Looking inside our models

Visualizing deep neural networks + BatchNorm

Sten-Oliver Salumaa & Joonas Kriisk
Intermediate activations are “useful for understanding how successive convnet layers transform their input, and for getting a first idea of the meaning of individual convnet filters.”

- François Chollet

Feature map visualization in tf/keras

https://github.com/gabrielpierobon/cnnshapes/
Feature map visualisation in tf/keras

https://github.com/gabrielpierobon/cnnshapes/
Feature map visualization in tf/keras

Create a callback for saving intermediate layers’ outputs:
- ReLu / BatchNorm / Conv / whatever

Needs (in tf/keras):
- Defining your layer output wishlist
- Forward pass (predict in keras)

Now all activations are in your model output

https://github.com/gabrielpierobon/cnnshapes/
Feature map visualization in tf/keras

input

after 1st conv

bottom of ‘U’

near output

https://github.com/gabrielpierobon/cnnshapes/
Visualizing convnet feature importance heatmap

```
In [ ]: from fastai.callbacks.hooks import *

In [ ]:
def hooked_backward(cat=y):
    with hook_output(m[0]) as hook_a:
        with hook_output(m[0], grad=True) as hook_g:
            preds = m(xb)
            preds[0, int(cat)].backward()
        return hook_a, hook_g

In [ ]: hook_a, hook_g = hooked_backward()

In [ ]:
acts = hook_a.stored[0].cpu()
acts.shape

Out[ ]: torch.Size([512, 11, 11])

In [ ]:
avgActs = acts.mean(0)
avgActs.shape

Out[ ]: torch.Size([11, 11])

In [ ]:
def show_heatmap(hm):
    _, ax = plt.subplots()
    xb_im.show(ax)
    ax.imshow(hm, alpha=0.5, extent=(0,352,352,0), interpolation='bilinear', cmap='magma');

https://github.com/fastai/course-v3/blob/master/nbs/dl1/lesson6-pets-more.ipynb
```
Filter activation maximization

1. Generate a random uniform input image
2. Register feature-saving hook to layer of interest (to cache activations)
3. Create an optimizer with that can only change the input image
4. Create a training loop which at each step:
   a. Makes a forward pass with the image
   b. Calculates loss (loss is \(-1*\text{filter\_of\_interest}.\text{mean()}\))
   c. Calculates gradients
   d. Takes a step

Filter activation maximization

https://towardsdatascience.com/how-to-visualize-convolutional-features-in-40-lines-of-code-70b7d87b0030
1. Do a forward pass of the image through the network
2. Calculate the scores for every class
3. Enforce derivative of score $S$ at last layer for all classes except class $C$ to be 0. For $C$, set it to 1
4. Backpropagate this derivative till the start
5. Now you have saliency maps

Visualizing training process

https://www.youtube.com/watch?v=TtFAXYQc54o
Tensorspace.js

Normalization

● Normalizing (and standardizing)
  ○ (centering) subtract its average, then
  ○ (scaling) divide by its standard deviation
  ○ Results in mean of 0 and standard deviation of 1

● Speeds up training and leads to faster convergence
  ○ But why?

● Ill-conditioned network
  ○ Small change in the independent variable (input) leads to a large change in the dependent variable (output)
Ill-conditioned

- Can be result of training data, network’s architecture or the initial weights
- Gradient descent requires a good learning rate
- Learning rates for different weights differ by a lot
- Results in long, narrow error function valleys
- Mathematically characterized by a certain number, but calculating it is a rather complicated procedure
- In short, either error too high or too low
Batch Normalization

- If it works for the input, why not use it for hidden layers as well?
  - Works really well!
  - Can use higher learning rates
  - Slight regularization
- Normalizes the output of a previous activation layer
  - Two trainable parameters - $\gamma$, $\beta$
- In general - preprocessing at every layer which speeds up and stabilizes

\[\begin{align*}
\text{Input: } & \text{ Values of } x \text{ over a mini-batch: } B = \{x_1, \ldots, x_m\}; \\
\text{Parameters to be learned: } & \gamma, \beta \\
\text{Output: } & \{y_i = \text{BN}_{\gamma, \beta}(x_i)\}
\end{align*}\]

\[
\begin{align*}
\mu_B & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean} \\
\sigma_B^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad \text{// scale and shift}
\end{align*}\]

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.
Batch Normalization

- Can be used before or after activation (depending on activation function)
- Less sensitive to weight initialization
- Outputs smoother error surface
- Used a lot in research which reaches (at that time) state-of-the-art results
- Research being done to understand why it actually works (internal covariate shift vs high-order effects)
References

2. https://blog.yani.io/sgd/
3. https://cnl.salk.edu/~schraudo/teach/NNcourse/precond.html
6. https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/?fbclid=IwAR0CWNodfMjhCtcUi9oDL7KBPoZ4A4NfVRmJQJiNyErcXwsoFB9Y_1iGs