLIGENCE COMMUNITY

Rachel Manzelli
4/25/18

https://goo.gl/gzcWDb

M I C

So far...

- → Learned the concepts of TensorFlow 1.2
- → Built and evaluated simple graphs
- → "Trained" a linear model
- → Learned about neural networks, VGG-19, and NST
- → Saw an implementation of NST
- → Began exploring pretrained VGG-19

https://github.com/bumic/TF-Workshops

M I C

Today's Plan

- ➔ *Finish project:* implementation of neural style transfer in TensorFlow
- ➔ *Implementations:* architecture/VGG implementation, training implementation

What You Need For Today

- A computer & text editor
- Installations of Python 2 or 3, TensorFlow, numpy, scipy, and Pillow
- These can all be installed via `pip install`

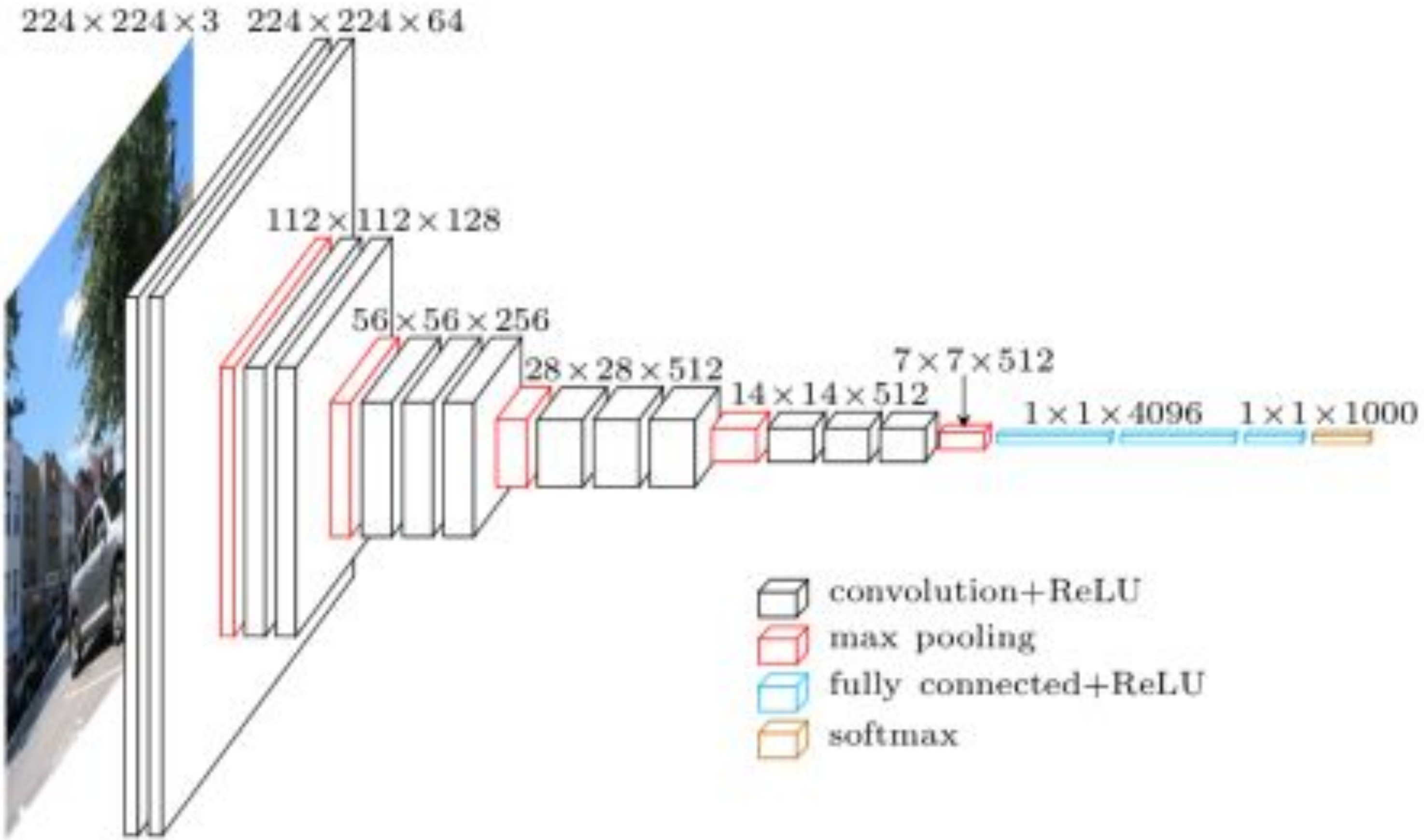
```
>> pip install numpy

>> pip install scipy

>> pip install Pillow
```

M I C

VGG-19

- A specific version of VGGNet with **19 weight layers**
- VGGNet is a convolutional neural network originally used for image classification
  - uses **3×3 convolutional layers** stacked on top of each other
  - reducing volume size is handled by **max pooling**
  - Two **fully-connected** layers, each with 4,096 nodes are followed by a **softmax classifier.**
- Original VGGNet paper: https://arxiv.org/abs/1409.1556

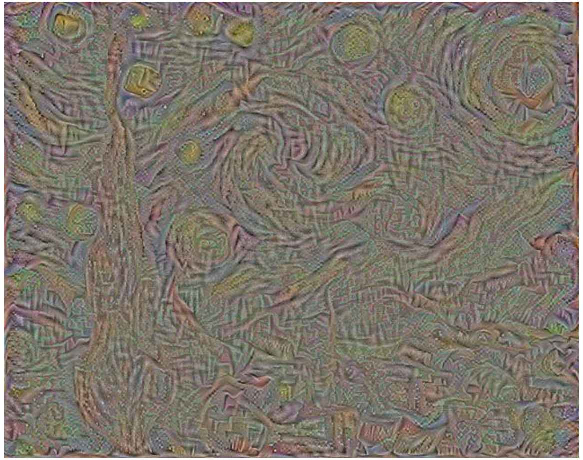
M I C


224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax

M I C

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Implementation of Neural Style Transfer



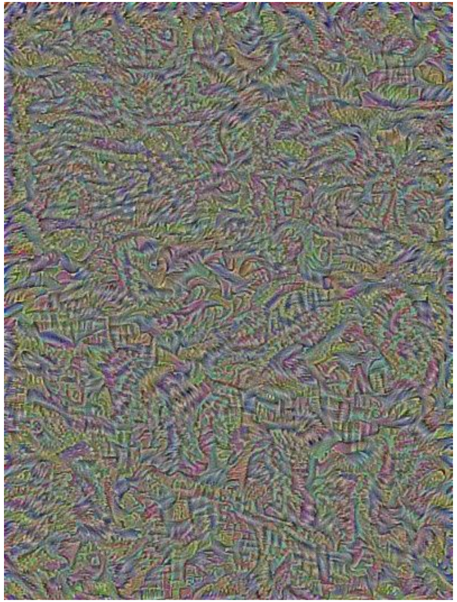
Iteration 10

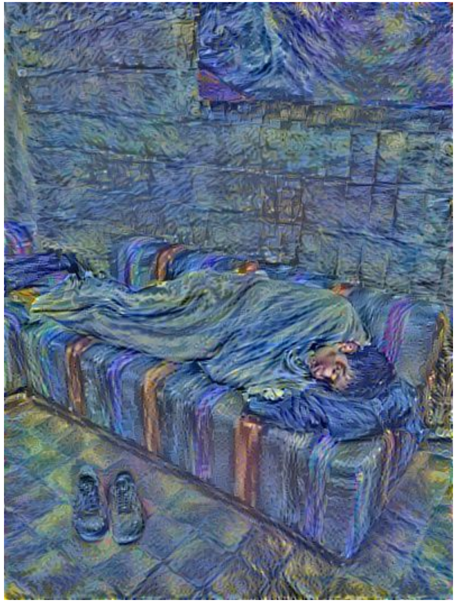
Iteration 300

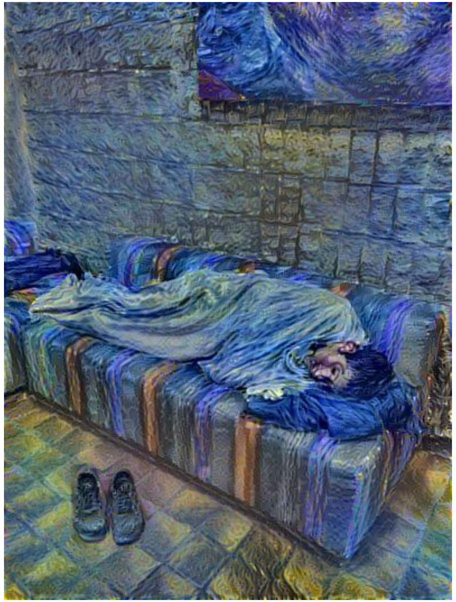
Iteration 990

M I C

Implementation of Neural Style Transfer


Iteration 10


Iteration 300


Iteration 990

M I C

Total Loss Function for NST

- **Total Loss** - Jointly minimize error of a white noise image from
  - Content representation of **content image** $\vec{p}$
  - Style representation of **style image** $\vec{a}$

$$L_{total}(\overrightarrow{p}, \overrightarrow{a}, \overrightarrow{x}) = \alpha L_{content}(\overrightarrow{p}, \overrightarrow{x}) + \beta L_{style}(\overrightarrow{a}, \overrightarrow{x})$$

- α, β : weighting factors for content and style reconstruction respectively

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \left(\alpha \mathcal{L}_{\text{content}}(\mathbf{c}, \mathbf{x}) + \beta \mathcal{L}_{\text{style}}(\mathbf{s}, \mathbf{x})\right)$$

M I C

Let's start building it!

- ➔ Go to https://github.com/anishathalye/neural-style
- ➔ This is the NST implementation we will be rebuilding.
- ➔ Download the file at the bottom:

`imagenet-vgg-verydeep-19.mat`

- ➔ This is our pretrained VGGNet that we will be using in our implementation. (pre-training a convnet is time-consuming!)

M I C

The Pre-trained Network

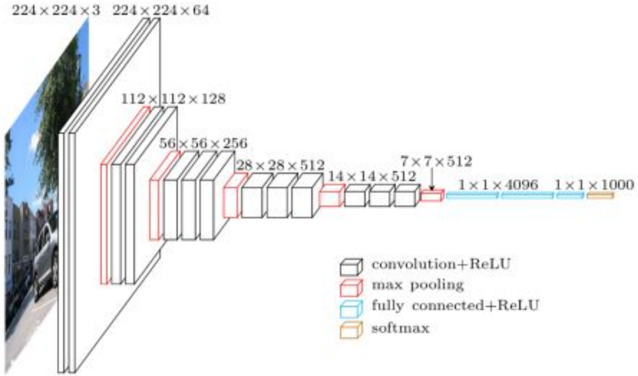
- This network was trained on ImageNet (it can presumably classify images)
- Open the file in MATLAB
- If you don't have MATLAB, that's ok - Python to come!

Implementing a Neural Network

- *Architecting*: constructing the architecture of the neural network
- *Preprocessing*: preprocessing the data so that it is in a form we can train on
- *Training*: getting those perfect weights
- *Testing/generating*: using the trained network to generate a result

Implementation: Architecture

- ➔ Open the template file `vgg_template.py` from our GitHub. This will be our initialization of the architecture of the network.
- ➔ First, we will initialize a dictionary of keys to refer back to the weight layers of our network, based on the .mat file.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

```python
VGG19_LAYERS = (
    'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',

    'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',

    'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3',
    'relu3_3', 'conv3_4', 'relu3_4', 'pool3',

    'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3',
    'relu4_3', 'conv4_4', 'relu4_4', 'pool4',

    'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3',
    'relu5_3', 'conv5_4', 'relu5_4'
)
```

Parsing Network Data

➔ We will now get wanted information from the pretrained VGG-19.
➔ We want the weights and mean pixel value to do our computations with.
➔ We also want to translate the given layers into functions that correspond with that layer (this is how the data is labeled)

M I C

```python
def load_net(data_path):
    data = scipy.io.loadmat(data_path)
    mean = data['normalization'][0][0][0]
    mean_pixel = np.mean(mean, axis=(0, 1))
    weights = data['layers'][0]
    return weights, mean_pixel
```

Defining Layer Operations

- ➔ Now, we define our max pooling and convolutional operations as layers of the network that we can call upon later.
- ➔ We also add preprocess and unprocess functions for the loss calculations.

M I C

```python
def _conv_layer(input, weights, bias):
    conv = tf.nn.conv2d(input, tf.constant(weights), strides=(1, 1, 1, 1),
            padding='SAME')
    return tf.nn.bias_add(conv, bias)


def _pool_layer(input, pooling):
    if pooling == 'avg':
        return tf.nn.avg_pool(input, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1),
                padding='SAME')
    else:
        return tf.nn.max_pool(input, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1),
                padding='SAME')

def preprocess(image, mean_pixel):
    return image - mean_pixel


def unprocess(image, mean_pixel):
    return image + mean_pixel
```

```python
def net_preloaded(weights, input_image, pooling):
    net = {}
    current = input_image
    for i, name in enumerate(VGG19_LAYERS):
        kind = name[:4] #First 4 letters of the layer name
        if kind == 'conv':
            kernels, bias = weights[i][0][0][0][0]
            # matconvnet: weights are [width, height, in_channels, out_channels]
            # tensorflow: weights are [height, width, in_channels, out_channels]
            kernels = np.transpose(kernels, (1, 0, 2, 3))
            bias = bias.reshape(-1)
            current = _conv_layer(current, kernels, bias)
        elif kind == 'relu':
            current = tf.nn.relu(current)
        elif kind == 'pool':
            current = _pool_layer(current, pooling)
        net[name] = current

    assert len(net) == len(VGG19_LAYERS)
    return net
```

Done with Architecture!

MIC

Neural Networks: Training

- Training a network means minimizing the cost function more and more with each forward pass through the network (in our case, that's the total loss = style loss + content loss)

$$L_{total}(\overrightarrow{p}, \overrightarrow{a}, \overrightarrow{x}) = \alpha L_{content}(\overrightarrow{p}, \overrightarrow{x}) + \beta L_{style}(\overrightarrow{a}, \overrightarrow{x})$$

MIC

Feeding forward through the network

- ➔ We are going to build the **static training graph** first, and then evaluate it through backpropagation
- ➔ To compute the loss, we want to feed the input forward through the network, which just means evaluating the activations at each layer of the network one time. (We'll repeat during training)

M I C

```python
# compute content features in feedforward mode
g = tf.Graph()
with g.as_default(), g.device('/cpu:0'), tf.Session() as sess:
    image = tf.placeholder('float', shape=shape)
    net = vgg.net_preloaded(vgg_weights, image, pooling)
    content_pre = np.array([vgg.preprocess(content, vgg_mean_pixel)])
    for layer in CONTENT_LAYERS:
        content_features[layer] = net[layer].eval(feed_dict={image: content_pre})

# compute style features in feedforward mode
for i in range(len(styles)):
    g = tf.Graph()
    with g.as_default(), g.device('/cpu:0'), tf.Session() as sess:
        image = tf.placeholder('float', shape=style_shapes[i])
        net = vgg.net_preloaded(vgg_weights, image, pooling)
        style_pre = np.array([vgg.preprocess(styles[i], vgg_mean_pixel)])
        for layer in STYLE_LAYERS:
            features = net[layer].eval(feed_dict={image: style_pre})
            features = np.reshape(features, (-1, features.shape[3]))
            gram = np.matmul(features.T, features) / features.size
            style_features[i][layer] = gram

initial_content_noise_coeff = 1.0 - initial_noiseblend
```

M I C

Content Loss

```
content_layers_weights = {}
content_layers_weights['relu4_2'] = content_weight_blend
content_layers_weights['relu5_2'] = 1.0 - content_weight_blend

content_loss = 0
content_losses = []
for content_layer in CONTENT_LAYERS:
    content_losses.append(content_layers_weights[content_layer] * content_weight * (2 * tf.nn.l2_loss(
            net[content_layer] - content_features[content_layer]) /
            content_features[content_layer].size))
content_loss += reduce(tf.add, content_losses)
```

M I C

Gram Matrix

3. Let $(V, \langle \cdot, \cdot \rangle)$ be a Euclidean space. The Gram matrix of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k \in V$ is

$$G(\mathbf{v}_1, \ldots, \mathbf{v}_k) = \begin{pmatrix} \langle \mathbf{v}_1, \mathbf{v}_1 \rangle & \cdots & \langle \mathbf{v}_1, \mathbf{v}_k \rangle \\ \vdots & & \vdots \\ \langle \mathbf{v}_k, \mathbf{v}_1 \rangle & \cdots & \langle \mathbf{v}_k, \mathbf{v}_k \rangle \end{pmatrix}.$$

M I C

Style Loss

```
style_loss = 0
for i in range(len(styles)):
    style_losses = []
    for style_layer in STYLE_LAYERS:
        layer = net[style_layer]
        _, height, width, number = map(lambda i: i.value, layer.get_shape())
        size = height * width * number
        feats = tf.reshape(layer, (-1, number))
        gram = tf.matmul(tf.transpose(feats), feats) / size
        style_gram = style_features[i][style_layer]
        style_losses.append(style_layers_weights[style_layer] * 2 * tf.nn.l2_loss(gram - style_gram) / style_gram.size)
    style_loss += style_weight * style_blend_weights[i] * reduce(tf.add, style_losses)
```

M I C

Computing Total Loss

```
loss = content_loss + style_loss + tv_loss
```

Total variation denoising loss

Neural Networks: Training

- **Gradient descent** is a classical optimizer: it allows us to **adjust weights and biases** by telling them to increase or decrease to minimize the cost function
- We are going to use the **Adam optimizer**, which is a combination of **AdaGrad** and **RMSProp**.
  - Per-parameter learning rate, adapted based on how quickly the weights are changing
  - Calculates exponential moving averages of the gradient and the squared gradient

M I C

Gradient Descent vs. ADAM

$$\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n)$$

$$\downarrow$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

M I C

Minimizing loss with ADAM

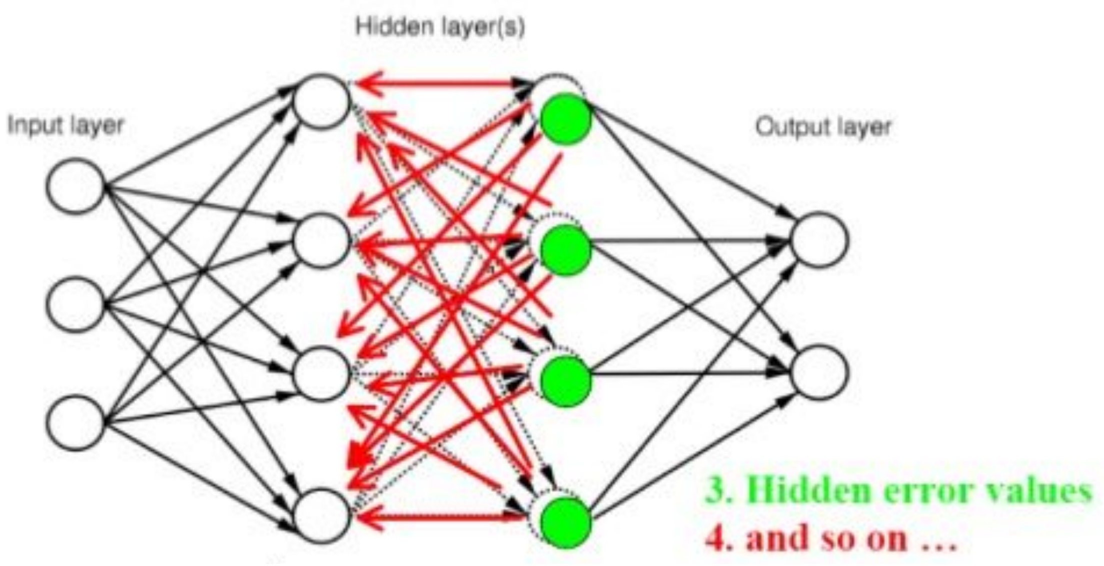
```
train_step = tf.train.AdamOptimizer(learning_rate,
    beta1, beta2, epsilon).minimize(loss)
```

$$\alpha, \beta$$

M I C

Neural Networks: Training

➔ We update our weights by **backpropagation**



M I C

```python
this_loss = loss.eval()
if this_loss < best_loss:
    best_loss = this_loss
    best = image.eval()
```

MIC

Run it!

- Make sure to change your file names to `vgg.py` and `stylize.py`
- Also, include all required arguments
  - To see these, run

```
python neural_style.py --help
```

```
python neural_style.py --content <content file>
--styles <style file> --output <output file>
```

M I C

Thanks for joining us!

M I C

Thank you to our sponsors!