ConvNetQuake

Methods

Input: 2D tensor

- rows: 3 channels of the waveform
- columns: time index (1000 elements long)
    - because 10-s long windows sampled at 100 Hz

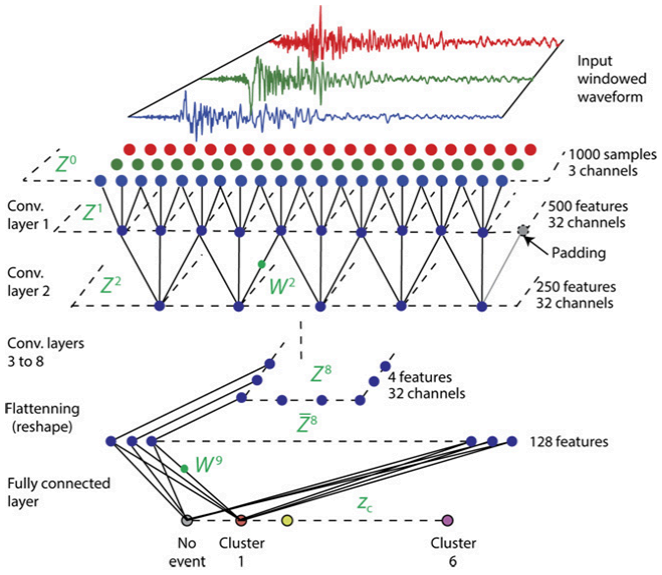
Output: M-dimensional vector

- where M is the number of classes - in the example case, M=7
    - class 0 is the noise
    - classes 1 to M-1 correspond to the geographic clusters
- each element of this vector is the probability that the input belongs to that geographic region

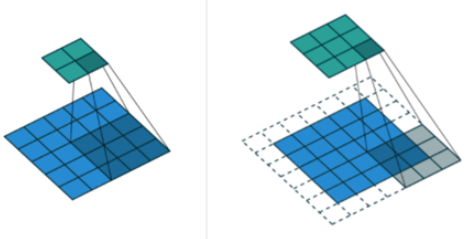
Network Architecture

8 convolutional layers Z1-Z8

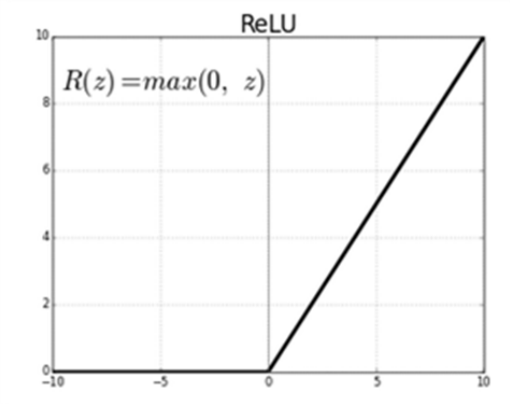
- 32 filters
    - controls the dimension of the output tensor
- kernel size = 3
    - this is the size of the “sliding window”



- padding
    - blue maps are inputs and green are outputs
    - note that this is an example - it’s a 2D conv, but ours is 1D
    - refresher: since the sliding window is 3 wide, you would need to pull from



outside of the input region to keep your output the same size
    - "zero" padding just means that the values of the padding is zero
- stride = 2
    - i.e., kernel slides in increments of two samples
    - output size will be cut in half every layer
- activation function: ReLU
    - negative numbers changed to 0 and positive numbers stay the same



Fully connected layer

- first, vectorized (flattened) Z8 into 1D tensor with 128 elements
- next: fully connected layer
    - fully connected vs. convolutional NN:
        - fully connected have large weight tensors (i.e., too many parameters)-> NN is slower and at higher risk for overfitting
            - CNN shares weight tensors
        - but you use the fully connected layers to combine information from the signal and output a "score" for each class
    - "score" is a raw "probability" of an event occurring in a given geographic cluster
    - output size is M-1 - length vector (so the number of clusters, i.e., 6)
- last: softmax function
    - normalizes the input vector into a probability distribution
        - i.e., forces the components to sum to 1 (and all components will be between 0 and 1)
    - prevents overfitting of data (so that the network will work on different data)

Training

- minimized a cross-entropy loss function
- measures the difference between two probability distributions: the predicted distribution (p - predicted by the NN) and the true distribution (q)
    - => p and q are vectors of probabilities that an event lies within the geographic region

- the first chunk of the L function is the standard cross-entropy function
- the part after the plus sign is the L2-regularization: puts a penalty on the weights
    - this just means you sum the squares of all the weights (and scale it by some factor lambda) → L increases for larger weights
        - so that network learns to prefer small weights → reduces overfitting
        - here, lambda = 10^-3

- used the ADAM algorithm to actually do this (same as in lab)
- trained for 32,000 iterations (took 1.5 hours)