2021/12/15

Regularization

1. Weight Regularization

A common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular.

This is called **weight regularization**, and it's done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors:

- L1 regularization—The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).
- L2 regularization—The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks.

Theoretical Minimum: L2 Regularization for Neural Network

As we have covered in Chapter 5, the idea of regularization is to add a cost (a regularization term) to the error (loss) function E, so we will have:

$$E^{improved} := E^{original} + RegularizationTerm$$

For L2 regularization, the error function is defined as:

$$E(\mathbf{w}) = E_0(\mathbf{w}) + \frac{\lambda}{2n}\sum_{\mathbf{w}} \mathbf{w}^2$$

where E_0 is the original error function without regularization. λ is a hyperparameter to adjust how much of the regularization we want to use (called the regularization parameter or regularization rate). The hyperparameter λ is divided by the size of the batch *n* used (to account for the fact that we want it to be proportional)

Why does L2 regularization reduce overfitting? Well, let's look at how the weights are updated during the backpropagation process. We learned previously that the optimizer calculates the derivative of the error, multiplies it by the learning rate, and subtracts this value from the old weight. Here is the backpropagation equation that updates the weights for each layer:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot \frac{\partial E}{\partial \mathbf{W}^{(l)}}$$

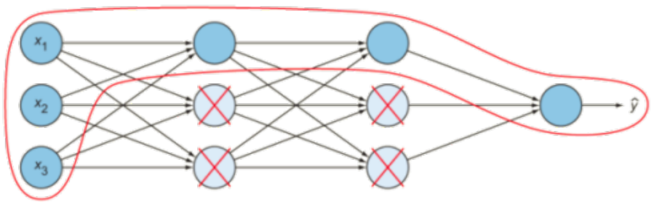
$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot (\frac{\partial E_0}{\partial \mathbf{W}^{(l)}} + \frac{\lambda}{n}\mathbf{W}^{(l)})$$

We know how to obtain $\frac{\partial E_0}{\partial \mathbf{W}^{(l)}}$ using backpropagation. The penalty term $\frac{\lambda}{n}\mathbf{W}^{(l)}$ adds a component term to the weight update that is in the negative direction of $\mathbf{w}$, i.e., toward zero weights – hence the term **weight decay**, as it forces the weights to decay toward zero (but not exactly zero).

Remark: reducing the weights leads to a simpler neural network

To see how this works, consider: if the regularization term is so large that, when multiplied with the learning rate, it will be equal to $\mathbf{W}^{(l)}$, then this will make the new weight equal to zero. This cancels the effect of this neuron, leading to a simpler neural network with fewer neurons.

In practice, L2 regularization does not make the weights equal to zero. It just makes them smaller to reduce their effect. A large regularization parameter (λ) lead to negligible weights. When the weights are negligible, the model will not learn much from these units. This will make the network simpler and thus reduce overfitting.

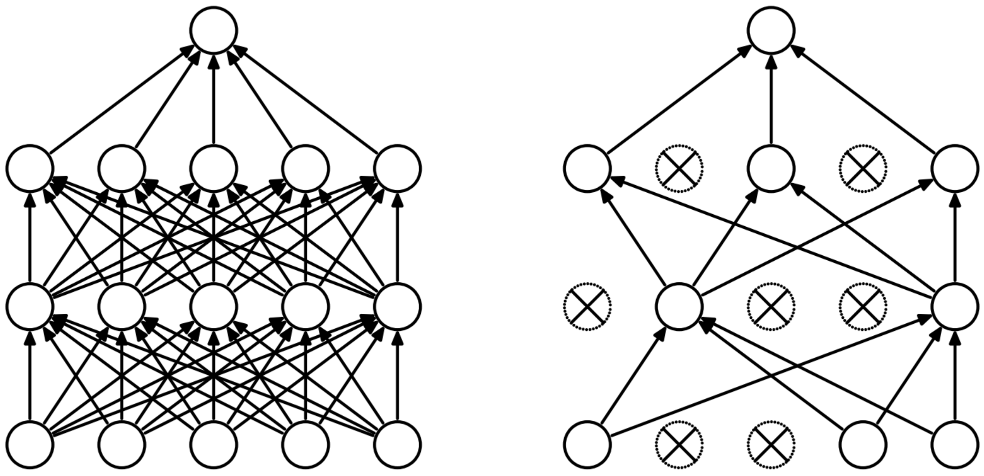


In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Let's add L2 weight regularization to the simple CIFAR-10 baseline network.

```
from tensorflow.keras import regularizers

# l2 regularization hyperparameter
weight_decay = 1e-4

# instantiate an empty sequential model
model = Sequential()

# CONV1

model.add(Conv2D(base_hidden_units, (3,3), padding='same',
              kernel_regularizer=regularizers.l2(weight_decay),
            input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
```

2. Dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks. Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.


(a) Standard Neural Net (b) After applying dropout.

The dropout algorithm is fairly simple: at every training iteration, every neuron has a probability p of being temporarily ignored (dropped out) during this training iteration. This means it may be active during subsequent iterations. While it is counterintuitive to intentionally pause the learning on some of the network neurons, it is quite surprising how well this technique works. The probability p is a hyperparameter that is called dropout rate and is typically set in the range of 0.3 to 0.5. Start with 0.3, and if you see signs of overfitting, increase the rate.

Remark: you can think of dropout as tossing a coin every morning with your team to decide who will do a specific critical task. After a few iterations, all your team members will learn how to do this task and not rely on a single member to get it done. The team would become much more resilient to change.

In `Keras` you can introduce dropout in a network via the Dropout layer, which gets applied to the output of layer right before it. Dropouts are usually advised **NOT to use after the convolution layers as you might lose important features**; they are mostly used after the dense layers of the network.

```
model.add(Dense(16, activation='relu', input_shape=(10000,)))
model.add(Dropout(0.3))
model.add(Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
```

In VGG-like convnet from `Keras`, **dropout is used after pooling**:

```
model = Sequential()
# input: 100x100 images with 3 channels -> (100, 100, 3) tensors.
# this applies 32 convolution filters of size 3x3 each.
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100,
3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

Remark: Both L2 regularization and dropout aim to reduce network complexity by reducing its neurons' effectiveness. The difference is that dropout completely cancels the effect of some neurons with every iteration, while L2 regularization just reduces the weight values to reduce the neurons' effectiveness. Both lead to a more robust, resilient neural network and reduce overfitting. It is recommended that you use both types of regularization techniques in your network.