MORE ABOUT ARCHITECTURES

Do this for all shifts.

Still Sharing Computation!
MORE ABOUT ARCHITECTURES

- We need downsampling because that quickly increases receptive field.
- But, we need to get an output for every pixel. So we apply the network on all overlapping receptive fields efficiently using dilated convolutions.

As it suggests, it is the transpose of the operation of Convolution with Stride.

In fact, this represents the operation for back-propagating gradients through a convolution-with-stride layer.

Lets go back to our matrix vector notation, represent convolution with $A_k$ and downsampling with $A_s$.

What is the transpose of this operation? Of $A_s A_k$?

$$y = A_s A_k x$$

What does $A_s^T$ represent?

Upsampling by filling in zeros. $A_s^T$ is still convolution (with a flipped kernel, but doesn’t matter).

So a convolution-transpose layer effectively does up-sampling with zeros, and then a regular convolution.

But up-sampling with zeros often leads to artifacts. Newer architectures don’t use convolution transpose. Instead, they do bilinear or nearest-neighbor interpolation on the feature maps to increase resolution, and then do a regular convolution.

MORE ABOUT ARCHITECTURES

Newell et al., Hour-glass Networks.

Another popular architecture: U-Net.
TRAINING IN PRACTICE

- Remember: Gradient Descent is Fragile

The Effect of Parameterization

\[ f(x; \theta) = \theta x \]
\[ f' (x; \theta') = 2\theta' x \]

- Two representations of the same hypothesis space.
- Let’s initialize \( \theta' = \theta/2 \). Say \( \theta = 10, \theta' = 5 \).
- Compute loss with respect to the same example.
  \[ \text{Say } \nabla f/L = \nabla f' = 1, x = 1 \]
  - What are \( \nabla \theta \) and \( \nabla \theta' \) ?
  \[ \nabla \theta = 1, \nabla \theta' = 2. \]
- Update with learning rate = 1
  - Updated \( \theta = 9, \theta' = 3 \)
  - \( f(x) = 9x, f'(x) = 6x \)

Initialization

- Initialize all biases to 0. Why? Don’t want to shift the mean.
- Now initialize weights randomly so that variance of outputs = variance of inputs = 1.
  - Use approximation \( \text{var}(wx) = \text{var}(w)\text{var}(x) \) for scalar \( w \) and \( x \).
  - Say you have a fully connection layer: \( y = W^T x \)
    - \( x \) is \( N \)-dimensional. We assume its 0-mean unit-variance coming in.
    - \( W \) is \( N \times M \) dimensional.
    - We will initialize \( W \) with 0-mean and variance \( \sigma^2 \).
    - What should be the value of \( \sigma^2 \)? Take 5 mins.
    - \( \sigma^2 = 1/N. \)
  - Now what about a convolution layer with kernel size \( K \times K \times C_1 \times C_2 \).
    - Initialize kernel with 0-mean and variance \( \sigma^2 \). What should \( \sigma^2 \) be?
    - \( \sigma^2 = 1/(K^2 C_1) \)

TRAINING IN PRACTICE

- Actually, using normal distributions is sometimes unstable.
- Probability for values very far from the mean is low, but not 0.
- When you sample millions of weights, you might end up with such a high value!
- Solution: Use truncated distributions that are forced to have 0 probability outside a range.
  - Uniform distribution, “truncated-normal”
  - Figure out what parameters of this distribution should be to have equivalent variance.

TRAINING IN PRACTICE

- Because we’re using a first order method, it is important to make sure that the activations of all layers are the same magnitude.
- Because we have RELUs \( y = \max(0, x) \), it is important to make sure roughly half the expectations are positive.
- Normalize your inputs to be 0-mean and unit variance.
  - Compute dataset mean and standard deviation, subtract and divide from all inputs.
  - For images, you usually compute the mean over all pixels—so single normalization for all pixels in the input.
    - But different for different color channels.
  - Sometimes, you will want ‘per-location’ normalization.
  - Other times, you will normalize each input by its own pixel mean and standard deviation.
TRAINING IN PRACTICE

Initialization

- But this only ensures zero-mean unit-variance at initialization.
- As your weights update, they can begin to give you biased weights.
- Another option, add normalization in the network itself!


BATCH-NORMALIZATION

- Batch-Norm Layer

\[
 y = BN(x) = \frac{x - \text{Mean}(x)}{\sqrt{\text{Var}(x)} + \epsilon}
\]

Here, mean and variance are interpreted per-channel.

So for each channel, you compute mean of the values of that channel activation at all spatial locations across all examples in the training set.

But this would be too hard to do in each iteration. So the BN layer just does this normalization over a batch.

And back-propagates through it.

Typically apply BN before a RELU.

Typical use: RELU(BN(Conv(x,k))+b)

- Don’t add bias before BN as it’s pointless
- Learn bias post-BN
- Can also learn a scale: RELU(a BN(Conv(x,k))+b)

Leads to significantly faster training.

But you need to make sure you are normalizing over a diverse enough set of samples.

Batch-Norm Layer

\[
 x = (x)_{b,h,c} \\
 \mu_c = \frac{1}{BH \cdot W} \sum_{b,h} x_{b,h,c} \\
 \sigma^2_c = \frac{1}{BH \cdot W} \sum_{b,h} (x_{b,h,c} - \mu_c)^2 \\
 y_{b,h,c} = \frac{x_{b,h,c} - \mu_c}{\sqrt{\sigma^2_c + \epsilon}}
\]

Typically apply BN before a RELU.

Typical use: RELU(BN(Conv(x,k))+b)

- Don’t add bias before BN as it’s pointless
- Learn bias post-BN
- Can also learn a scale: RELU(a BN(Conv(x,k))+b)

Leads to significantly faster training.

But you need to make sure you are normalizing over a diverse enough set of samples.

The BN layer has no parameters.

What about during back-propagation?

- When you back-propagate through it to \( x \), you go back-propagate through the computation of mean and variance.
- At test time, replace \( \mu_c \) and \( \sigma^2_c \) as mean and variance over the entire training set.
Given a limited amount of training data, deep architectures will begin to overfit.

**Important:** Keep track of training and dev-set errors

Training errors will keep going down, but dev will saturate. Make sure you don’t train to a point when dev errors start going up.

So how do we prevent, or delay, overfitting so that our dev performance increases?

Solution 1: Get more data.

Data Augmentation

- Think of transforms to the images that you have that would still keep them in the distribution of real images.
- Typical Transforms
  - Scaling the image
  - Taking random crops
  - Applying Color-transformations (change brightness, hue, saturation randomly)
  - Horizontal Flips (but not vertical)
  - Rotations upto ± 5 degrees.
- Are a good way of getting more training data for ‘free’.
- Teaches your network to be invariant to these transformations ....
- Unless your output isn’t. If your output is a bounding box, segmentation map, or other quantities that would change with these augmentation operations, you need to apply them to the outputs too.

Weight Decay

- Add a squared or absolute value penalty on all weight values (for example, on each element of every convolutional kernel, matmul matrix) except biases. $\sum_i w_i^2$ or $\sum_i |w_i|$
- So now your effective loss is $L' = L + \lambda \sum_i w_i^2$
- How would you train for this?
  - Let’s say you use backprop to compute $\nabla w, L$
  - What gradient would you apply to your weights? What is $\nabla_w L'$?
    $$\nabla L' = \nabla L + 2\lambda w_i$$
- So in addition to the standard update, you will also be subtracting a scaled version of the weight itself.
- What about for $L' = L + \lambda \sum_i |w_i|$?
  $$\nabla L' = \nabla L + \lambda \text{sign}(w_i)$$