DL in Applied Mathematics

Lecture 7: Avoiding Overfitting Through Regularization. Max-Norm Regularization

Avoid Overfitting with Regularization

In this class We'll do three things: (a) define the problem that we want to tackle with regularization; then (b) examine how exactly regularization helps; and finally (c) explain how regularization works in action.

What is the problem?

Let's say you want to predict house prices based on some features. You start with one feature, floor area, and you build your first regression model.

$$house\_price = a + b1 * floor\_area + e$$

At the other extreme, you could end up selecting 200 different features that can potentially impact house prices. So you built a really complex model and tested it on the training data and found that it performed great!

So how to find the sweet spot where a model is NOT too complex but complex enough to pick up the signal and performs relatively well in out-of-sample data?

How exactly regularization helps?

Ideally, if we had a large number of features, we'd add in features one by one and in different combinations to see their impacts on model performance and choose the best model based on the performance metric.

- *house_price = floor_area + garage_condition — — — — — — — (model 1)*
- *house_price = garage_condition+ bedrooms — — — — — — — -(model 2)*
- *house_price = floor_area + garage_condition+ bedrooms — —(model 3)*
- … and so on.

Do you see the problem here?

Overfitting with lots of features that we actually want to avoid?

How does it work in practice?

Let’s start with the **cost function** (a.k.a **objective function)** that we want to optimize in regression.

You know what a cost function is, right?

There are several cost functions out there such as Mean Squared Error (MSE), Mean Absolute Error (MAE), Root Mean Squared Error (RMSE) etc.

How MSE works:

- it takes differences between observed and predicted values (Y — Y-hat) for each data point (*i*),
- squares the difference,
- repeats the process for all points,
- sums them up, and finally
- takes an average by dividing by the number of data points (*n*).

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

If our regression model is $\hat{Y} = \alpha + \theta i\ Xi$ (where θ is the coefficient of X), then the cost function following MSE formulation above is:

$$\mathbf{Cost} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \theta_i X_i)^2$$

So the purpose of regularization is to add a small bias in the error function:

$$\text{Cost}(x, y) = \text{Error}(x, y) + \text{Regularization term}$$

There are two kinds of regularization terms — L1 and L2. Depending on which term is used, a normal multiple regression is called by different names.

Ridge regression

We call a normal regression the “Ridge regression” when it uses **L2 Regularization.**

The purpose of L2 is to shrink feature coefficients to close to zero, but not exactly zero.

So the cost function we want to minimize with Ridge regression is:

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \theta_i X_i)^2 + \lambda \sum \theta_i^2$$

L2 regularization term

LASSO regression

It sets some feature coefficients to zero through **L1 Regularization.** This process essentially eliminates those features from the model instead of minimizing their impacts.

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \theta_i X_i)^2 + \lambda \sum |\theta_i|$$

L1 regularization term

For example, assuming you have just one hidden layer with weights weights1 and one output layer with weights weights2, then you can apply ℓ1 regularization like this:

- **[...]** ***# construct the neural network***
- **base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")**
- **reg_losses = tf.reduce_sum(tf.abs(weights1)) + tf.reduce_sum(tf.abs(weights2))**
- **loss = tf.add(base_loss, scale * reg_losses, name="loss")**

However, if there are many layers, this approach is not very convenient.

The following code puts all this together:

- **with arg_scope(**
- **[fully_connected],**
- **weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)):**
- **hidden1 = fully_connected(X, n_hidden1, scope="hidden1")**
- **hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")**
- **logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope ="out")**

You just need to add these regularization losses to your overall loss, like this:

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)

loss = tf.add_n([base_loss] + reg_losses, name="loss")

Hyperparameter λ:

This is the only parameter responsible for penalizing the features.

What values does λ take and how to find the perfect value?

There are several ways to find the λ value. But know that two popular methods are: **gradient descent** and **cross-validation**.

There is another easy getaway — using both L1 and L2 regularization in the same regression. When we use both regularization terms in the model, it is called **ElasticNet Regression**.

Example:

```
#Add Regulization
# https://stackoverflow.com/questions/36706379/how-to-exactly-add-l1-regularisation-to-tensorflow-error-function
total_loss = meansq #or other loss calcuation
l1_regularizer = tf.contrib.layers.l1_regularizer(
scale=0.005, scope=None
)
weights = tf.trainable_variables() # all vars of your graph
regularization_penalty = tf.contrib.layers.apply_regularization(l1_regularizer, weights)
regularized_loss = total_loss + regularization_penalty # this loss needs to be minimized
train_step = tf.train.GradientDescentOptimizer(0.05).minimize(regularized_loss)
```

Dropout

- The most popular regularization technique for deep neural networks is arguably dropout. It is a fairly simple algorithm: at every training step, every neuron has a probability p of being temporarily “dropped out,” meaning it will be entirely ignored during this training step, but it may be active during the next step. The hyperparameter p is called the dropout rate, and it is typically set to 50%.

Example 2

```
# Set up the pooling layer with dropout using tf.nn.max_pool
with tf.name_scope("pool3"):
pool3 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding="VALID")
pool3_flat = tf.reshape(pool3, shape=[-1, pool3_fmaps * 14 * 14])
pool3_flat_drop = tf.layers.dropout(pool3_flat, pool3_dropout_rate,
training=training)
```

Data Augmentation

- One last regularization technique, data augmentation, generating new training instances from existing ones, artificially boosting the size of the training set. This will reduce overfitting. The trick is to generate realistic training instances

- For example, if your model is meant to classify pictures of mushrooms, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set.

- *#Image Augmentation*
  for img_id **in** imges:
  image = np.array(cv2.imread(train_dir + img_id))
  label = train_df[train_df['id'] == img_id]['has_cactus'].values[0]
  X_tr.append(image)
  Y_tr.append(label)
  X_tr.append(np.flip(image))
  Y_tr.append(label)
  X_tr.append(np.flipud(image))
  Y_tr.append(label)
  X_tr.append(np.fliplr(image))
  Y_tr.append(label)
  X_tr = np.asarray(X_tr).astype('float32')/225
  Y_tr = np.asarray(Y_tr)
- Check the full GitHub code at https://github.com/Cikbok/Week3Project.

Summary

To summarise, in this class we've focused on three things:

- **The problem**: if there is a large number of features in a dataset, we could easily end up overfitting the model by poorly selecting the features.
- **How regularization helps**: regularization is a technique to optimize model performance and it does so by adding a small bias in the cost function. This small bias shrinks feature coefficients and reduces their sensitivity.
- **How it works:** Two types of regularization are used — L1 (LASSO regression) and L2 (Ridge regression). They are controlled by a hyperparameter λ.