caffe.berkeleyvision.org

Caffe | Layer Catalogue

- View On GitHub

To create a Caffe model you need to define the model architecture in a protocol buffer definition file (prototxt).

Caffe layers and their parameters are defined in the protocol buffer definitions for the project in caffe.proto.

Data Layers

Data enters Caffe through data layers: they lie at the bottom of nets. Data can come from efficient databases (LevelDB or LMDB), directly from memory, or, when efficiency is not critical, from files on disk in HDF5 or common image formats.

Common input preprocessing (mean subtraction, scaling, random cropping, and mirroring) is available by specifying `TransformationParameter` s by some of the layers. The bias, scale, and crop layers can be helpful with transforming the inputs, when `TransformationParameter` isn't available.

Layers:

- Image Data - read raw images.
- Database - read data from LEVELDB or LMDB.
- HDF5 Input - read HDF5 data, allows data of arbitrary dimensions.
- HDF5 Output - write data as HDF5.

- Input - typically used for networks that are being deployed.
- Window Data - read window data file.
- Memory Data - read data directly from memory.
- Dummy Data - for static data and debugging.

Note that the Python Layer can be useful for create custom data layers.

Vision Layers

Vision layers usually take *images* as input and produce other *images* as output, although they can take data of other types and dimensions. A typical “image” in the real-world may have one color channel (c=1$c = 1$), as in a grayscale image, or three color channels (c=3 $c = 3$) as in an RGB (red, green, blue) image. But in this context, the distinguishing characteristic of an image is its spatial structure: usually an image has some non-trivial height h>1$h > 1$ and width w>1$w > 1$. This 2D geometry naturally lends itself to certain decisions about how to process the input. In particular, most of the vision layers work by applying a particular operation to some region of the input to produce a corresponding region of the output. In contrast, other layers (with few exceptions) ignore the spatial structure of the input, effectively treating it as “one big vector” with dimension chwchw.

Layers:

- Convolution Layer - convolves the input image with a set of learnable filters, each producing one feature map in the output image.
- Pooling Layer - max, average, or stochastic pooling.
- Spatial Pyramid Pooling (SPP)
- Crop - perform cropping transformation.

- Deconvolution Layer - transposed convolution.

- Im2Col - relic helper layer that is not used much anymore.

Recurrent Layers

Layers:

- Recurrent

- RNN

- Long-Short Term Memory (LSTM)

Common Layers

Layers:

- Inner Product - fully connected layer.

- Dropout

- Embed - for learning embeddings of one-hot encoded vector (takes index as input).

Normalization Layers

- Local Response Normalization (LRN) - performs a kind of “lateral inhibition” by normalizing over local input regions.

- Mean Variance Normalization (MVN) - performs contrast normalization / instance normalization.

- Batch Normalization - performs normalization over mini-batches.

The bias and scale layers can be helpful in combination with normalization.

Activation / Neuron Layers

In general, activation / Neuron layers are element-wise operators, taking one bottom blob and producing one top blob of the same size. In the layers below, we will ignore the input and out sizes as they are identical:

- Input
  - n * c * h * w
- Output
  - n * c * h * w

Layers:

- ReLU / Rectified-Linear and Leaky-ReLU - ReLU and Leaky-ReLU rectification.
- PReLU - parametric ReLU.
- ELU - exponential linear rectification.
- Sigmoid
- TanH
- Absolute Value
- Power - f(x) = (shift + scale * x) ^ power.
- Exp - f(x) = base ^ (shift + scale * x).

- Log - f(x) = log(x).
- BNLL - f(x) = log(1 + exp(x)).
- Threshold - performs step function at user defined threshold.
- Bias - adds a bias to a blob that can either be learned or fixed.
- Scale - scales a blob by an amount that can either be learned or fixed.

Utility Layers

Layers:

- Flatten
- Reshape
- Batch Reindex
- Split
- Concat
- Slicing
- Eltwise - element-wise operations such as product or sum between two blobs.
- Filter / Mask - mask or select output using last blob.
- Parameter - enable parameters to be shared between layers.

- Reduction - reduce input blob to scalar blob using operations such as sum or mean.
- Silence - prevent top-level blobs from being printed during training.
- ArgMax
- Softmax
- Python - allows custom Python layers.

Loss Layers

Loss drives learning by comparing an output to a target and assigning cost to minimize. The loss itself is computed by the forward pass and the gradient w.r.t. to the loss is computed by the backward pass.

Layers:

- Multinomial Logistic Loss
- Infogain Loss - a generalization of MultinomialLogisticLossLayer.
- Softmax with Loss - computes the multinomial logistic loss of the softmax of its inputs. It’s conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.
- Sum-of-Squares / Euclidean - computes the sum of squares of differences of its two inputs, 12N∑Ni=1‖x1i−x2i‖22$\frac{1}{2N}\sum_{i=1}^{N} \| x_i^1 - x_i^2\|_2^2$.
- Hinge / Margin - The hinge loss layer computes a one-vs-all hinge (L1) or squared hinge loss (L2).

- Sigmoid Cross-Entropy Loss - computes the cross-entropy (logistic) loss, often used for predicting targets interpreted as probabilities.
- Accuracy / Top-k layer - scores the output as an accuracy with respect to target – it is not actually a loss and has no backward step.
- Contrastive Loss