Neural network programming: improving things

Erik Spence

SciNet HPC Consortium

25 April 2019

Today's code and slides

You can get the slides for today's class at the SciNet Education web page.

`https://support.scinet.utoronto.ca/education`

Click on the link for the class, and look under "File Storage" and find the file "improvements.pdf".

Today's class

This class will cover the following topics:

- Regularization.
- Dropout.
- Different activation functions, cost functions.
- Our Keras network, improved.

Please ask questions if something isn't clear.

A review of last class

Recall what we did last class.

- We discussed the various neural network frameworks which are available.
- We rebuilt our MNIST neural network, using Keras.
- We trained the neural network.
- We got a result similar to our original network, with significant over-fitting (training: 84%, test: 62%).

The goal of this class is to eliminate the over-fitting, and get the highest accuracy we can using this approach.

Where did the over-fitting come from?

```
In [1]:
In [1]: import model1 as m1
In [2]: model = m1.get_model(30)
Using Theano backend.
In [3]: model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 30)                23550
_________________________________________________________________
output (Dense)               (None, 10)                310
=================================================================
Total params:  23,860
Trainable params:  23,860
Non-trainable params:  0
_________________________________________________________________
In [4]:
```

Over-fitting

Over-fitting occurs when a model is excessively fit to the noise in the training data, resulting in a model which does not generalize well to the test data.

It commonly occurs when there are too many free parameters (23,860) relative to the number of training data points (500).

This can be a serious issue with neural networks since it's trivially easy to have multitudes of weights and biases. How do we deal with this?

- More data! Either real (original), or artificially created.
- Regularization.
- Dropout.

The first is self-explanatory. We'll go over the later two cases.

Regularization

Regularization is an *ad hoc* technique by which parameters in a model are penalized to prevent individual parameters from becoming excessively important to the fit.

- This shows up all the time in contexts where over-fitting is to be expected, or the problem is ill-posed (inverse problems).
- It goes by many names: Tikhonov regularization, ridge regression...
- This usually manifests itself as a modification to the cost function, though there are other forms as well.

$$C = \frac{1}{2}\sum_i (a_{M_i} - v_i)^2 + \frac{\lambda}{2}\sum_j w_j^2$$

where λ is the "regularization parameter", and the sum over j is over all weights in the network.

Regularization, continued

If we call our previous cost function C_0, then the new cost function is

$$C = C_0 + \frac{\lambda}{2} \sum_j w_j^2$$

The addition of another term to the cost function changes the derivatives used to perform gradient descent.

$$\frac{\partial C}{\partial w_k} = \frac{\partial C_0}{\partial w_k} + \lambda w_k$$

where the derivatives of C_0 are calculated the usual way, using backpropagation. Our gradient descent parameter update now becomes

$$\begin{aligned} w_k &\rightarrow w_k - \eta \left[\frac{\partial C_0}{\partial w_k} + \lambda w_k \right] \\ &\rightarrow (1 - \eta\lambda)\, w_k - \eta \frac{\partial C_0}{\partial w_k} \end{aligned}$$

Regularization using Keras

Regularization, as you might expect, is built right into Keras.

- Regularization is done on a layer-by-layer basis.
- This allows you to only regularize some layers, and leave others out, if you so desire.
- Keras has three types of regularizers:
  - Kernel regularization: regularizes against the weights of the layer.
  - Bias regularization: regularizes against the biases of the layer.
  - Activity regularization: regularizes against the output of the layer.
- Because the regularization is specified layer-by-layer, the regularization is initialized as part of the layer declaration, rather than as part of the cost function.
- You can specify l1, l2 and custom regularizers.

Any combination of the regularizers can also be used.

Regularization using Keras, continued

```
# model3.py
import keras.models as km
import keras.layers as kl
import keras.regularizers as kr

def get_model(numnodes, lam = 0.0):

  model = km.Sequential()
  model.add(kl.Dense(numnodes,
    input_dim = 784, name = 'hidden',
    activation = 'sigmoid',
    kernel_regularizer = kr.l2(lam)))

  model.add(kl.Dense(10, name = 'output',
    activation = 'sigmoid',
    kernel_regularizer = kr.l2(lam)))

  return model
```

```
In [4]:
In [4]: from keras.datasets import mnist
In [5]: import keras.utils as ku
In [6]:
In [6]: (x_train, y_train), (x_test, y_test) = \
                    mnist.load_data()
In [7]:
In [7]: x_train2 = x_train[0:500, :, :].reshape(500, 784)
In [8]: x_test2 = x_test[0:100, :, :].reshape(100, 784)
In [9]:
In [9]: y_train2 = ku.to_categorical(y_train[0:500], 10)
In [10]: y_test2 = ku.to_categorical(y_test[0:100], 10)
In [11]:
```

Regularization using Keras, continued more

```
In [11]: import model3 as m3
In [12]: model3 = m3.get_model(30, lam = 0.001)
In [12]: model3.compile(optimizer = 'sgd', metrics = ['accuracy'], loss = "mean_squared_error")
In [13]:
In [13]: fit = model3.fit(x_train2, y_train2, epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
0s - loss: 0.1704 - acc: 0.1230
Epoch 2/1000
0s - loss: 0.1217 - acc: 0.1410
⋮
Epoch 1000/1000
0s - loss: 0.0433 - acc: 0.9860
In [14]: model3.evaluate(x_test2, y_test2)
100/100 [==============================] - 0s 11us/step
Out[14]: [0.06426530569791794, 0.78000000000000003]
In [15]:
```

Regularization, continued more

Why the improvement in the fitting of the test data?

- The regularization keeps the network from depending on any one particular weight, or set of weights (or biases), too much.
- This results in the network not focusing too much on any given feature, resulting in better generalization to the test data.
- As you might imagine, the regularization parameter is yet another hyperparameter in the development of our model.
- We've discussed some approaches for optimizing hyperparameters in previous classes.

Regularization isn't used as often as it used to be. Dropout is a much more popular approach.

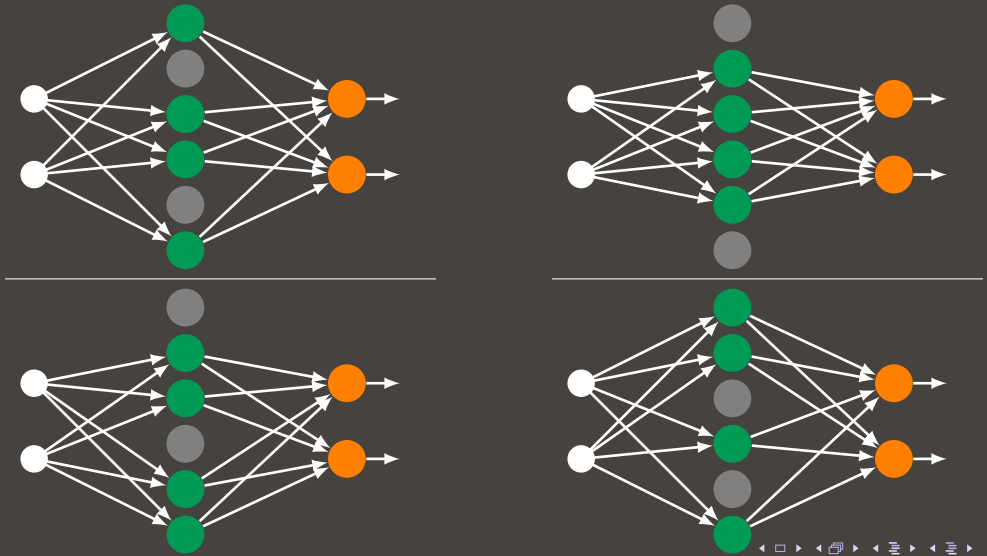
Dropout

Dropout is a uniquely-neural-network technique to prevent over-fitting.

- The principle is simple: randomly "drop out" neurons from the network during each batch of the stochastic gradient descent.
- Like regularization, this results in the network not putting too much importance on any given weight, since the weights keep randomly disappearing from the network.
- It can be thought of as averaging over several different-but-similar neural networks.
- Different fractions of different layers can be specified for dropout. A general rule of thumb is 30 - 50%.
- In the final model (after training):
  - the neurons in the dropout layer are no longer dropped out, and
  - the output from the neurons in the dropout layer is scaled by $(1 - p)$, where p is the probability of being dropped out.

This form of over-fitting control is quite common to encounter.

Dropout, visualized



Dropout using Keras

```
# model4.py
import keras.models as km
import keras.layers as kl

def get_model(numnodes, d_rate = 0.4):

  model = km.Sequential()

  model.add(kl.Dense(numnodes,
    input_dim = 784, name = 'hidden',
    activation = 'sigmoid'))

  model.add(kl.Dropout(d_rate,
    name = 'dropout'))

  model.add(kl.Dense(10, name = 'output',
    activation = 'sigmoid'))
  return model
```

```
In [15]: import model4 as m4
In [16]: model4 = m4.get_model(30, d_rate = 0.2)
In [16]: model4.compile(loss = "mean_squared_error",
    ...:   optimizer = 'sgd', metrics = ['accuracy'])
In [17]: fit = model4.fit(x_train2, y_train2,
    ...:   epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
0s - loss: 0.1727 - acc: 0.1600
⋮
Epoch 1000/1000
0s - loss: 0.0389 - acc: 0.8140
In [18]:
In [18]: model4.evaluate(x_test2, y_test2)
100/100 [==============================] - 0s
11us/step
Out[18]: [0.04947481036186218, 0.76000000000000001]
In [19]:
```

The next steps

We can do better. What's the plan? There are a few simple approaches:

- Use more data.
- Change the activation function.
- Change the cost function.
- Change the optimization algorithm.
- Change the way things are initialized.
- Add regularization, to try to deal with the over-fitting.

We'll try some of these next, but there are some not-so-simple approaches:

- Completely overhaul the network strategy.

We'll take a look at this as well, next class.

Other activation functions: relu

Two commonly-used functions:

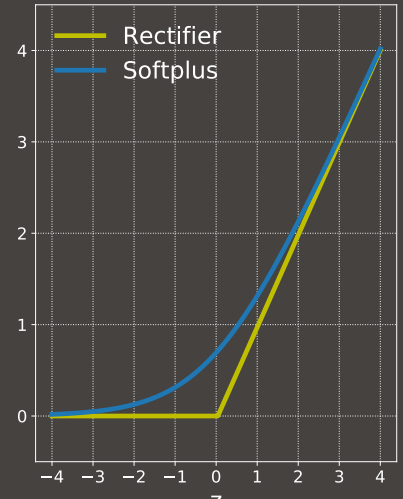
- Rectifier (also called Rectifier Linear Units, or RELUs):

  $f(z) = \max(0, z)$.

- Softplus:

  $f(z) = \ln(1 + e^z)$.

- Good: doesn't suffer from the vanishing-gradient problem.
- Bad: unbounded, could blow up.
- Other variants: leaky RELU, and SELU (scaled exponential).

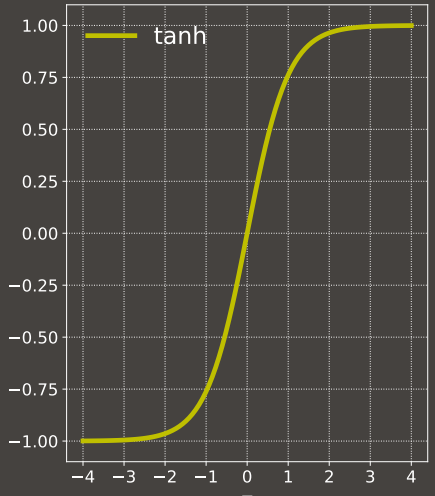


Other activation functions: tanh

Another commonly-used activation function is tanh:

$$f(z) = \tanh(z).$$

- Good: stronger gradients than sigmoid, faster learning rate, doesn't suffer from the vanishing-gradient problem.
- Good: because the function is anti-symmetric about zero. This also results in faster learning, at least for deeper networks.



Other activation functions: softmax

One of the more-commonly used output-layer activation functions is the softmax function:

$$s(\mathbf{z}_j) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}},$$

where N is the number of output neurons. The advantage of this function is that it converts the output to a probability.

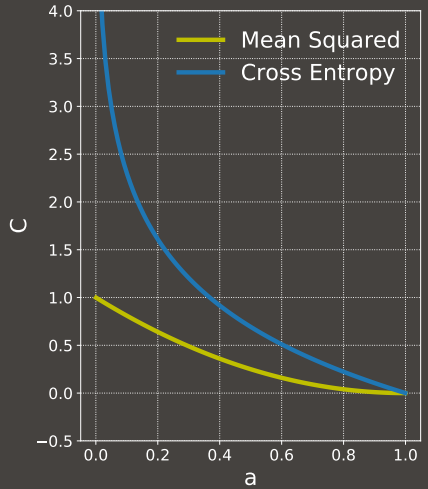
This is the activation function that is always used on the output layer when doing classification.

Other cost functions: cross entropy

The most-commonly used cost function for categorical output data is cross entropy:

$$C = -\frac{1}{n}\sum_{i}^{n}\left[v_i \log(a_i) + (1 - v_i)\log(1 - a_i)\right]$$

- Good: the gradient of cross entropy is directly proportional to the error; learning is faster than with mean squared error.
- Because $0 \leq a \leq 1$, this is always used with the softmax activation function as output.
- $v = 1$ in the example on the right.



Other optimization algorithms

There are many algorithms used to minimize the cost function.

- Gradient Descent, and its variations (RMSprop, Adam).
- Newton's method, uses the second derivatives of the cost function. Its variations ("Quasi-Newton") are gaining in popularity, especially DFP, and L-BFGS.
- Conjugate Gradient, which is like a combination of Gradient Descent and Newton's method.
- Levenberg-Marquardt (damped-least-squares) algorithm. Only works on squared cost functions (doesn't work on cross entropy).

If you find that your network won't train on a given optimization algorithm, it may be worth the effort to try a different one.

Our Keras network, revisited

What's our new strategy for our MNIST neural network?

- Use all of the data.
- Change our hidden layer activation function to tanh.
- Change our output layer activation function to softmax.
- Use the cross-entropy cost function.
- Use the Adam minimization algorithm.
- We won't add regularization or dropout, as the data set is larger than the number of parameters in the model.

Using regular gradient descent would also probably work. Using the rectifier linear unit activation function on the hidden layer is also an option.

Our Keras network, revisited

```
# model5.py
import keras.models as km
import keras.layers as kl

def get_model(numnodes):

  model = km.Sequential()

  model.add(kl.Dense(numnodes,
    input_dim = 784, name = 'hidden',
    activation = 'tanh'))

  model.add(kl.Dense(10, name = 'output',
    activation = 'softmax'))

  return model
```

```
In [19]:
In [19]: x_train.shape
Out[19]: (60000, 28, 28)
In [20]: x_test.shape
Out[20]: (10000, 28, 28)
In [21]:
In [21]: x_train = x_train.reshape(60000, 784)
In [22]: x_test = x_test.reshape(10000, 784)
In [23]:
In [23]: y_train = ku.to_categorical(y_train, 10)
In [24]: y_test = ku.to_categorical(y_test, 10)
In [25]:
```

Our Keras network revisited, continued

```
In [25]: import model5 as m5
In [26]: model5 = m5.get_model(30)
In [27]:
In [27]: model5.compile(loss = "categorical_crossentropy", optimizer = "adam",
...:            metrics = ['accuracy'])
In [28]:
In [28]: fit = model5.fit(x_train, y_train, epochs = 100, batch_size = 100, verbose = 2)
Epoch 1/100
2s - loss:  0.0688 - acc:  0.4576
Epoch 2/100
2s - loss:  0.3661 - acc:  0.7246
⋮
Epoch 100/100
2s - loss:  0.0103 - acc:  0.9338
In [29]:
```

Our Keras network revisited, continued more

Now check against the test data.

93%! Better!

```
In [29]:
In [29]: score = model5.evaluate(test_in, test_out)
In [30]:
In [30]: score
Out[30]: [0.010993927612225524, 0.92949999999999999]
In [31]:
```

Batch normalization

A relatively new introduction to the world of neural networks is the technique called 'batch normalization'.

- During regular SGD, the distribution of inputs to a given layer will evolve as the weights and biases change during training. This has a name: "covariate shift".
- Neural networks train more quickly if the input to a given layer is transformed such that it has a Gaussian distribution.
- Batch normalization performs the operation $x \rightarrow \frac{x - \mu}{\sigma}$.
- Rather than do it over all the input, it is just done over each SGD batch. Hence the name "batch normalization".
- This has the side effect of also acting like a type of regularization.

This is especially used in deep neural networks, which are harder to successfully train.

Batch normalization using Keras

```
# model6.py
import keras.models as km
import keras.layers as kl

def get_model(numnodes):

  model = km.Sequential()

  model.add(kl.Dense(numnodes,
    input_dim = 784, name = 'hidden',
    activation = 'tanh'))

  model.add(kl.BatchNormalization(
    name = 'batch_normalization'))

  model.add(kl.Dense(10, name = 'output',
    activation = 'softmax'))
  return model
```

```
In [31]: import model6 as m6
In [32]: model6 = m6.get_model(30)
In [32]:
In [32]: model6.compile(optimizer = 'adam',
    ...:  metrics = ['accuracy'],
    ...:  loss = "categorical_crossentropy")
In [33]:
In [33]: fit = model6.fit(x_train, y_train,
    ...:  epochs = 100, batch_size = 100, verbose = 2)
Epoch 1/100
0s - loss: 0.9230 - acc: 0.7111
Epoch 2/100
0s - loss: 0.5252 - acc: 0.8380
⋮
Epoch 100/100
0s - loss: 0.2337 - acc: 0.9318
In [34]:
```

Batch normalization using Keras, continued more

We always test against the test data.

93%! About the same as the previous run.

Batch normalization speeds up the training (as seen on the previous slide), but doesn't necessarily improve the final result.

That being said, if you're training deep network it is worth considering.

```
In [34]:
In [34]: score = model6.evaluate(test_in, test_out)
In [35]:
In [35]: score
Out[35]: [0.23953827554583548, 0.93000000000000005]
In [36]:
```

Other topics

This course is short. There is not time to cover every topic. Some topics you should look into further, if you're going to use NNs in your research:

- preprocessing data: remove unnecessary degrees of freedom, scale and centre the data.
- parameter initialization: how the weights and biases are initialized sometimes matters.
- activation functions: there are several activation functions which are used in specific areas of neural networks. Learn which ones are used in your field.
- more cost functions: there are other cost functions which are used in specific applications.
- training failures: the disappearing gradient problem, the exploding gradient problem.

But at this point we have covered enough of the very basics to get you started.