Deep Learning Lab 12-2: Visualization & Style Transfer

Ruei-Yang Su & DataLab

Department of Computer Science,
National Tsing Hua University,
Taiwan

What are deep ConvNets learning?

Outline

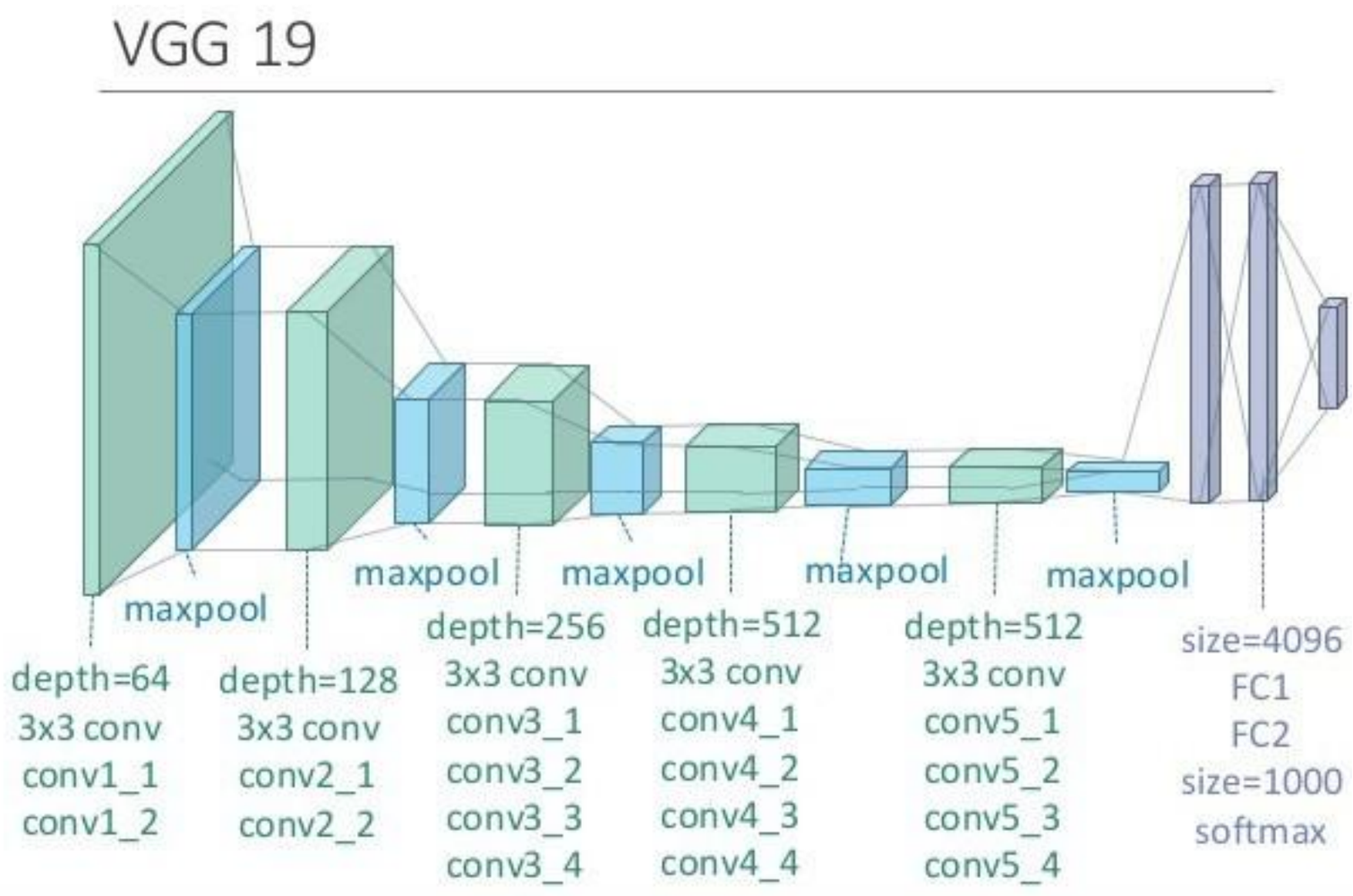
- Visualization

- Neural Style Transfer
  - A Neural Algorithm of Artistic Style
  - AdaIN (Adaptive Instance Normalization)

- Save and Load Models

VGG19

- VGG19 is known for its simplicity, using only 3×3 convolutional layers stacked on top of each other in increasing depth



VGG19

- Convolutional Neural Networks learns how to create useful representations of the data to differentiate between different classes

VGG19

- In this tutorial, we are going to use VGG19 pretrained on ImageNet for visualization

- ImageNet is a large dataset used in ImageNet Large Scale Visual Recognition Challenge(ILSVRC). The training dataset contains around 1.2 million images composed of 1000 different types of objects

More about Pretrained Networks…

- Pretrained network is useful and convenient for several further usages, such as **style transfer**, **transfer learning**, **fine-tuning**, and so on

- Generally, using pretrained network can save a lot of time and also easier to train a model on more complex dataset or small dataset

Outline

- Visualization

- Neural Style Transfer
  - A Neural Algorithm of Artistic Style

  - AdaIN (Adaptive Instance Normalization)

- Save and Load Models

Visualize Filters

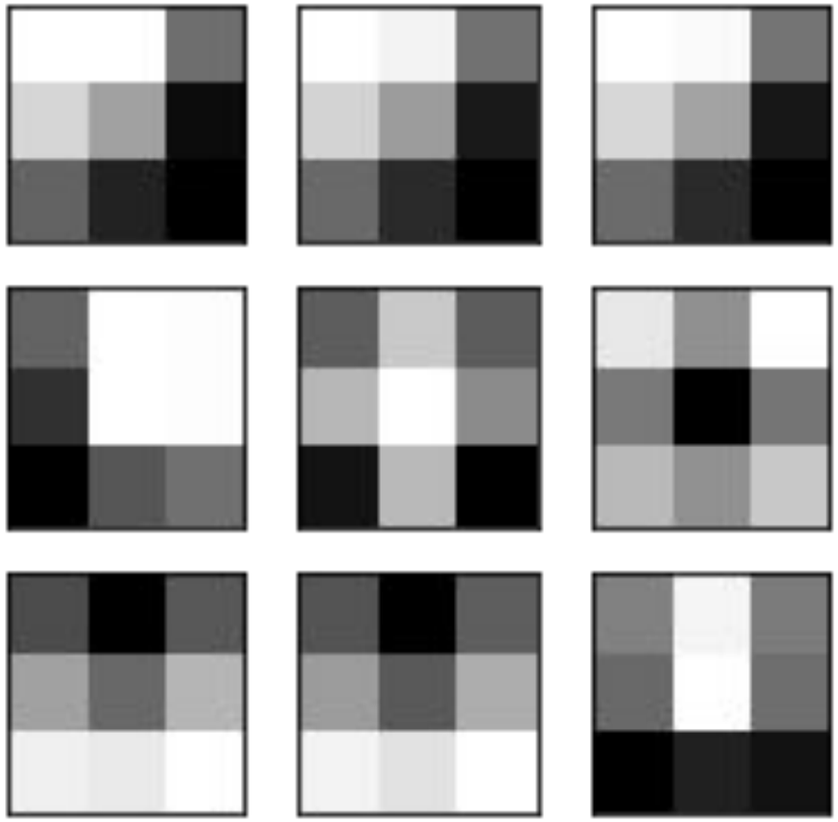
- Visualize the weights of the convolution filters to help us understand **what neural network have learned**

- Learned filters are simply weights, yet because of the specialized two-dimensional structure of the filters, the weight values have a spatial relationship to each other and plotting each filter as a two-dimensional image is meaningful (or could be)

Visualize Filters

- Let's look at some filters in the first convolutional layer

- The dark squares indicate small or inhibitory weights and the light squares represent large or excitatory weights



Visualize Feature Maps

- Activation maps, called feature maps, capture the result of applying the filters to input, such as the input image or another feature map

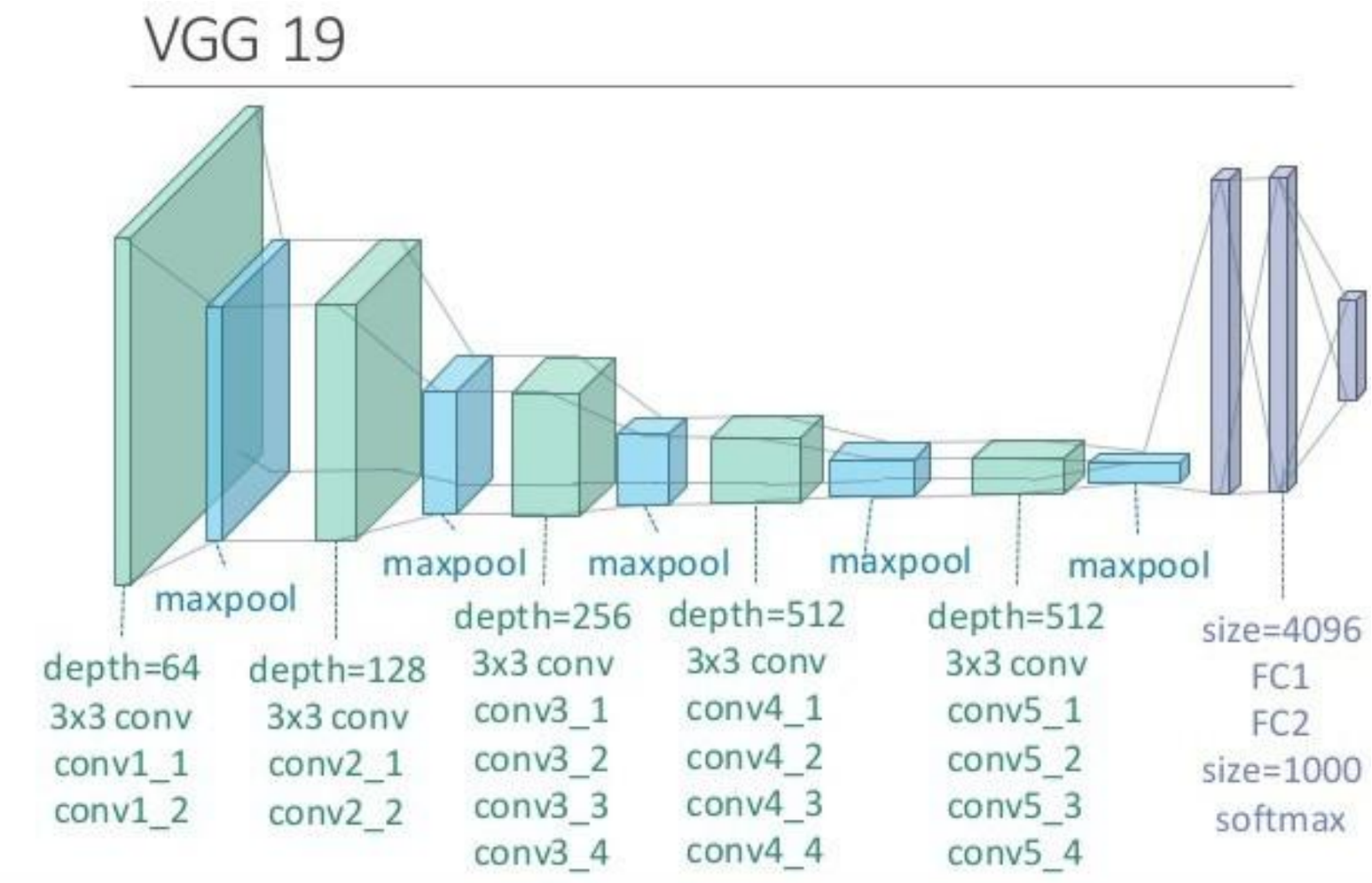
- The idea of visualizing a feature map for a specific input image would be to understand **what features of the input are detected or preserved** in the feature maps

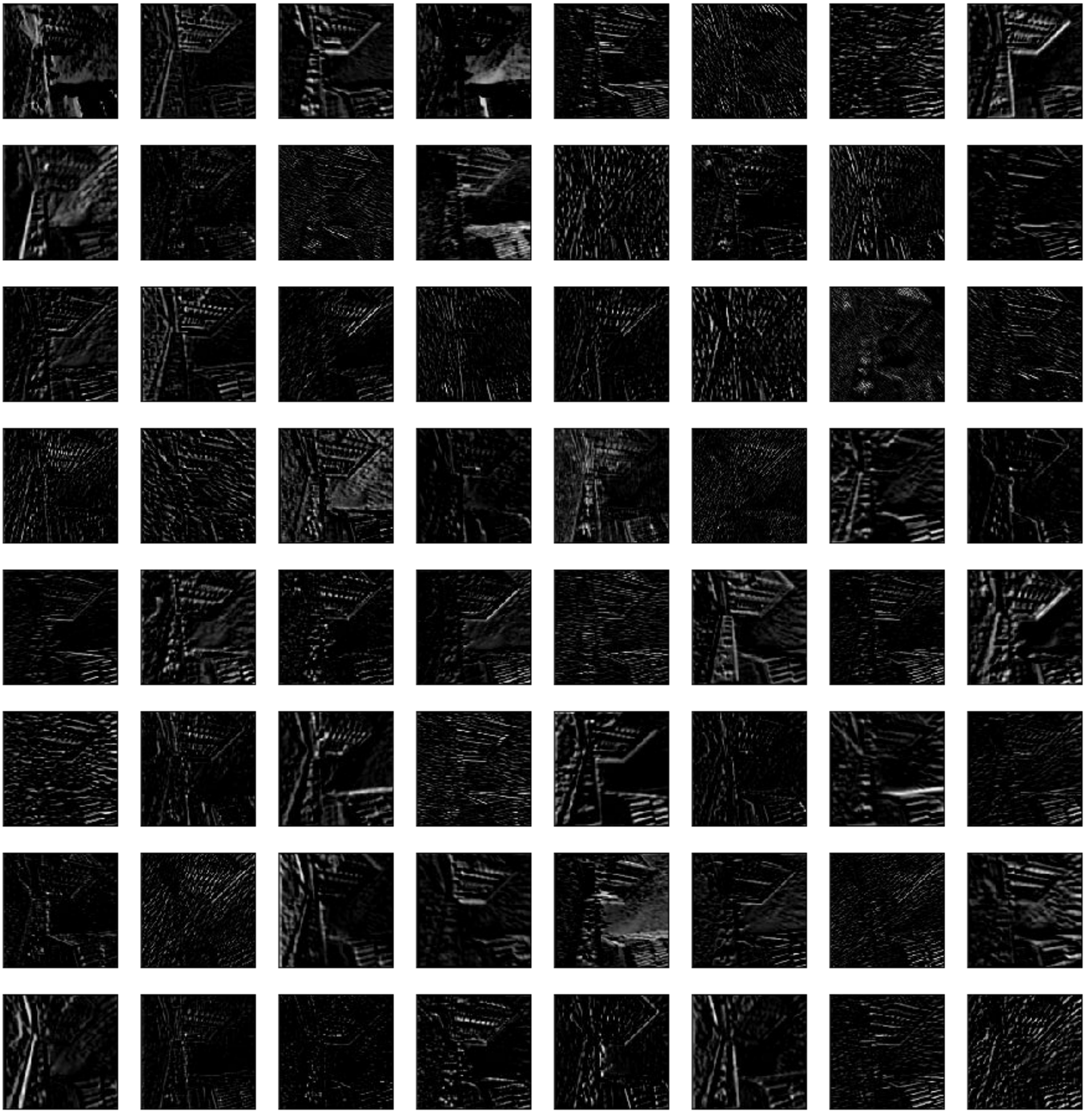
Visualize Feature Maps

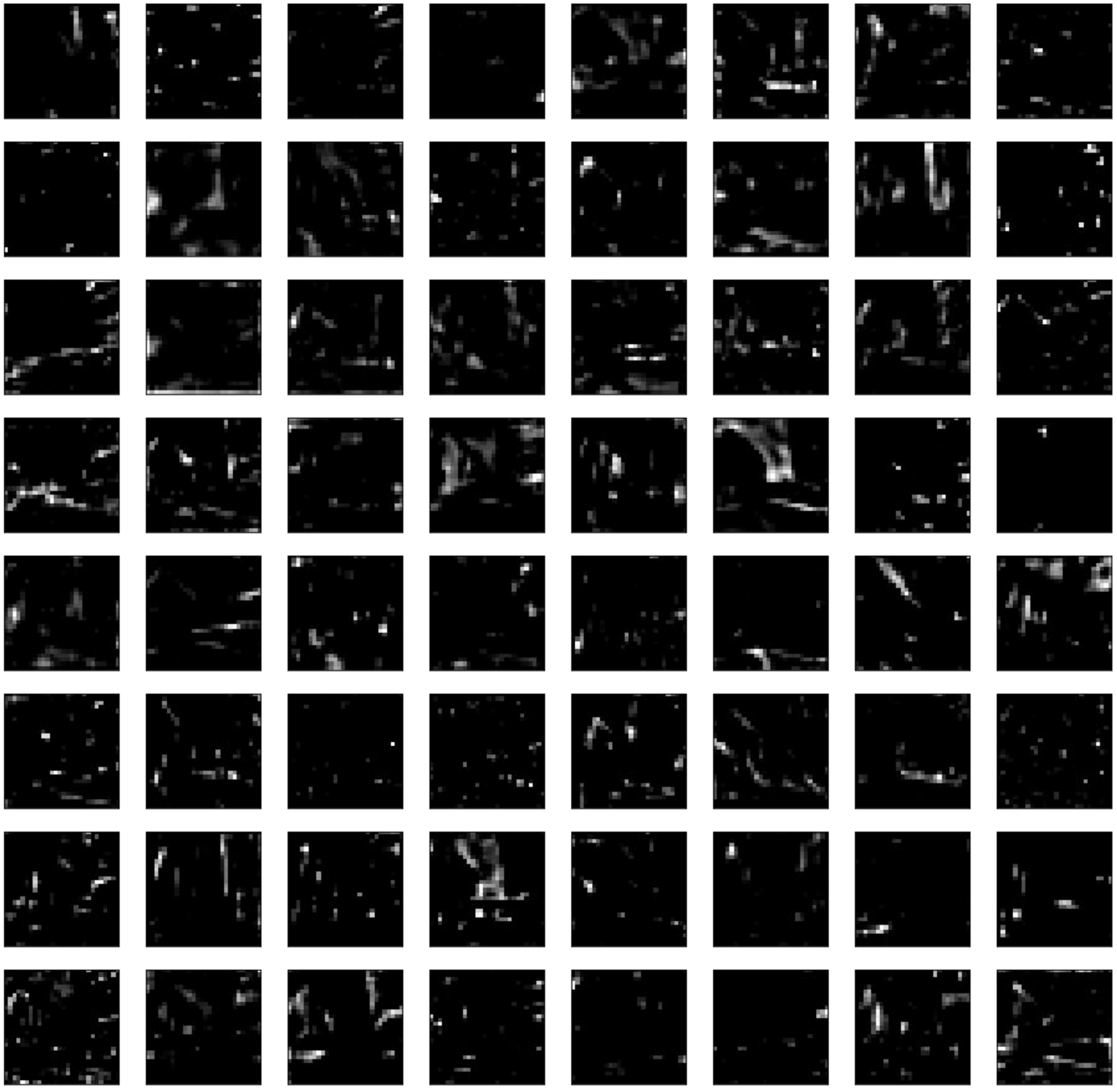
- Result of applying the filters in the first convolutional layer is a lot of versions of the input image with different features highlighted

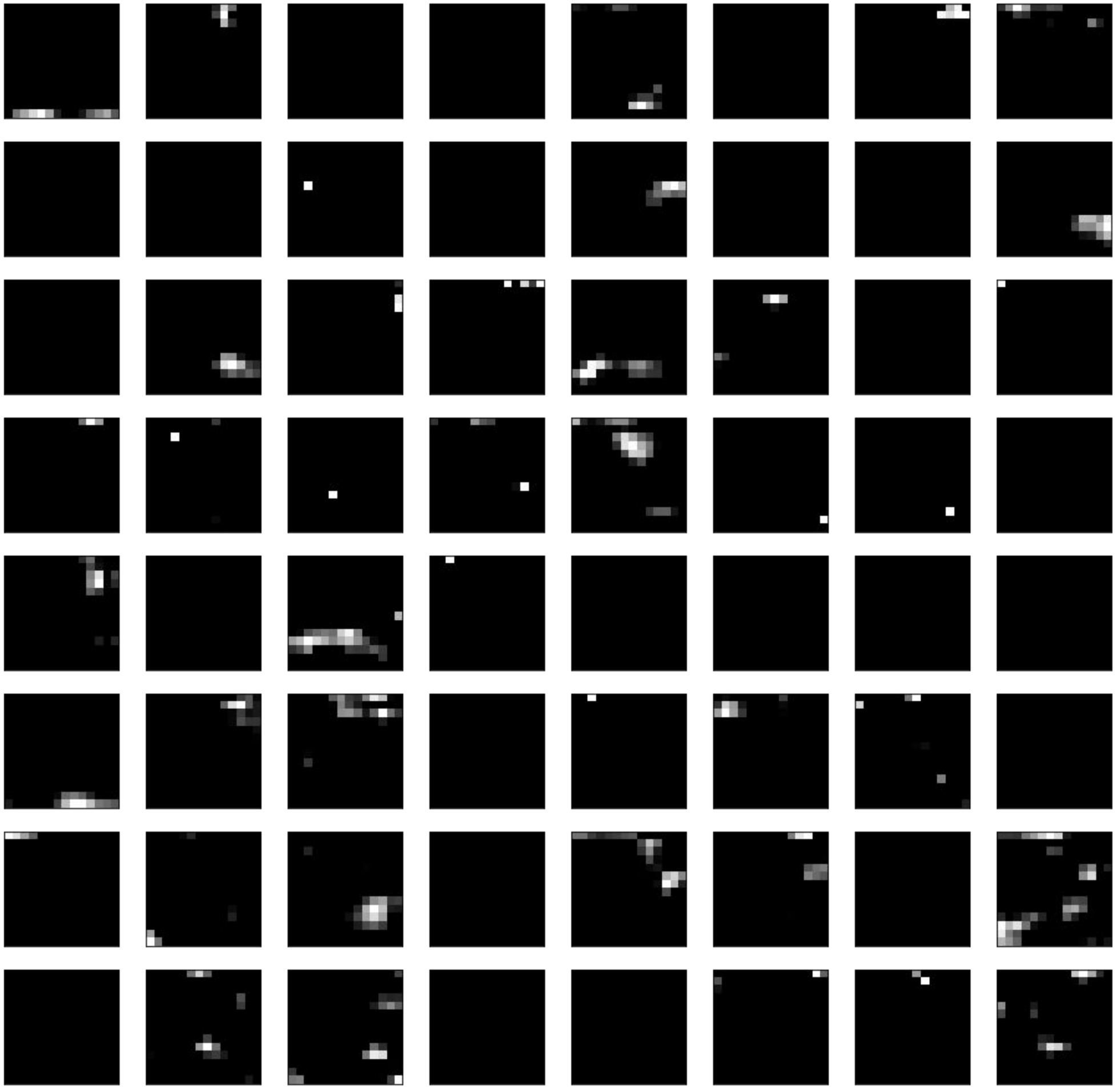
Visualize “Deeper” Feature Maps

- Let's visualize the feature maps output from each block of VGG19



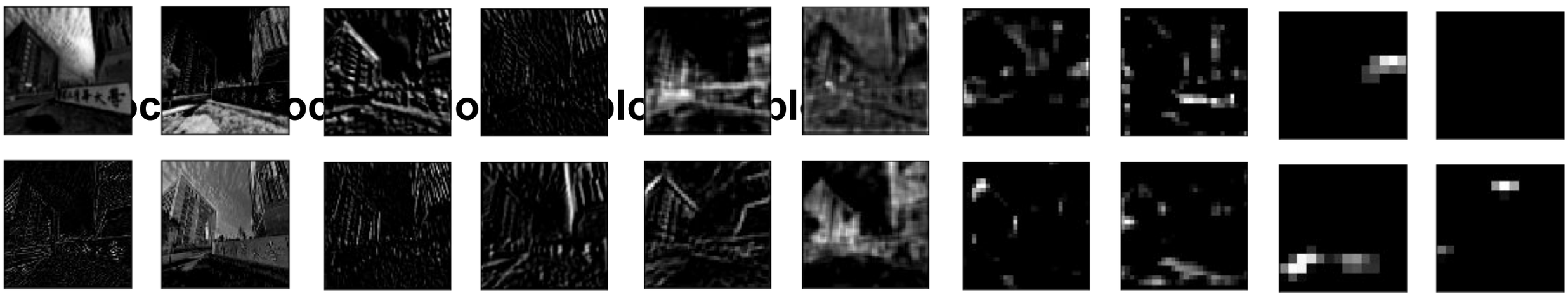






Visualize “Deeper” Feature Maps

- Feature maps closer to the input of the model capture a lot of fine detail in the image and that as we progress deeper into the model, the feature maps show less and less detail
- This pattern was to be expected, as the model abstracts the features from the image into more general concepts that can be used to make a classification



Visualize Gradients

- Visualizing convolutional output is a pretty useful technique for visualizing shallow convolution layers

- Unfortunately, when we get into the deeper layers, it's hard to understand them just by just looking at the convolution output

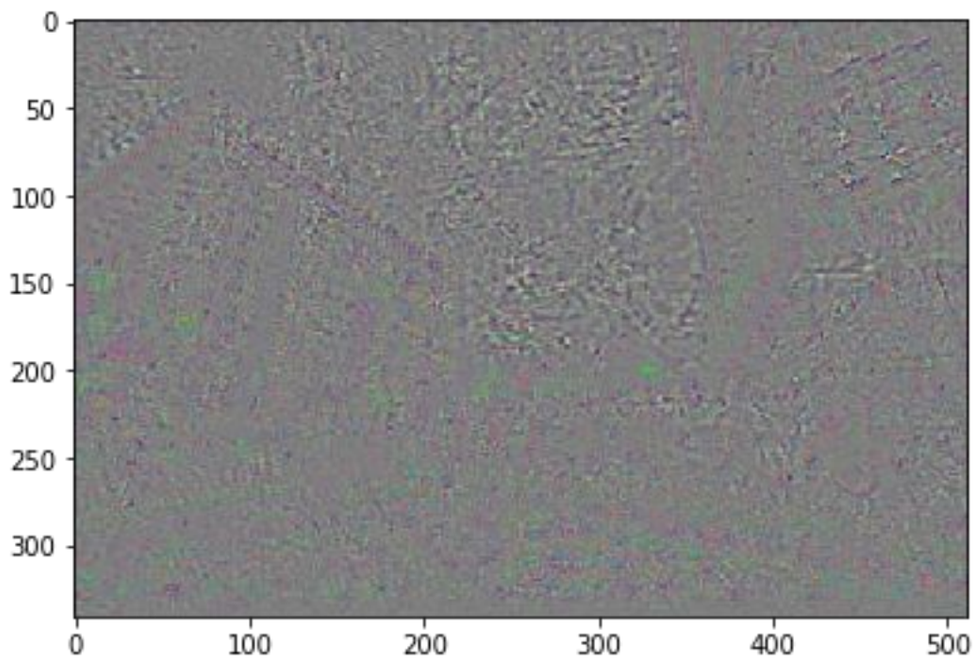
Visualize Gradients

- If we want to understand what the deeper layers are really doing, we can try to use backpropagation to show us the gradients of a particular neuron with respect to our input image, which is called **saliency map**

- We will make a forward pass up to the layer that we are interested in, and then backpropagate to help us understand **which pixels contributed the most** to the activation of that layer

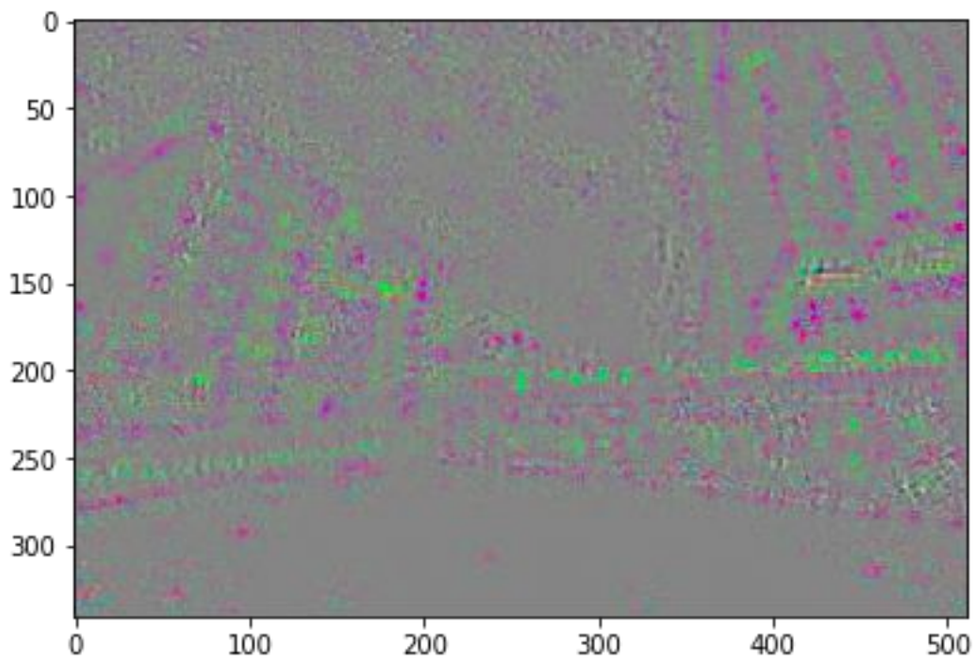
Visualize Gradients

- Compute the gradient of maximum neurons among all activations in the required layer with respect to the input image



Visualize Gradients

- We can also visualize the gradient of any single feature map



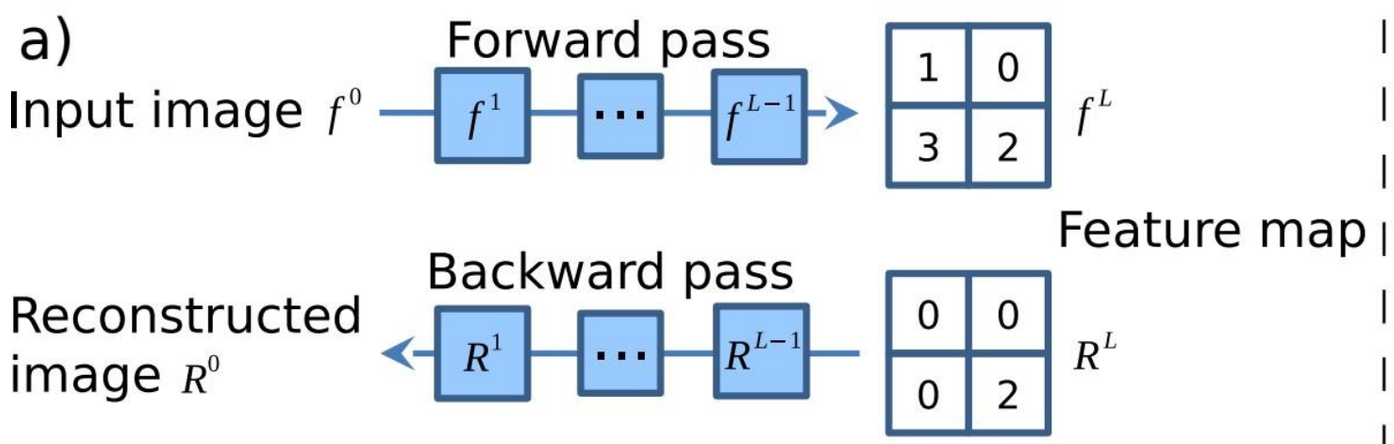
Guided-Backpropagation

- As we can see above, the results are still hard to explain and not very satisfying

- Ideally, neurons act like detectors of particular image features. We are only interested in **what image features the neuron detects**, not in what kind of stuff it doesn't detect. Therefore, when propagating the gradient, we set all the negative gradients to 0

Guided-Backpropagation

- Thus, the gradient is **“guided” by both the input and the error signal**



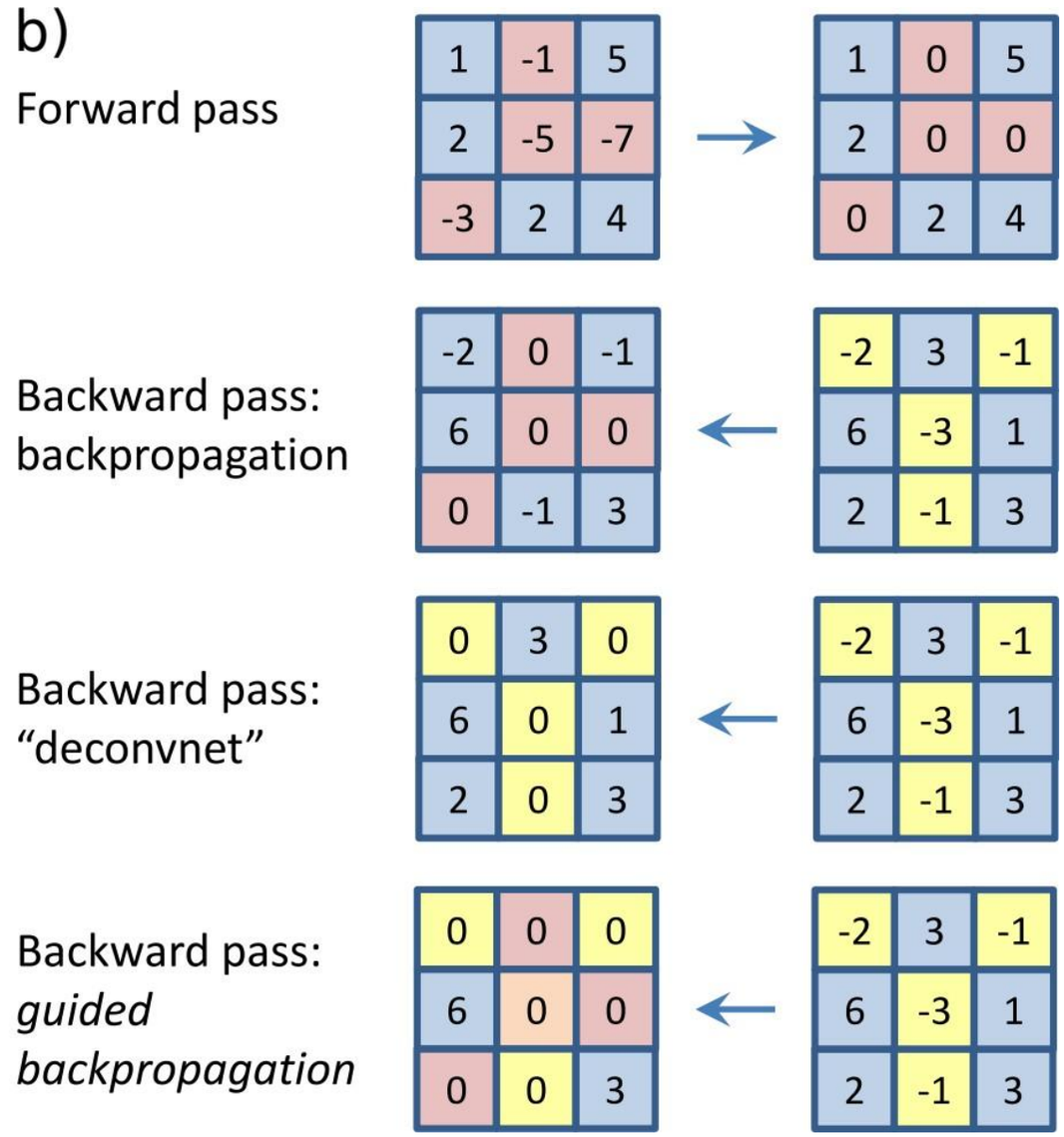
c)

activation: $f_i^{l+1} = relu(f_i^l) = \max(f_i^l, 0)$

backpropagation: $R_i^l = (f_i^l > 0) \cdot R_i^{l+1}$, where $R_i^{l+1} = \frac{\partial f^{out}}{\partial f_i^{l+1}}$

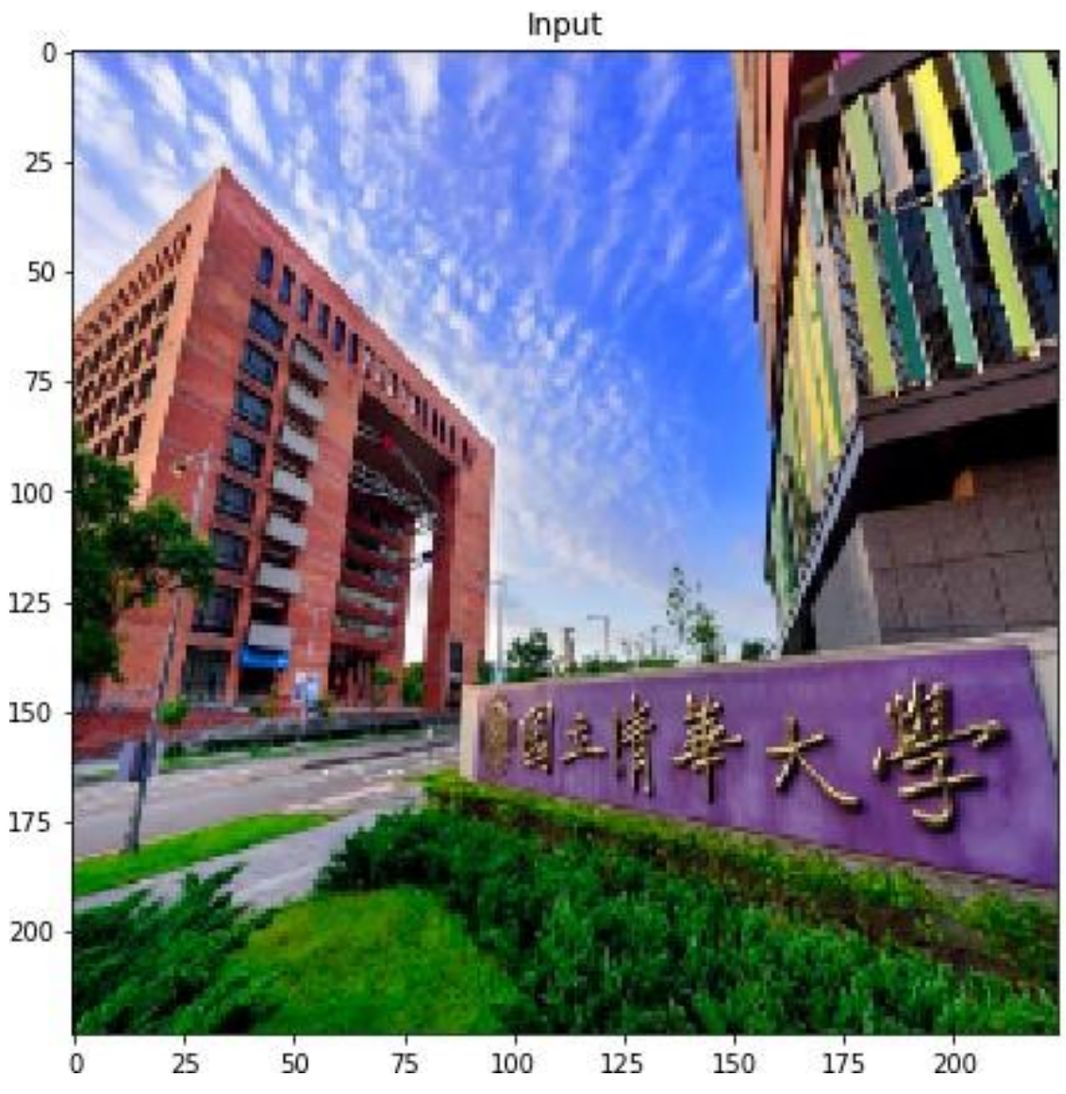
backward 'deconvnet': $R_i^l = (R_i^{l+1} > 0) \cdot R_i^{l+1}$

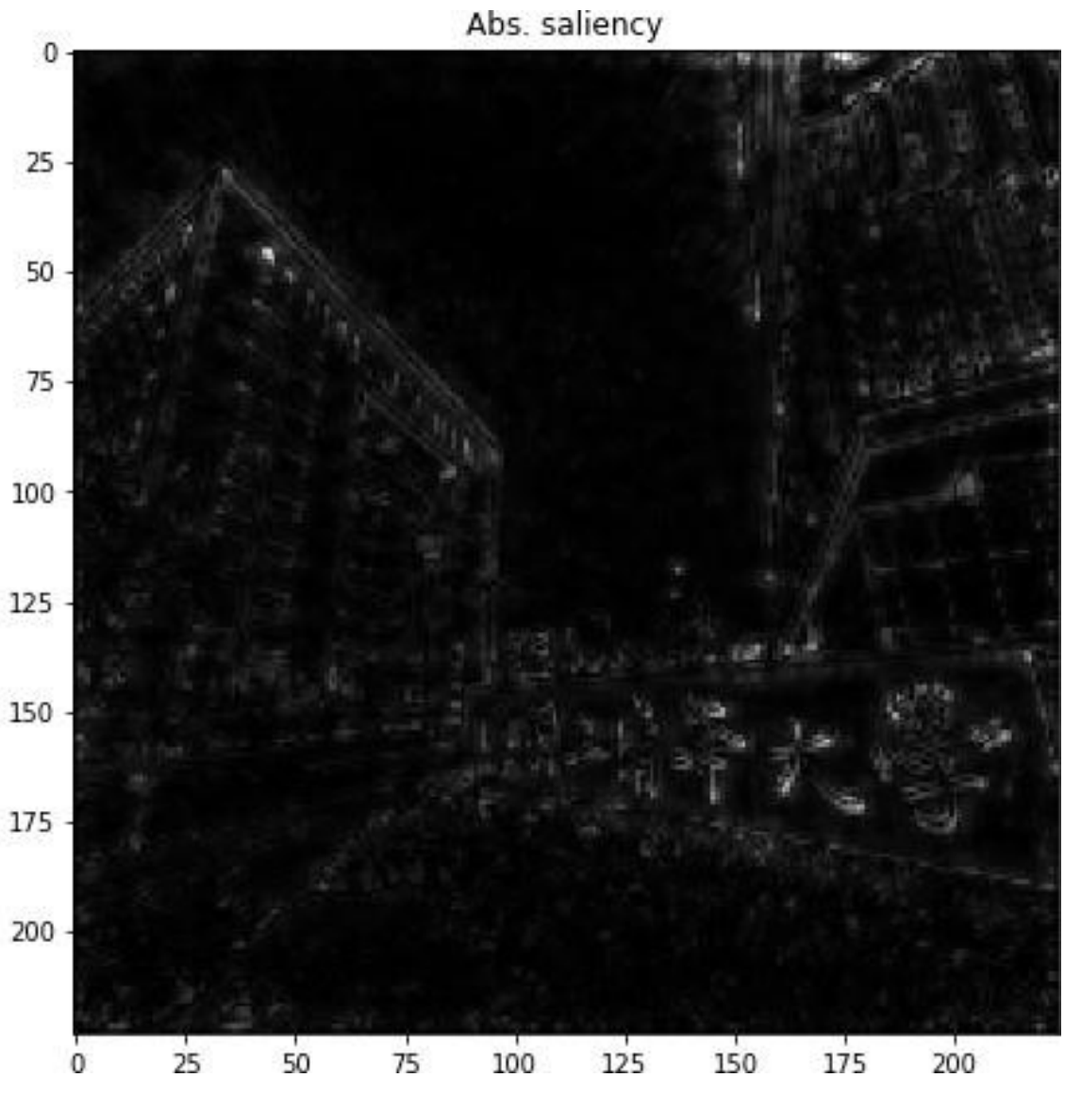
guided backpropagation: $R_i^l = (f_i^l > 0) \cdot (R_i^{l+1} > 0) \cdot R_i^{l+1}$



Guided-Backpropagation

- Thus, the gradient is “guided” by both the input and the error signal





Outline

- Visualization

- Neural Style Transfer
  - A Neural Algorithm of Artistic Style

  - AdaIN (Adaptive Instance Normalization)

- Save and Load Models

Neural Style Transfer

- Leon Gatys and his co-authors has a very interesting work called "A Neural Algorithm of Artistic Style" that uses neural representations to separate and recombine content and style of arbitrary images, providing a neural algorithm for the creation of artistic images

Neural Style Transfer

Outline

- Visualization

- Neural Style Transfer
  - A Neural Algorithm of Artistic Style

  - AdaIN (Adaptive Instance Normalization)

- Save and Load Models

A Neural Algorithm of Artistic Style

- At a high level, in order for a network to perform image classification, it must understand the image

- This requires taking the raw image as input pixels and building an internal representation that converts the raw image pixels into a complex understanding of the features present within the image

A Neural Algorithm of Artistic Style

- Use the intermediate layers of the model to get the content and style representations of the image

- Content representation

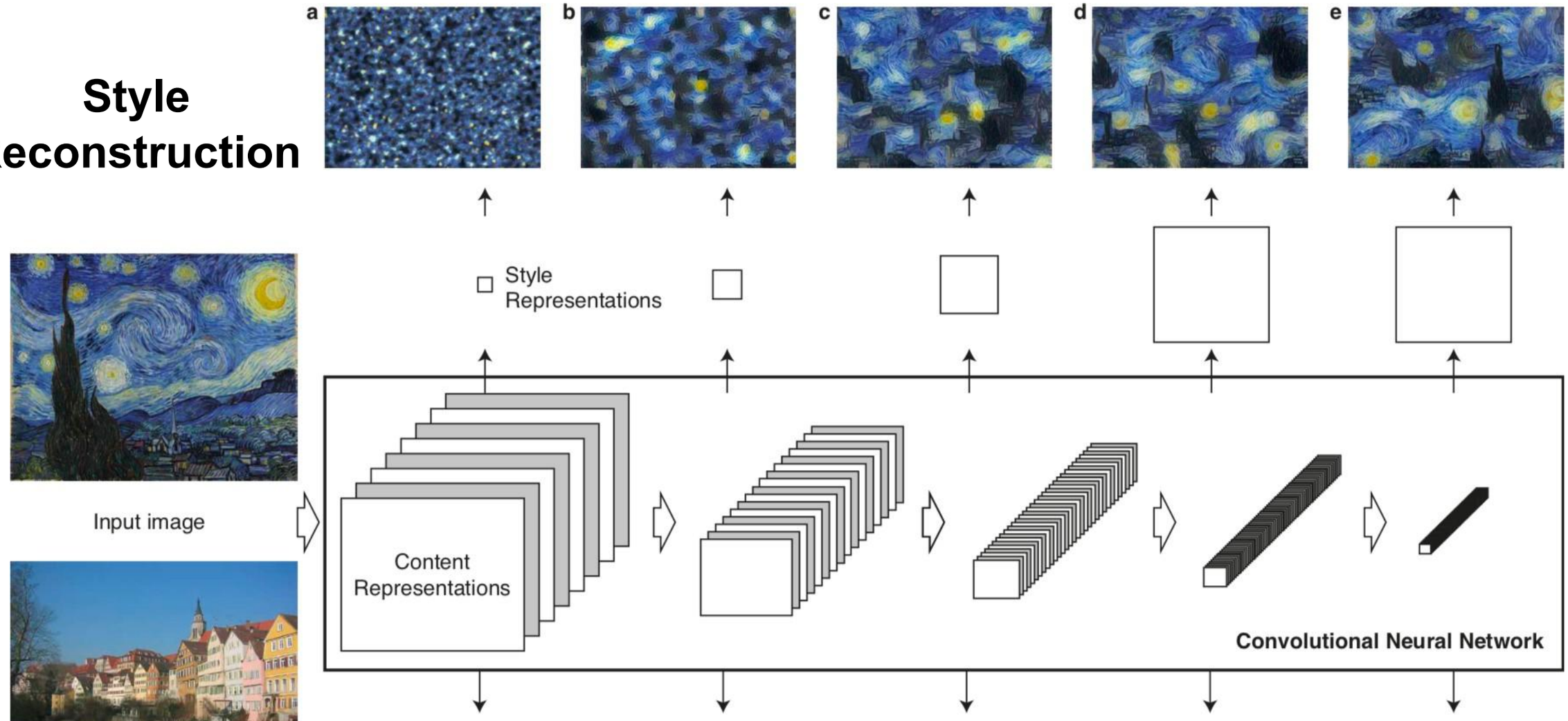
  Content features of the content image is calculated by feeding the content image into the neural network, and extract the activations of those `content_layers`

- Style representation

  - For style features, we extract the correlation of the features of the style-image layer-wise (gram matrix). By adding up the feature correlations of multiple layers, which corresponding to `style_layers`, we obtain a multi-scale representation of the input image, which captures its texture information instead of the object arrangement in the input image

A Neural Algorithm of Artistic Style

Style Reconstruction



Content Reconstruction

A Neural Algorithm of Artistic Style

- Our goal is to create an output image which is synthesized by finding an image that simultaneously matches the **content** features of the photograph and the **style** features of the respective piece of art

- We can define the loss function as the composition of:
  - The dissimilarity of the content features between the output image and the content image
  - The dissimilarity of the style features between the output image and the style image

A Neural Algorithm of Artistic Style

- Content loss
  - Minimize squared-error loss between two feature representations, where F and P are the representations of original and the generated image respectively

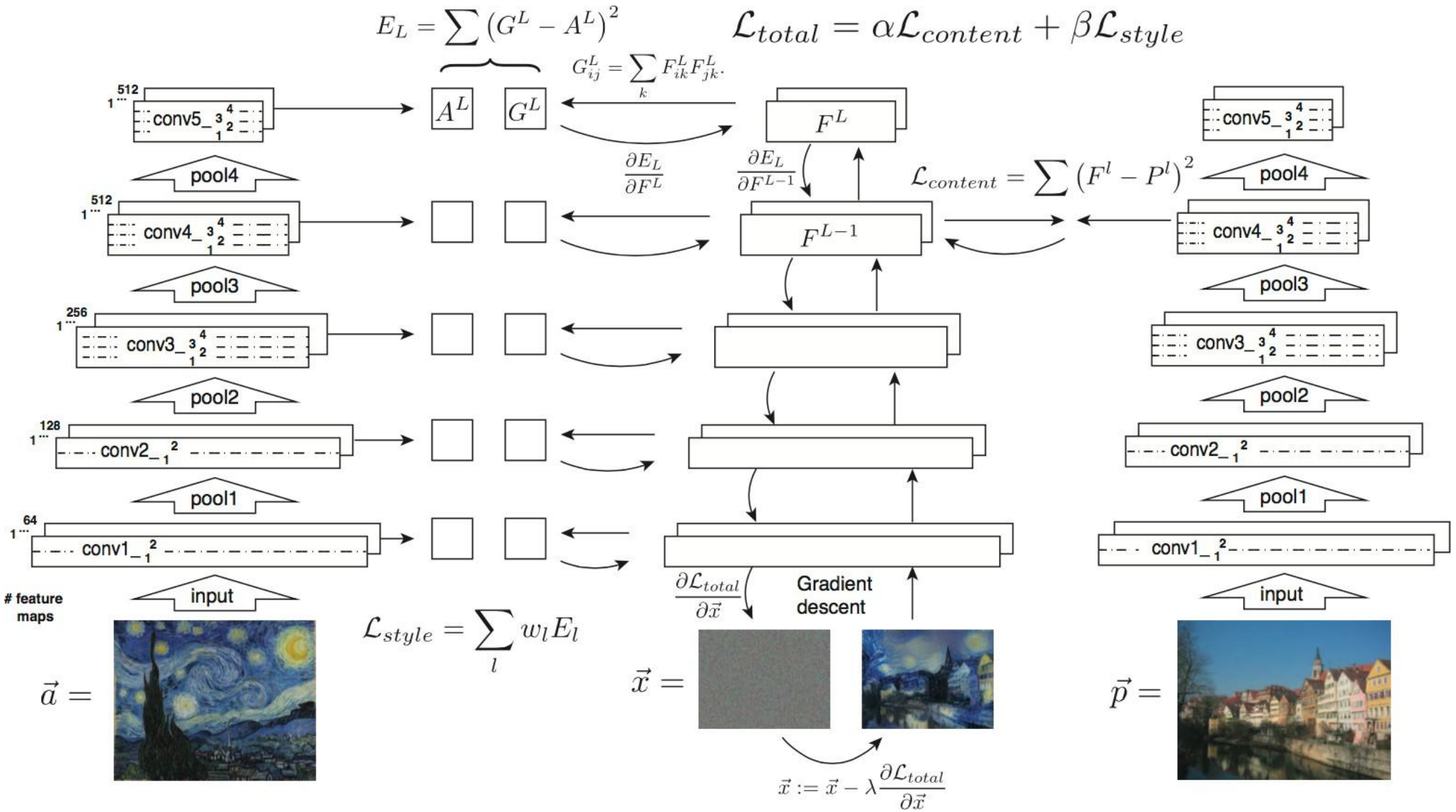
$$\mathcal{L}_{\text{content}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^{l} - P_{ij}^{l}\right)^2$$

- Style loss
  - Use Gram matrix to calculate style loss, consisting feature correlations between the different filter responses. Minimize the mean-squared distance between two Gram matrices. A and G represents original and generated image respectively, and w are weighting factors

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l} \qquad E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^{l} - A_{ij}^{l}\right)^2$$

$$\mathcal{L}_{\text{style}}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

A Neural Algorithm of Artistic Style



A Neural Algorithm of Artistic Style

- So far, we have learned how to train a neural network by using gradient descent to update the weights

$$\frac{\partial L}{\partial W}$$

- How to use the loss function we have defined?

- We have to update our input image!

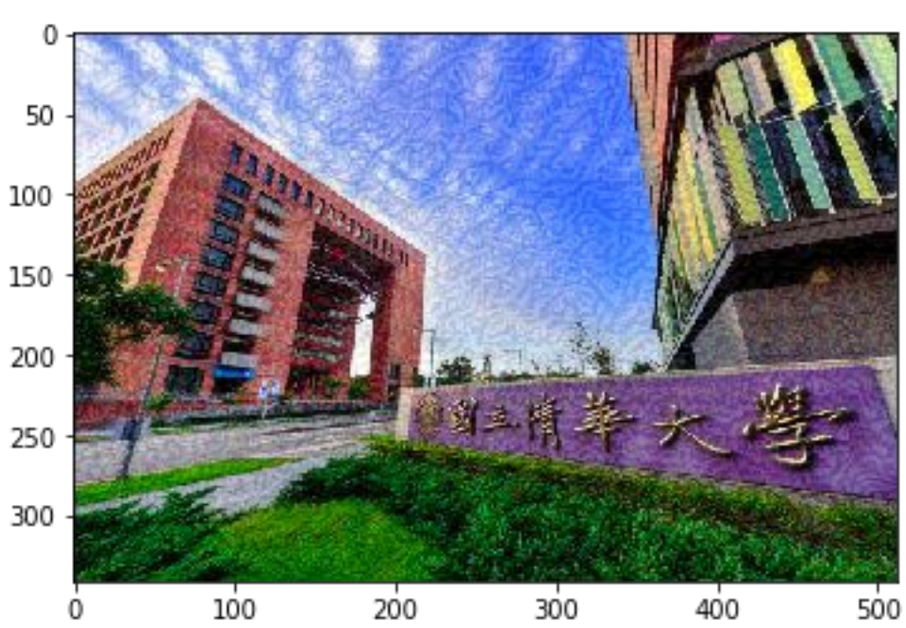
$$\frac{\partial L}{\partial x}$$

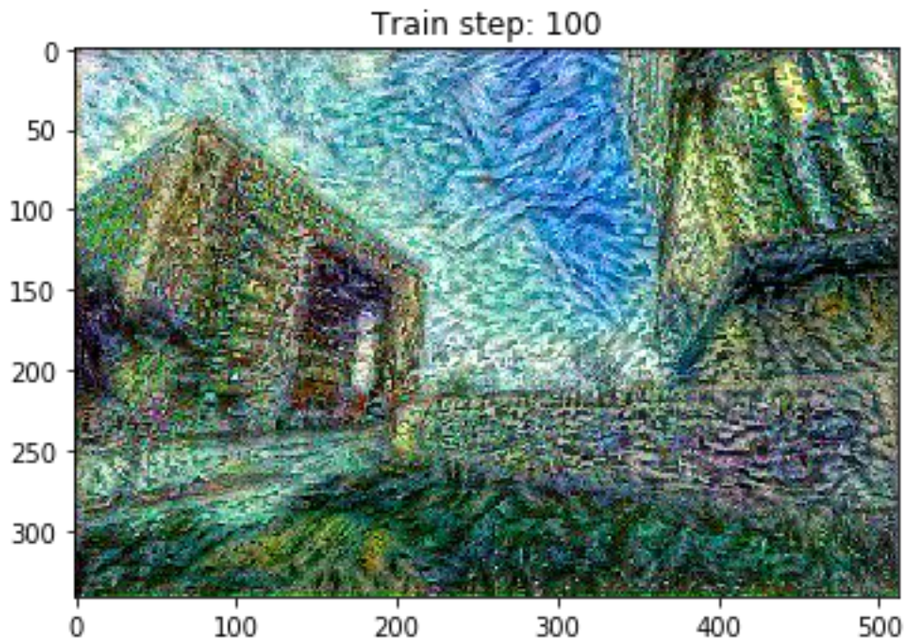
A Neural Algorithm of Artistic Style

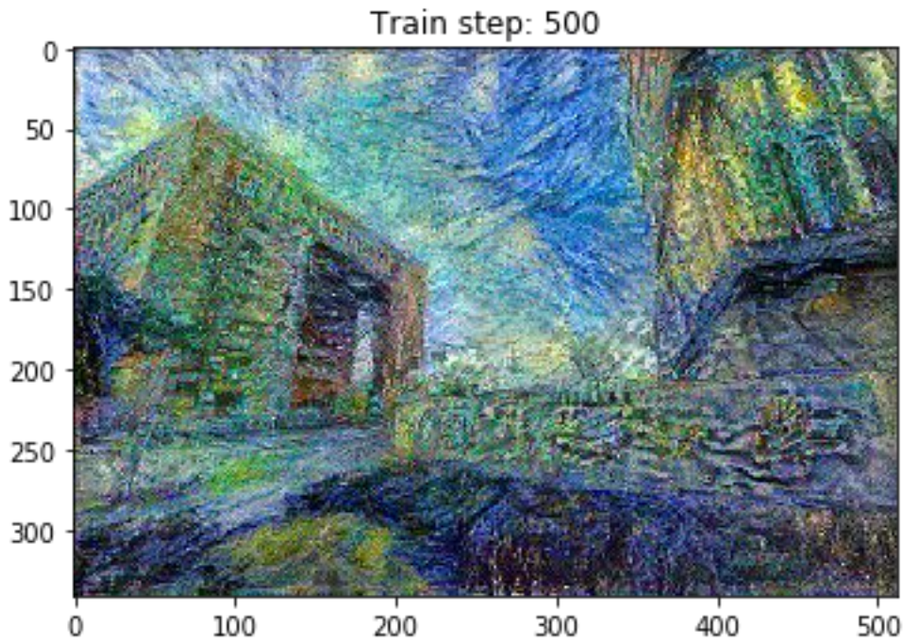
Content Image

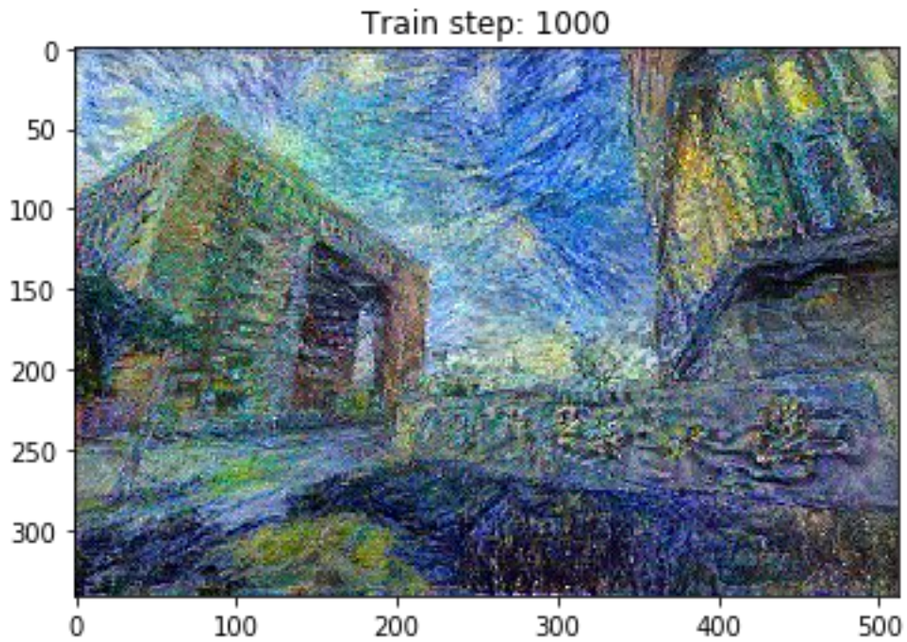
Style Image

A Neural Algorithm of Artistic Style









A Neural Algorithm of Artistic Style

- One downside to this basic implementation is that it produces a lot of high frequency artifacts

- Decrease these using an explicit regularization term on the high frequency components of the image. In style transfer, this is often called the total variation loss

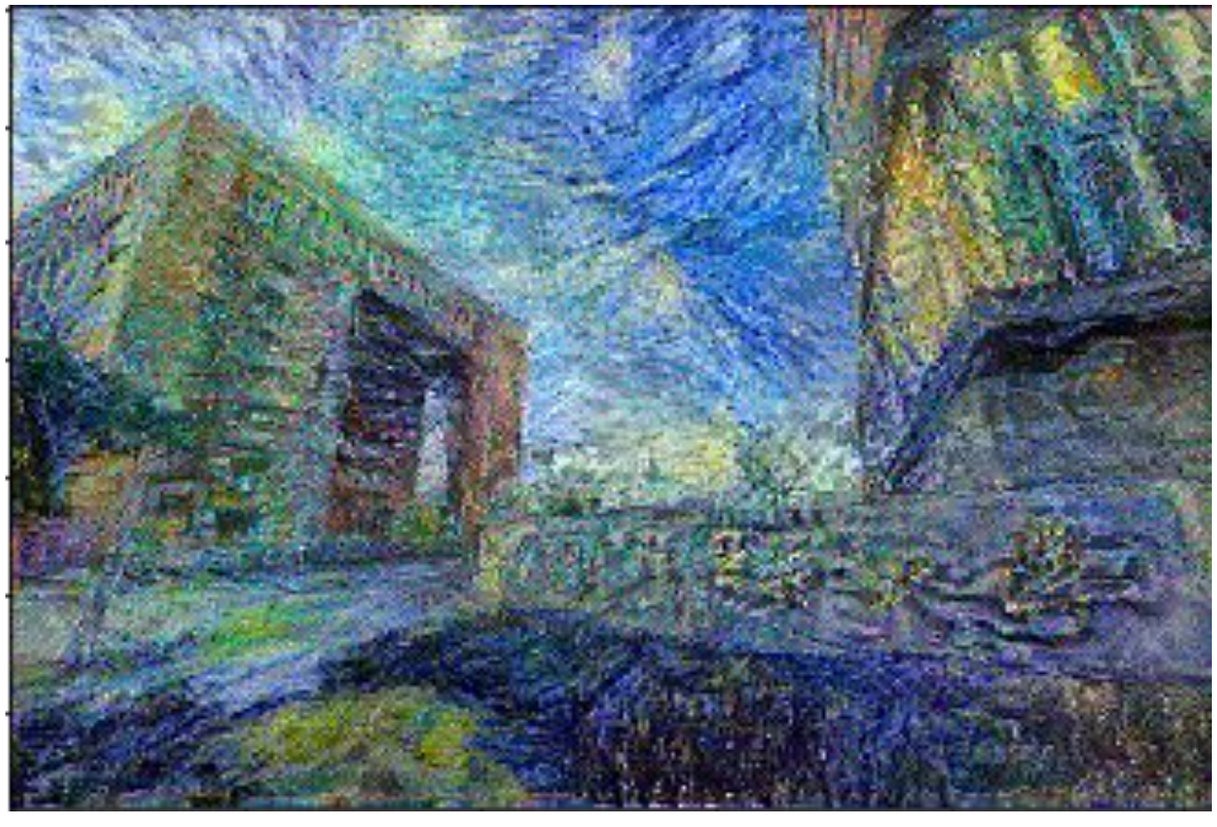
$$V(y) = \sum_{i,j} \sqrt{|y_{i+1,j} - y_{i,j}|^2 - |y_{i,j+1} - y_{i,j}|^2}$$

- In practice, to speed up the computation, we implement the following version instead:

$$V(y) = \sum_{i,j} |y_{i+1,j} - y_{i,j}| + |y_{i,j+1} - y_{i,j}|$$

A Neural Algorithm of Artistic Style

without total variation loss



with total variation loss

Outline

- Visualization

- Neural Style Transfer
  - A Neural Algorithm of Artistic Style

  - AdaIN (Adaptive Instance Normalization)

- Save and Load Models

AdaIN (Adaptive Instance Normalization)

- The method we mentioned above requires a slow iterative optimization process, which limits its pratical application

- Xun Huang and Serge Belongie from Cornell University propose another framework, which enables arbitrary style tranfer in real-time, known as "Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization"

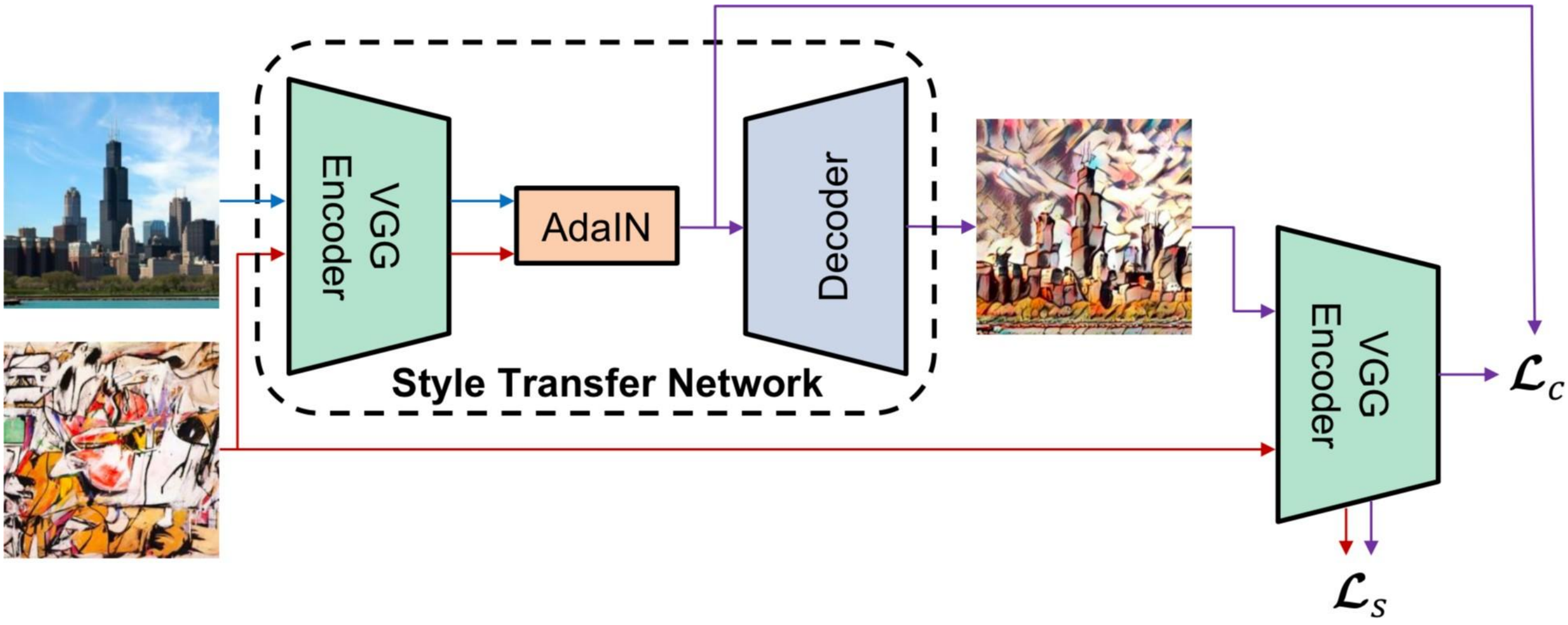
  - AdaIN can transfer arbitrary new styles in real-time, combining the flexibility of the optimization-based framework and the speed similar to the fastest feed-forward approaches

AdaIN (Adaptive Instance Normalization)

- At the heart of this method is a novel Adaptive Instance Normalization (AdaIN) layer aligning the mean and variance of the content features with those of the style features

- Instance normalization performs style normalization by normalizing feature statistics, which have been found to carry the style information of an image in the earlier works

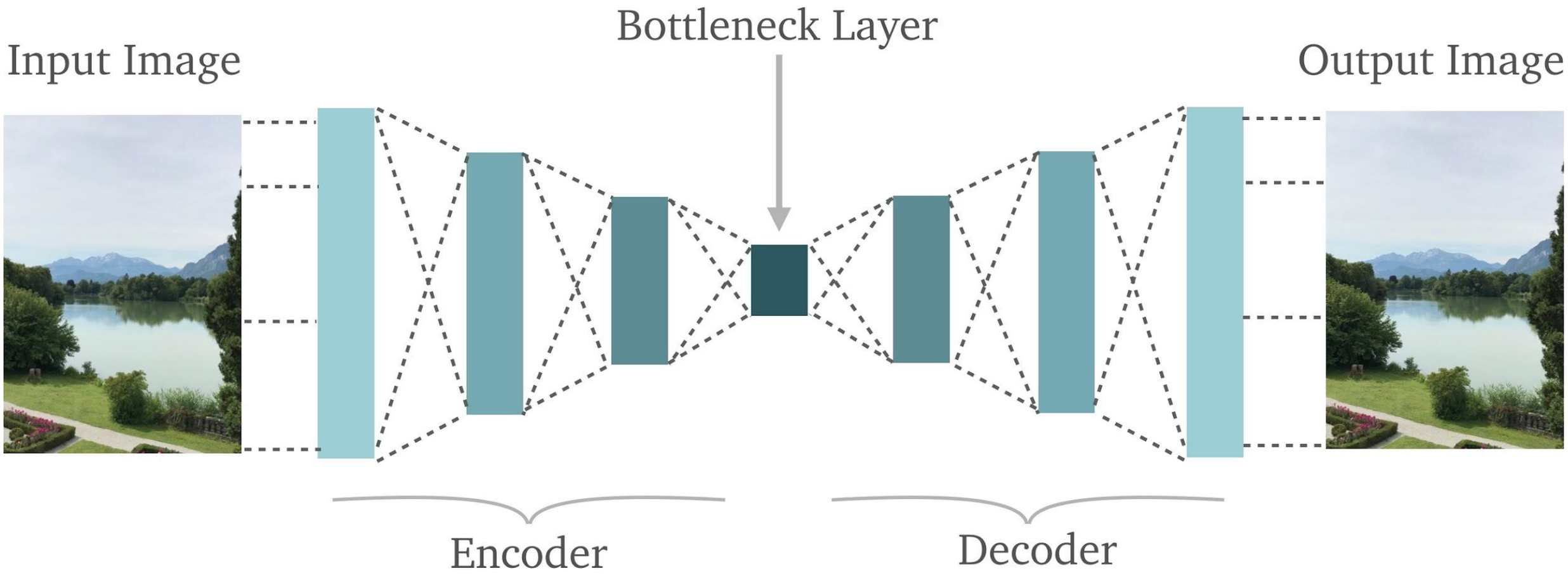
AdaIN (Adaptive Instance Normalization)



AdaIN (Adaptive Instance Normalization)

- What are encoder and decoder?

- Take autoencoder as an example:

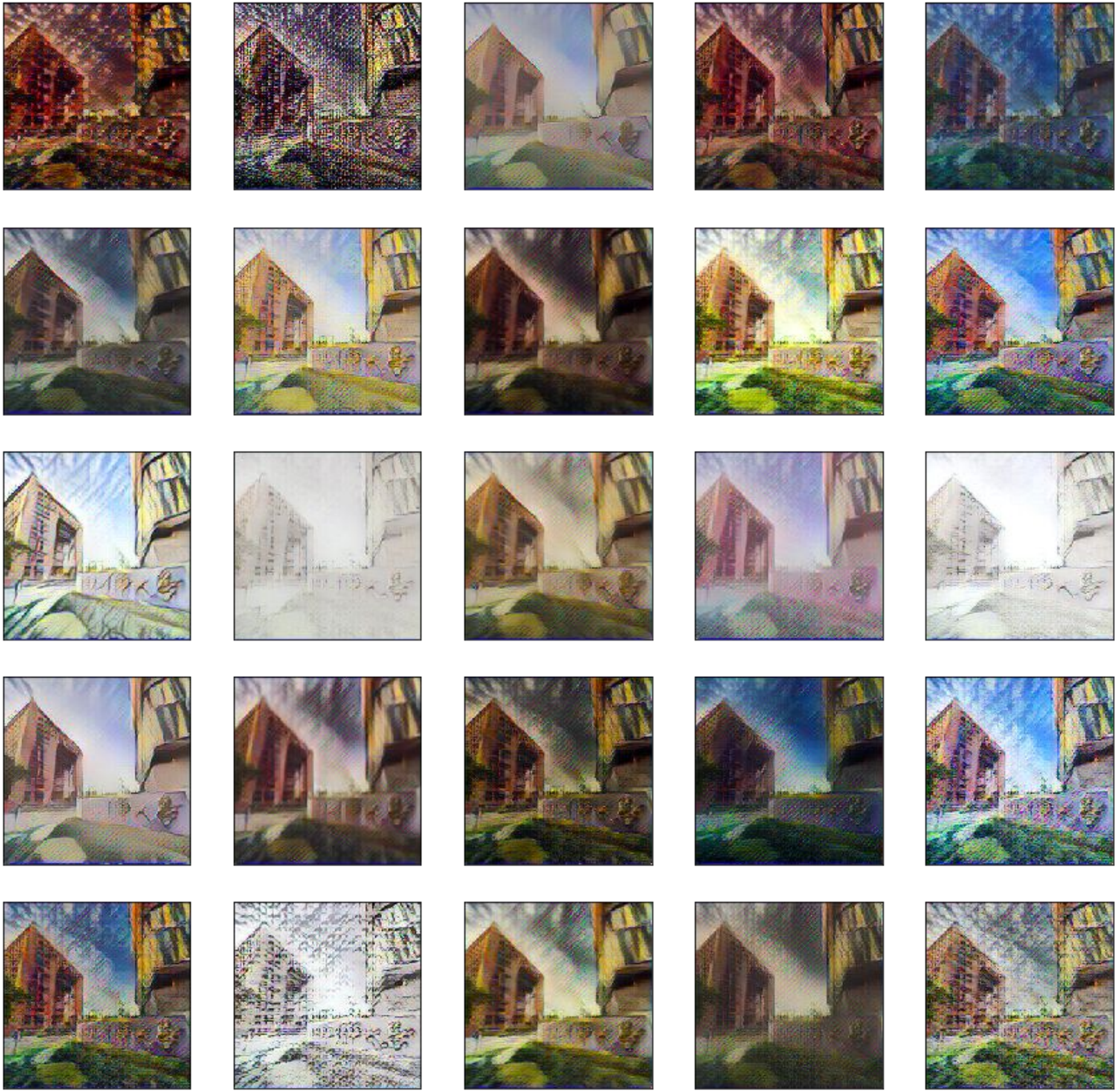


AdaIN (Adaptive Instance Normalization)

- AdaIN receives a content input and style input , and simply aligns the **channel-wise mean** and **variance** of to match those of

- It is worth knowing that unlike BN(Batch Normalization), IN(Instance Normalization) or CIN(Conditional Instance Normalization), AdaIN has no learnable affine parameters. Instead, it adaptively computes the affine parameters from the style input:

$$\text{AdaIN}(x, y) = \sigma(y)\left(\frac{x - \mu(x)}{\sigma(x)}\right) + \mu(y)$$



Be Aware of Pretrained Model

- Pretrained model is quite useful for many reasons.

  However, before diving into pretrained model, we should understand:
  - Input and output
  - Data preprocessing

```
def preprocess_image(path, init_shape=(448, 448)):
    image = tf.io.read_file(path)
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, init_shape)
    image = tf.image.random_crop(image, size=IMG_SHAPE)
    image = tf.cast(image, tf.float32)

    # Convert image from RGB to BGR, then zero-center each color channel with
    # respect to the ImageNet dataset, without scaling.
    image = image[..., ::-1] # RGB to BGR
    image -= (103.939, 116.779, 123.68) # BGR means
    return image
```

Outline

- Visualization
- Neural Style Transfer
  - A Neural Algorithm of Artistic Style
  - AdaIN (Adaptive Instance Normalization)
- Save and Load Models

Save and Load Models

- Model progress can be saved during and after training.

  This means a model can resume where it left off and avoid long training times. Saving also means you can share your model and others can recreate your work

- "Saving a TensorFlow model" typically means one of two things:
  - Checkpoints, or
  - SavedModel

Checkpoints

- Checkpoints capture the exact value of **all parameters** (`tf.Variable` objects) used by a model

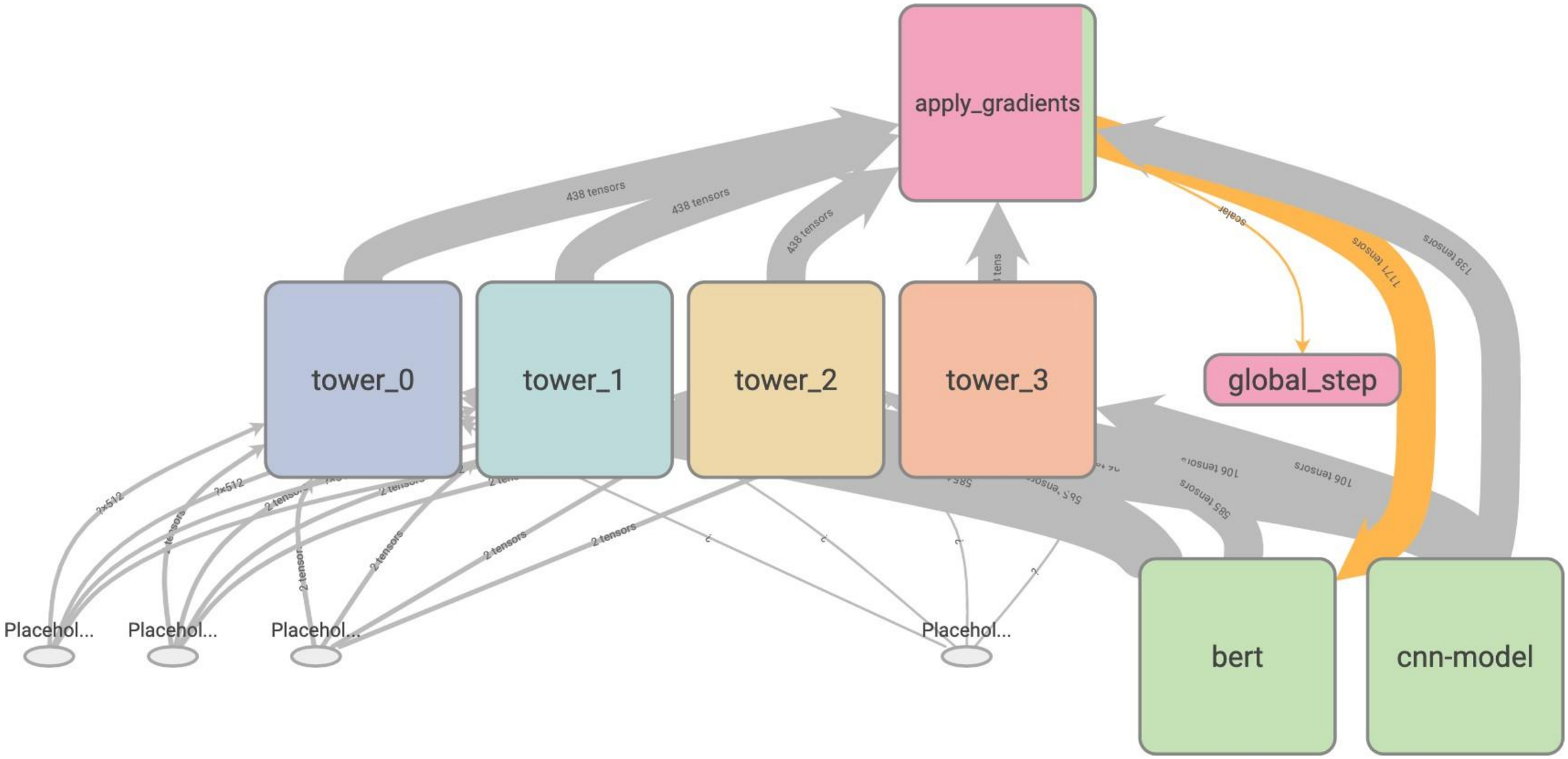
- Checkpoints do not contain any description of the computation defined by the model and thus are typically only useful when source code that will use the saved parameter values is available

SavedModel

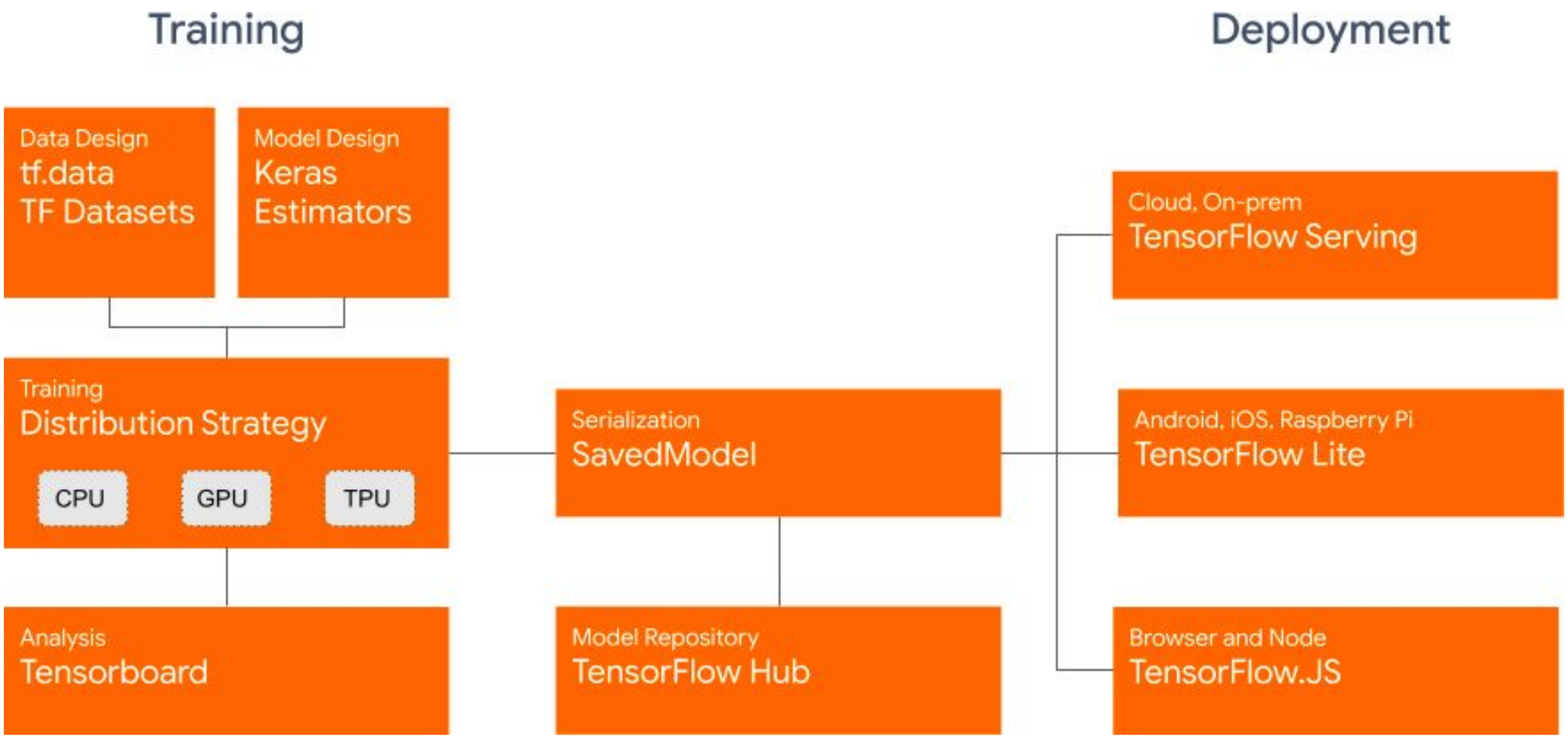
- The SavedModel format on the other hand includes a serialized **description of the computation** defined by the model in addition to the **parameter values** (checkpoint)

- Models in this format are independent of the source code that created the model. They are thus suitable for deployment via TensorFlow Serving, TensorFlow Lite, TensorFlow.js, or programs in other programming languages (the C, C++, Java, Go, Rust, C# etc. TensorFlow APIs)

SavedModel

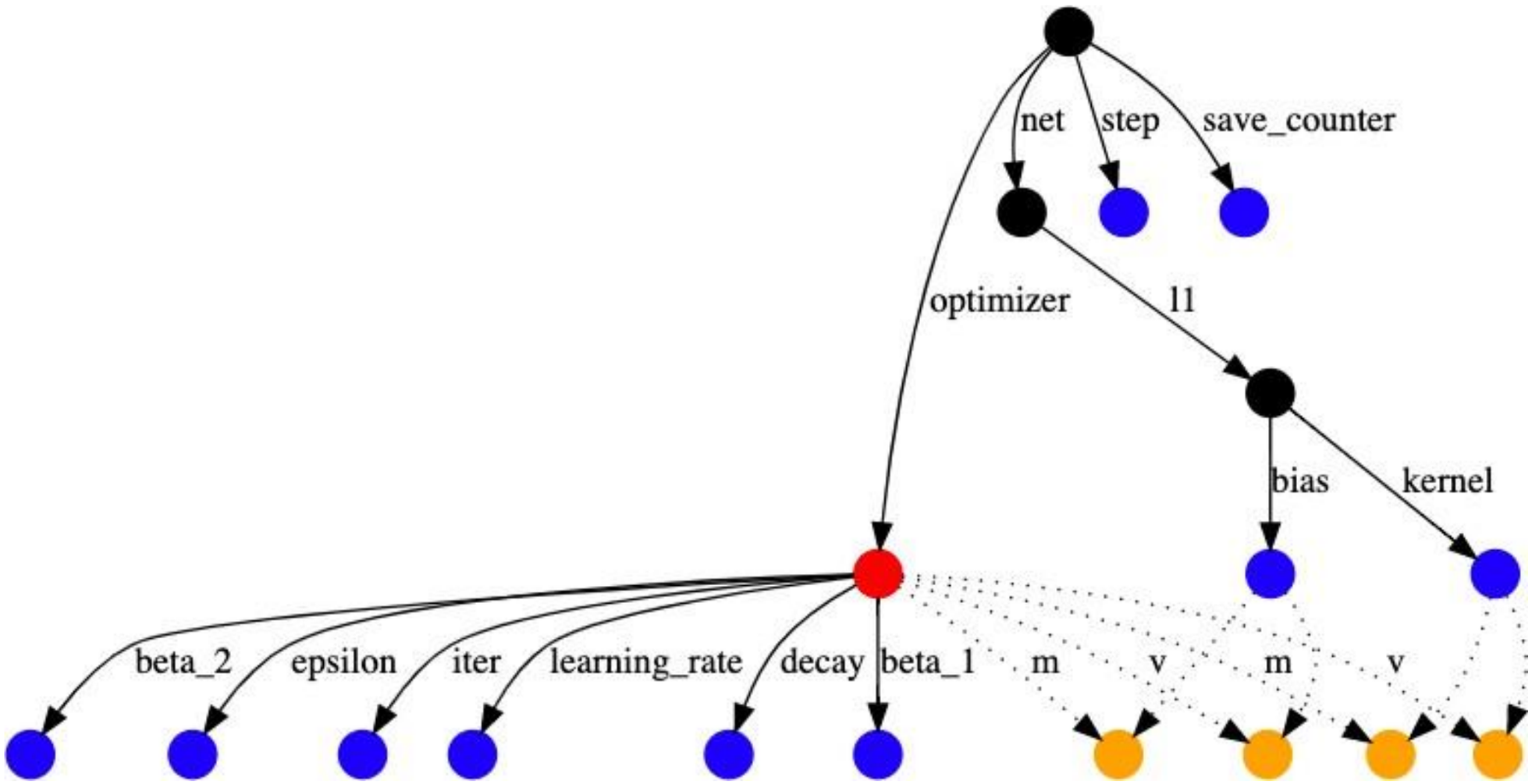

apply_gradients
438 tensors
438 tensors
438 tensors
scalar
1171 tensors
138 tensors
tower_0
tower_1
tower_2
tower_3
global_step
106 tensors
585 tensors
2×512
2 tensors
Placehol...
Placehol...
Placehol...
Placehol...
bert
cnn-model

SavedModel


Training
Data Design
tf.data
TF Datasets
Model Design
Keras
Estimators
Training
Distribution Strategy
CPU
GPU
TPU
Analysis
Tensorboard
Serialization
SavedModel
Model Repository
TensorFlow Hub
Deployment
Cloud, On-prem
TensorFlow Serving
Android, iOS, Raspberry Pi
TensorFlow Lite
Browser and Node
TensorFlow.JS

Inside Checkpoints

- There are various parameters used by the model, including hyperparameters, weights and optimizer slot variables



Checkpoints

- There are several ways to save TensorFlow models, depending on the API you are using. In this section, we are going to demonstrate
  - tf.keras.callbacks.ModelCheckpoint
  - Model.save_weights
  - tf.train.Checkpoints

tf.keras.callbacks.ModelCheckpoint

- The `tf.keras.callbacks.ModelCheckpoint` callback allows to continually save the model both during and at the end of training, and this method **saves all parameters used by a model, including weights and optimizer**

tf.keras.callbacks.ModelCheckpoint

```
EPOCHS = 5

# Checkpoint path and its name
CKP_DIR_SAVE_CALLBACKS = './checkpoints_save_callbacks/ckpt-{epoch}.ckpt'
checkpoint_dir = os.path.dirname(CKP_DIR_SAVE_CALLBACKS)

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Create a callback that saves the model's weights every 1 epochs
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=CKP_DIR_SAVE_CALLBACKS,
    verbose=1,
    save_weights_only=True,
    period=1)

# Train the model with the new callback
model.fit(train_images,
          train_labels,
          epochs=EPOCHS,
          callbacks=[cp_callback],
          validation_data=(test_images,test_labels))
```

tf.keras.callbacks.ModelCheckpoint

Before restore weights

```
# Create a new model instance
model = MyModel()
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
# Re-evaluate the model
loss, acc = model.evaluate(test_images,  test_labels, verbose=2)
print("Restored model, accuracy: {:5.2f}%".format(100*acc))
```

```
1000/1 - 0s - loss: 2.5291 - accuracy: 0.1040
Restored model, accuracy: 10.40%
```

After restore weights

```
# Load the previously saved weights
latest = tf.train.latest_checkpoint(checkpoint_dir)
model.load_weights(latest)

# Re-evaluate the model
loss, acc = model.evaluate(test_images,  test_labels, verbose=2)
print("Restored model, accuracy: {:5.2f}%".format(100*acc))
```

```
1000/1 - 0s - loss: 0.5423 - accuracy: 0.8550
Restored model, accuracy: 85.50%
```

Model.save_weights

- Manually saving them is just as simple with the `Model.save_weights` method, and it is quite useful during **custom training**. In our Deep Learning course, most of assignments and competitions are required custom training

- Another thing you should notice is the difference between `tf.keras.callbacks.ModelCheckpoint` and `Model.save_weights`. The former one saves all parameters used in model, including weights and optimizers, while the latter one **only saves weights**. No information about optimizer is saved

Model.save_weights

```
CKP_DIR_SAVE_WEIGHTS = './checkpoints_save_weights'

for epoch in range(EPOCHS):
    for images, labels in train_ds:
        train_step(images, labels)

    for test_images, test_labels in test_ds:
        test_step(test_images, test_labels)

    template = 'Epoch {:0}, Loss: {:.2f}, Accuracy: {:.2f}, Test Loss: {:.2f}, Test Accuracy: {:.2f}'
    print (template.format(epoch+1,
                           train_loss.result(),
                           train_accuracy.result()*100,
                           test_loss.result(),
                           test_accuracy.result()*100))

    # Use Model.save_weights during training
    # You can modify the saving frequency by simply using "if epoch == ?, then save"
    print("Saved checkpoint for step {}: {}".format(int(epoch+1), CKP_DIR_SAVE_WEIGHTS + f'/ckpt-{epoch+1}'))
    model.save_weights(os.path.join(CKP_DIR_SAVE_WEIGHTS, f'ckpt-{epoch}'))

    # Reset the metrics for the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()
```

tf.train.Checkpoint

- Another way to save checkpoint during custom training is to use `tf.train.Checkpoint` API, capturing the exact value of **all parameters used by model**

- Additionally, this API allows you to decide what you want to save exactly

```
CKP_DIR_SAVE_CHECKPOINTS = './checkpoints_save_checkpoints'

# Place the models and optimizers you want to store
# as the arguments of tf.train.Checkpoint
# You can store several different models and optimizers at the same time
ckpt = tf.train.Checkpoint(step=tf.Variable(1), optimizer=optimizer, model=model)
manager = tf.train.CheckpointManager(ckpt, CKP_DIR_SAVE_CHECKPOINTS, max_to_keep=3)
```

Checkpoints

- tf.keras.callbacks.ModelCheckpoint
  - The easiest one, but not so flexible
  - Save weights and optimizer
- Model.save_weights
  - Flexible, which can be used in custom training
  - Save weights
- tf.train.Checkpoints
  - Flexible, which can be used in custom training
  - Save weights and optimizer
  - Able to specify what you want to save exactly

Assignment

- Part I (A Neural Algorithm of Artistic Style)
  - Implement total variational loss. `tf.image.total_variation` is not allowed **(10%)**
  - Change the weights for the style, content, and total variational loss **(10%)**
  - Use other layers in the model **(10%)**
  - Write a brief report. Explain how the results are affected when you change the weights, use different layers for calculating loss **(10%)**
- Part II (AdaIN)
  - Implement AdaIN layer and use single content image to create 25 images with different styles **(60%)**

Assignment

- Requirements:
  - Submit to iLMS, including your code file `Lab12-2_{student id}.ipynb` and result images
  - **Honest code**. Students will get 0 if plagiarism is found
  - Deadline: **2021-11-18(Thur) 23:59**

Reference

- VGG 19 model

- The code of style transfer is based on Tersorflow official tutorial, while the code of visualization is based on How to Visualize Filters and Feature Maps in Convolutional Neural Networks by Jason Brownlee

- Original work of Style Transfer's TensorFlow implementation is from Anish Athalye's GitHub account anishathalye

- Guided-backpropagation: Striving for Simplicity: The All Convolutional Net J. T. Springenberg and A. Dosovitskiy et al., ICLR'15 Workshop

- Style transfer: A Neural Algorithm of Artistic Style, Gatys et al., arXiv'15

- AdaIN: "Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization", Huang et al., ICLR'17 Workshop / ICCV'17

- The tutorial of "Save and Load Model" is based on Tensorflow official tutorial