Neural network programming: neural network frameworks

Erik Spence

SciNet HPC Consortium

14 April 2026

Today's code and slides

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

`https://scinet.courses/1400`

Click on the link for the class, and look under "Lectures", click on "Frameworks".

Today's class

This class will cover the following topics:

- Review some of the available neural network frameworks,
- Redo the MNIST example using Keras,
- Dropout,
- Different activation functions, cost functions,
- Updated Keras network.

We will be using Keras for the rest of the course. You will need to install it, by installing Tensorflow. If you have problems, let me know.

A review of last class

Recall what we did last class.

- We built a neural network, consisting of three layers: an input layer, a single hidden layer, and an output layer.
- We defined a cost function, which measured the inaccuracy of the neural network's predictions.
- We used backpropagation to calculate the derivatives of the cost function with respect to the weights and biases.
- Using gradient descent, we trained the network to identify images from the MNIST data set.

However, we built all the parts of the network by hand. There are better ways to do this.

Neural network frameworks

Now that we have a sense of how neural networks work, we're ready to switch gears and use a 'framework'. Why would we do that?

- Coding your own networks from scratch can be a bit of work. (Though it's easier and cleaner if you use classes.)
- Neural network (NN) frameworks have been specifically designed to solve NN problems.
- Python, of course, is not a high-performance language.
- The neural networks which are built using frameworks are compiled before being used, thus being much faster than Python.
- The NN frameworks are also designed to use GPUs, which make things significantly faster than just using CPUs.

The training of neural networks is particularly well suited to GPUs.

TensorFlow

TensorFlow is Google's NN framework.

- Released as open source in November 2015.
- The second-generation machine-learning framework developed internally at Google, successor to DistBelief.
- More flexible than some other neural network frameworks.
- Capable of running on multiple cores and GPUs.
- Provides APIs for Python, C++, Java and other languages.
- Used to be quite a challenge to learn (many ways to do the same thing).
- With Tensorflow 2.0, Keras became the main high-level API. A significant consolidation of the API was performed.
- Will be succeeded by JAX?

This framework was popular, though not necessarily the fastest.

SciNet

JAX

JAX is a numpy-like machine learning framework.

- Released by Google as open source in 2018.
- Designed for high-performance numerical computing.
- Easily takes numpy commands and runs them on GPUs, using a just-in-time compiler.
- Very fast.
- A number of libraries have been built on top of JAX, to extend its capabilities: Flax, Equinox, RLax, jraph, and others.

This framework is growing in popularity.

PyTorch

Another framework used for neural networks is PyTorch.

- Based on Torch, which was first released in 2002. Quite mature at this point.
- PyTorch was released by Facebook in January 2018. This is now the most-commonly used interface to Torch, though there is also a C++ interface.
- PyTorch is more flexible than just NN. It is more of a generic scientific computing framework.
- Very strong on GPUs.
- Very fast. Often the fastest depending on the problem being considered.
- Used and maintained by Facebook, Twitter and other high-profile companies.
- PyTorch Lightning has been released. This gives a more Keras-like interface to PyTorch, making it easier to use.

This framework is the other major player, other than JAX.

Keras

We will use Keras for the rest of this course.

- Keras is a NN framework, but it's only the top-most level.
- More accurately, it's an API standard for creating neural networks.
- Designed for fast development of networks.
- Runs on top of a 'back end', which by default is now TensorFlow.
- It can currently can run on Tensorflow, Torch or JAX.
- Historically it ran on top of many other backends also: Theano, CNTK, MXNet, TypeScript, JavaScript, PlaidML, Scala, CoreML, and others.
- Because it's a proper framework, all of the NN goodies you need are already built into it.
- Keras 3 had a complete rewrite in 2023, and is now independent of TensorFlow.
- I'll be using Keras 3.9.2.

Getting the data

Because it is so commonly used, the MNIST data set is built into most NN frameworks.

Keras is no different, so we'll just grab it from Keras directly.

```
In [1]:
In [1]: from keras.datasets import mnist
In [2]:
In [2]: (x_train, y_train), (x_test, y_test) =
                    mnist.load_data()
In [3]:
In [3]: x_train.shape
Out[3]: (60000, 28, 28)
In [4]:
```

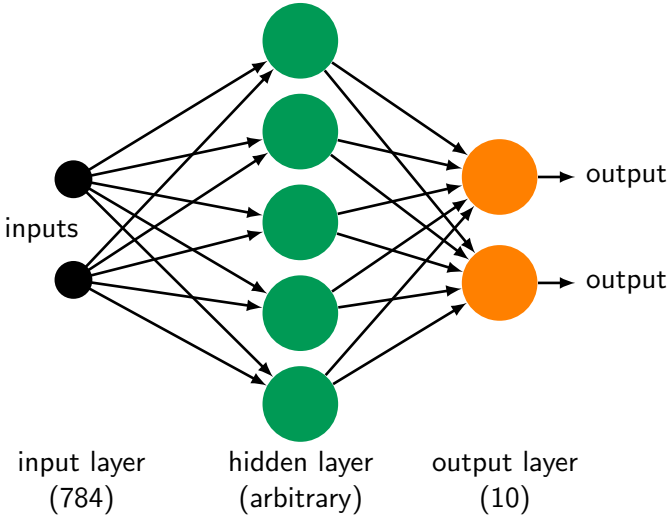
Prepping the data

As with last time, we need the data in a specific format:

- Instead of 28 x 28, we flatten the data into a 784-element vector.
- We only take the first 500 data points for training, to be consistent with last class.
- The labels must be changed to a categorical format (one-hot encoding).

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

Our neural network



Our network using Keras

Let us re-implement our second network using Keras.

- A "Sequential" model means the layers are in a line.
- A "Dense" ("fully-connected") layer is the regular layer we've been using.
- Use "input_dim" in the first layer to indicate the shape of the incoming data.
- The "activation" is the output function of the neuron.
- The "name" of the layer is optional.
- You may get a warning message, saying you should use an input layer.

```
# model1.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,
    activation = 'sigmoid', name = 'hidden'))
  model.add(kl.Dense(10, name = 'output',
    activation = 'sigmoid'))
  return model
```

```
In [10]: import model1 as m1
In [11]: model = m1.get_model(30)
In [12]: model.output_shape
Out[12]: (None, 10)
In [13]:
```

SciNet

Our network using Keras, continued

```
In [13]:
```

```
In [13]: model.summary()
Model:  "sequential"
---------------------------------------------------------------
| Layer (type)              |  Output Shape       |      Param #  |
---------------------------------------------------------------
| hidden (Dense)            |  (None, 30)         |       23,550  |
---------------------------------------------------------------
| output (Dense)            |  (None, 10)         |          310  |
---------------------------------------------------------------
Total params:  23,860 (93.20 KB)
Trainable params:  23,860 (93.20 KB)
Non-trainable params:  0 (0.00 B)
```

```
In [14]:
```

Our optimization flag

We will use the optimization flag "sgd".

- This stands for "Stochastic Gradient Descent".
- This is similar to regular gradient descent that we used previously.
  - Regular gradient descent is ridiculously slow on large amounts of data.
  - To speed things up, SGD uses a randomly-selected subset of the data (a "batch") to update the weights and biases.
  - This is repeated many times, using different batches, until all of the data has been used. This is called an "epoch".
- In practice, regular gradient descent is never used, stochastic gradient descent is used instead, since it's so much faster.
- The only real advantage of regular gradient descent is that it's easier to code, which is why I used it in previous classes.
- There are many variations on SGD that are also used.

Our network using Keras, continued more

```
In [14]:
In [14]: model.compile(optimizer = 'sgd', metrics = ['accuracy'], loss = "mean_squared_error")
In [15]:
In [15]: fit = model.fit(x_train2, y_train2, epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
100/100 - 0s - 254us/step - accuracy: 0.1170 - loss: 0.1963
Epoch 2/1000
100/100 - 0s - 254us/step - accuracy: 0.1720 - loss: 0.1338
⋮
Epoch 999/1000
100/100 - 0s - 254us/step - accuracy: 0.8440 - loss: 0.0394
Epoch 1000/1000
100/100 - 0s - 254us/step - accuracy: 0.8440 - loss: 0.0394
In [16]:
```

Our network using Keras, notes

Some notes about the compilation of the model.

- We must specify the loss (cost) function with the "loss" argument.
- We must specify the optimization algorithm, using the "optimizer" flag.
- The optimizer can be generic ('sgd'), as in this example, or you can specify parameters using the optimizers in the keras.optimizers module.
- I sometimes specify the optimizer explicitly so that I can specify the value of η (using 'lr', the 'learning rate').
- The 'metrics' argument is optional, but is needed if you want the accuracy to be printed.

Our network using Keras, continued even more

Now check against the test data.

We see the over-fitting rearing its head (84% versus 62%).

We can do better!

```
In [16]:
In [16]: score = model.evaluate(x_test2, y_test2)
In [17]:
In [17]: score
Out[17]: [0.056402873396873471, 0.62]
In [18]:
```

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 dropout today.

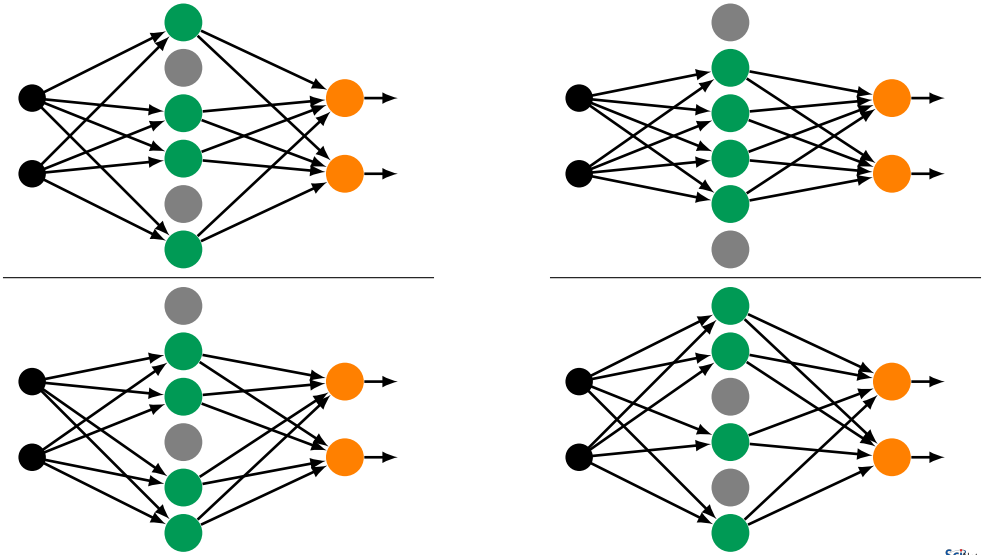
Dropout

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 $(\mathbf{1} - \boldsymbol{p})$, where $\boldsymbol{p}$ is the probability of being dropped out.

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

Dropout, visualized



Dropout using Keras

```
# model2.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 [18]: import model2 as m2
In [19]: model2 = m2.get_model(30, d_rate = 0.2)
In [19]: model2.compile(loss = "mean_squared_error",
    ...:  optimizer = 'sgd', metrics = ['accuracy'])
In [20]: fit = model2.fit(x_train2, y_train2,
    ...:  epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
100/100 - 0s - 257us/step - acc: 0.1600 - loss: 0.1727
⋮
Epoch 1000/1000
100/100 - 0s - 254us/step - acc: 0.7640 - loss: 0.0431
In [21]:
In [21]: model2.evaluate(x_test2, y_test2)
100/100 [==============================] - 0s
11us/step
Out[21]: [0.05245877802371979, 0.6399999856948853]]
In [22]:
```

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 class, but there are also some not-so-simple approaches:

- Completely overhaul the network strategy.

We'll take a look at this on Thursday.

Training maladies, an aside

There are three main ways that training can fail. Changing the activation function and cost function can assist with avoiding these modes of failure.

- Falling into a local minima.
  - Gradient descent in high-dimensional spaces can find local minima, and get stuck in them.
  - Some SGD variants will attempt to get unstuck.
  - Sometimes you just need to restart your training.
- Exploding gradient problem.
  - The gradient of the cost function blows up. Finite activation functions can help.
- Vanishing gradient problem.
  - The gradient of the cost function goes to zero, and training stops.
  - This happens in deep networks in particular, because the chain rule of derivatives results in smaller and smaller final derivatives.
  - Certain activation functions are more-prone to this failure mode, sigmoid in particular.
  - Techniques have been developed to avoid this problem.

Other activation functions: relu

Two commonly-used functions:

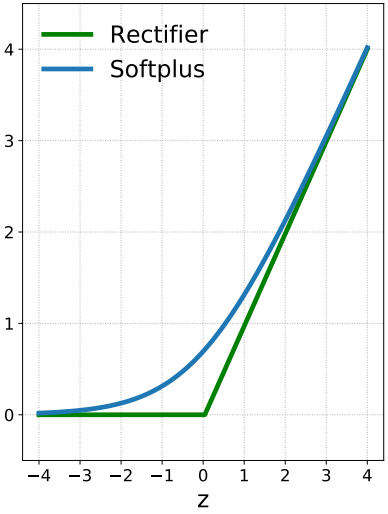
- Rectifier Linear Units (or RELUs):

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

- 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).

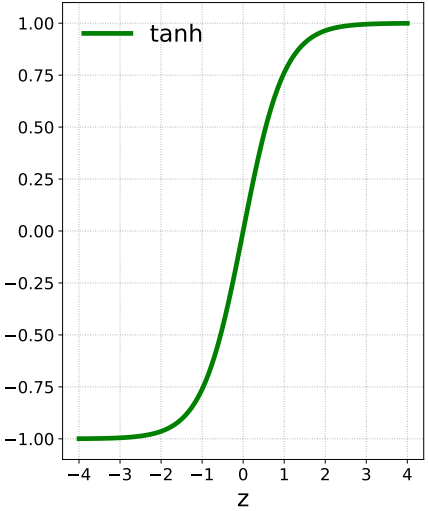


Other activation functions: tanh

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.
- Bad: the function saturates at large or small values of z.



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.

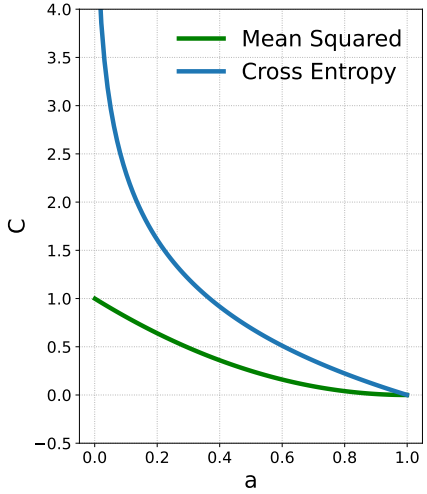
This is the activation function that is always used on the output layer when doing classification.

Other cost functions: cross entropy

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

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

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



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 stochastic 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

```
# model3.py
import tensorflow.keras.models as km
import tensorflow.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 [22]:
In [22]: x_train.shape
Out[22]: (60000, 28, 28)
In [23]: x_test.shape
Out[23]: (10000, 28, 28)
In [24]:
In [24]: x_train = x_train.reshape(60000, 784)
In [25]: x_test = x_test.reshape(10000, 784)
In [26]:
In [26]: y_train = ku.to_categorical(y_train, 10)
In [27]: y_test = ku.to_categorical(y_test, 10)
In [28]:
```

Our Keras network revisited, continued

```
In [28]: import model3 as m3
In [29]: model3 = m3.get_model(30)
In [30]:
In [30]: model3.compile(loss = "categorical_crossentropy", optimizer = "adam",
...:            metrics = ['accuracy'])
In [31]:
In [31]: fit = model3.fit(x_train, y_train, epochs = 100, batch_size = 128, verbose = 2)
Epoch 1/100
469/469 - 0s - 374us/step - acc: 0.4576 - loss: 0.0688
Epoch 2/100
469/469 - 0s - 374us/step - acc: 0.7246 - loss: 0.3661
⋮
Epoch 100/100
469/469 - 0s - 374us/step - acc: 0.9338 - loss: 0.0103
In [32]:
```

Our Keras network revisited, continued more

Now check against the test data.

93%! Better!

```
In [32]:
In [32]: score = model3.evaluate(x_test, y_test)
In [33]:
In [33]: score
Out[33]: [0.010993927612225524, 0.92949999999999999]
In [34]:
```

Linky goodness

Keras:

- `https://keras.io`

Other frameworks:

- `https://www.tensorflow.org`
- `https://pytorch.org`
- `https://docs.jax.dev/en/latest/quickstart.html`