Machine Learning Course - CS-433

Neural Nets – Regularization, Data Augmentation, and Dropout

Dec 13, 2016

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Regularization

We have seen that for standard regression/classification tasks it is common to add a regularization term when learning. The same is true for neural networks. E.g., we might want to add a term of the form

$$\frac{1}{2}\sum_{l=1}^{L+1}\mu^{(l)}\|\mathbf{W}^{(l)}\|_F^2,$$

where $\mu^{(l)}$ is a non-negative constant that can depend on the layer. Note that it is common *not to penalize the bias terms* but only the weights.

Such a regularization term favors small weights and, combined with the right constants $\mu^{(l)}$, can avoid overfitting.

How does the gradient descent algorithm change if we use this form of regularization?[1] Assume that we use the same constant in all layers, i.e., $\mu^{(l)} = \mu$. Let $\Theta = w_{i,j}^{(l)}$ be the weight of the edge going from node i at layer $l-1$ to node j at layer l and let t be discrete time, increasing by one in each update step. We then get the update rule

$$\underbrace{\Theta[t+1]}_{\text{new value}} = \underbrace{\Theta[t]}_{\text{old value}} - \underbrace{\eta}_{\text{step size}} \Big(\underbrace{\nabla_\Theta \mathcal{L} + \mu\Theta[t]}_{\text{grad. data/regularization}} \Big)$$
$$= \underbrace{\Theta[t](1-\eta\mu)}_{\text{weight decay}} - \eta\nabla_\Theta\mathcal{L}.$$

We see that in one update step the weight is decreased by a factor $1-\eta\mu$ and in addition we add a small step in the

[1] This discussion is not restricted to neural nets but applies generally. We just happen to discuss this topic in the context of neural nets.

negative direction of the gradient. We say that regularization leads to *weight decay*.
Another popular method (Hinton et al, 2013) is not to put a penality on the square of the L_2 norm of the weights, but rather ask that the weight vector must have an L_2 norm no more than a constant, call it r. This can be easily incorporated into the gradient descent algorithm as follows. After each gradient step check wether the L_2 norm of the weight vector is above r or not. If it is, rescale the whole weight vector so that it has L_2 norm equal to r. This rescaling operation is trivial for the L_2 norm. The resulting algorithm is called the *projected* gradient descent algorithm (since we project after the gradient step the weight vector back to the sphere of radius r if necessary).
As a side remark – if we had added a regularization term which is the L_1 norm of the weight vector then this projection is not nearly as easy. But fortunately for neural nets the L_2 norm is the most useful regularization.

Dataset Augmentation

Data is scarce and valuable and the more data we have the better we can train. In some instances we can generate new data from the data we are given.
Consider a classification task with training set $S_t = \{(y_n, \mathbf{x}_n)\}$. Assume that there exists a transformation $\tau : \mathbb{R}^D \to \mathbb{R}^D$ that keeps the labels unchanged.
E.g., consider a hand-writing recognition task. We are given small squares (see Figure 1) each containing a digit from 0 to 9. The absolute position of the digit inside the square and

the exact orientation of the digit do not matter and can be changed without changing the label.
We can therefore take the data we are given, create variants of it, and add it to the date. Figure 1 shows some characters from the MNIST data set. Figure 2 shows some rotated variants and Figure 3 shows some shifted variants. This way

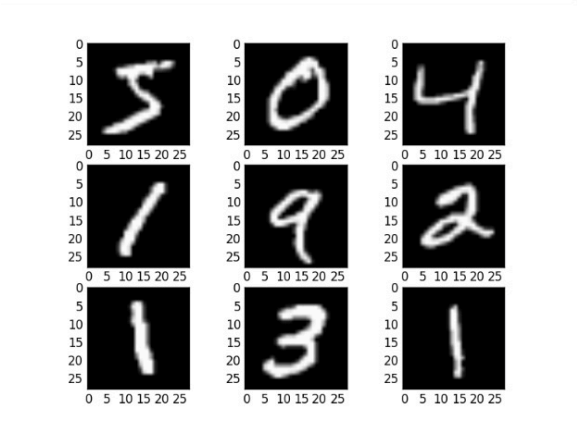

Figure 1: Some characters from the MNIST data set.

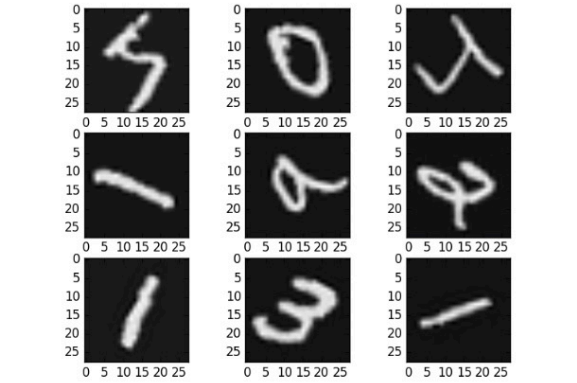

Figure 2: Rotated characters of the MNIST data set.

we can significantly increase our data which helps in training. In addition, if we train our network on this augmented data set then the network will automatically learn to become invariant to these transformations.

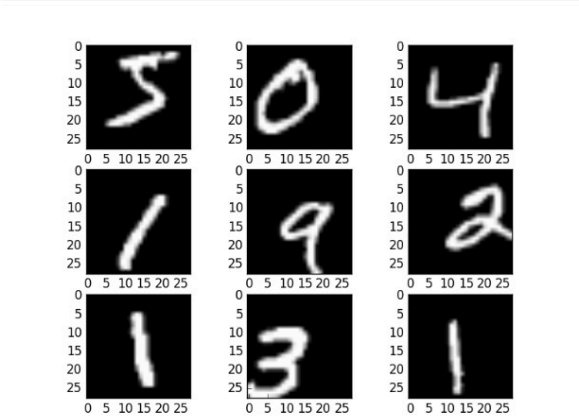

Figure 3: Shifted characters from the MNIST data set.

These transformations, if they exist, are very task specific. If we consider image recognition task some other possible transformations are *cropping* or *resizing*. More subtle, we can use the PCA and "compress" the image by only keeping the components corresponding to the largest singular values. This changes the photo globally but introduces only a minimal distortion (in the L_2 sense). In a similar sense, we can add some small amount of noise to our data.
There is another way in which we can "augment" our data. Assume that we have several distinct but related tasks. In this case we can train a network jointly whose "core" is used jointly for all tasks and where only the last layer is task specific. This is shown in Figure 4. The idea here is that for related tasks the same features are useful for the task.

Dropout

Dropout is a method both to avoid overfitting as well as to do model averaging (Hinton et al, 2012). By now, many variants have been proposed. Here is the original version.

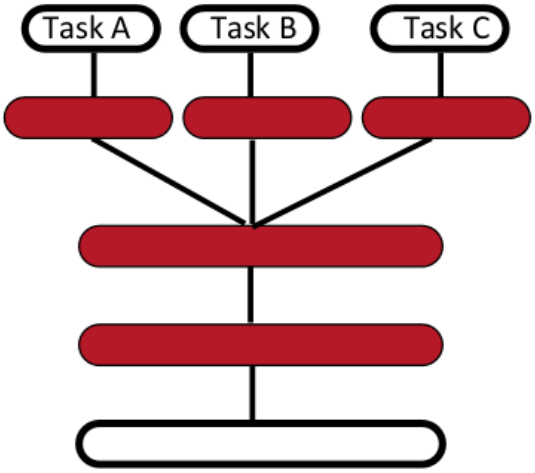

Figure 4: Neural network structure for multi-task learning.

Define the probability $p_i^{(l)}$. It is the probability of "keeping" the node i in layer l. Typical values are $p_i^{(0)} = 0.8$ (i.e., we keep in expectation 80 precent of all input nodes) and $p_i^{(l)} = 0.5$ for $l \geq 1$, (i.e., we keep in expectation 50 precent of all hidden nodes).
At every training step decide for each node i at level l according to the probability $p_i^{(l)}$ whether to keep this node or not. This defines a "subnetwork." Run one step of SGD (or perhaps a minibatch) and update the weights. Iterate until training is done.
For the prediction phase several variants are possible. Either generate K subnets in the same manner as before, predict for each and average the prediction. Alternatively use the whole network for the prediction. But in this case scale the output of node i at level l by the factor $p_i^{(l)}$. This guarantees that the expected input at each node stays the same as the expected input during training.
There are two benefits to this "dropout" procedure. First, it has been observed that this procedure limits overfitting.

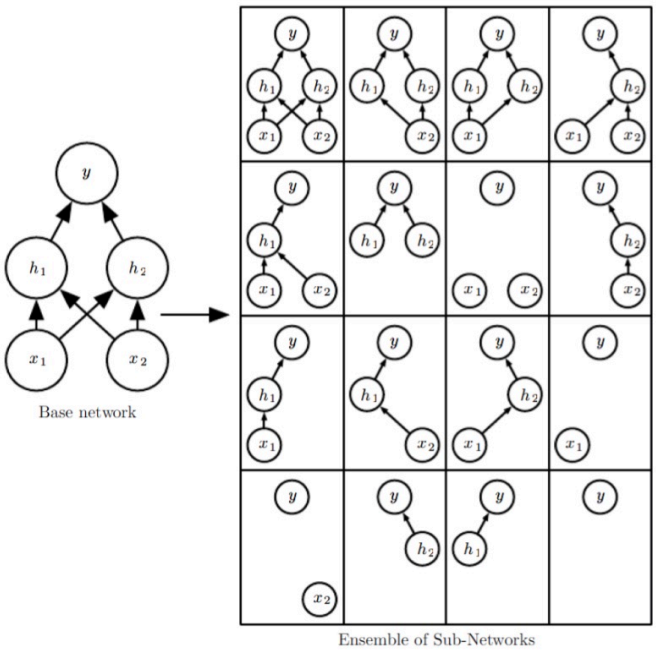

Figure 5: All the subnetworks of a given network.

Intuitively, nodes cannot "rely" on other nodes being present. Second, note that there is an exponential number of "sub-networks." This is shown in Figure 5 for a very small toy network. The effect of dropout is that we are performing an average over several (sub)networks. We either do this explicitly by running over several of them, computing the output, and then average. Or we do this implicitly by using the whole network but with reduced weights. We therefore get the advantage that comes with model averaging.

Averaging over many models is a standard ML trick and it is called *bagging*. It typically leads to improved performance. But dropout is quite different from standard bagging since we do not train K networks and then average. Rather, all these networks share the same weights. In fact, this characteristic seems to be an important component to explain their good performance.

In dropout we remove whole nodes. It is of course also pos-

sible to remove individual edges independently from each other.