Chapter 10. Introduction to Artificial Neural Networks

Key idea inspired *artificial neural networks*(ANNs): study brain's architecture for inspiration on how to build an intelligent machine.

ANNs are the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks.

From Biological to Artificial Neurons

ANNS first introduced as *propositional logic* in 1943 by Warren McCulloch and Walter Pitts.

In the early 1980s there was a revival of interest in ANNs as new network architectures were invented and better training techniques were developed. But by the 1990s, powerful alternative Machine Learning techniques such as Support Vector Machines.

Reasons to believe this wave of interest in ANNs is different and will have a much more profound impact on our lives:

- Huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- Computing power (Moore's Law, GPUs)
- The training algorithms have been improved.
- Some theoretical limitations of ANNs have turned out to be benign in practice.
- ANNs seem to have entered a virtuous circle of funding and progress.

Biological Neurons

Each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a vast network of fairly simple neurons.

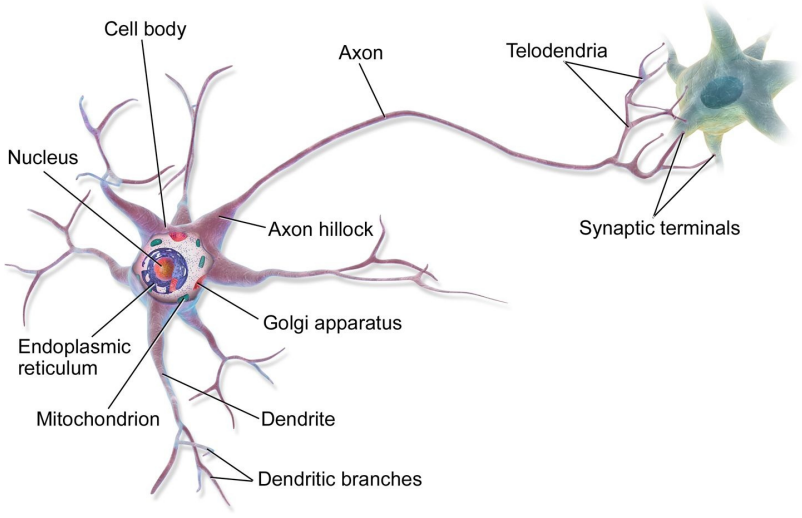

Figure 10-1. Biological neuron

Logical Computations with Neurons

Artificial neuron: one or more binary (on/off) inputs and one binary output. (Such simplified model can build a network of artificial neurons that computes any logical proposition.)

The perceptron

Perceptron: one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron called a *linear threshold unit* (LTU): the inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight.

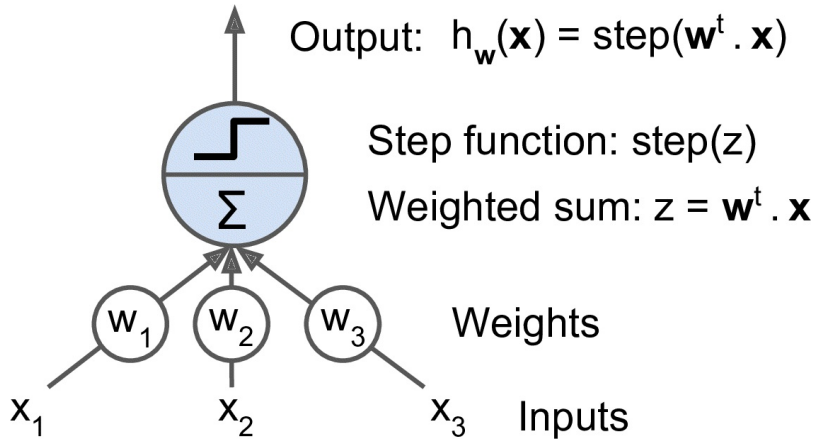

Figure 10-4. Linear shreshold unit

Common step functions used in Perceptrons

$$heaviside\,(z) = \begin{cases} 0 & if\, z < 0 \\ 1 & if\, z \geq 0 \end{cases} \qquad sgn\,(z) = \begin{cases} -1 & if\, z < 0 \\ 0 & if\, z = 0 \\ 1 & if\, z \geq 0 \end{cases}$$

A Perceptron is simply composed of a single layer of LTUs, with each neuron connected to all the inputs. These connections are often represented using special passthrough neurons called *input neurons*: they just output whatever input they are fed. Moreover, an extra bias feature is generally added ($x_0 = 1$). This bias feature is typically represented using a special type of neuron called a *bias neuron*, which just outputs 1 all the time.

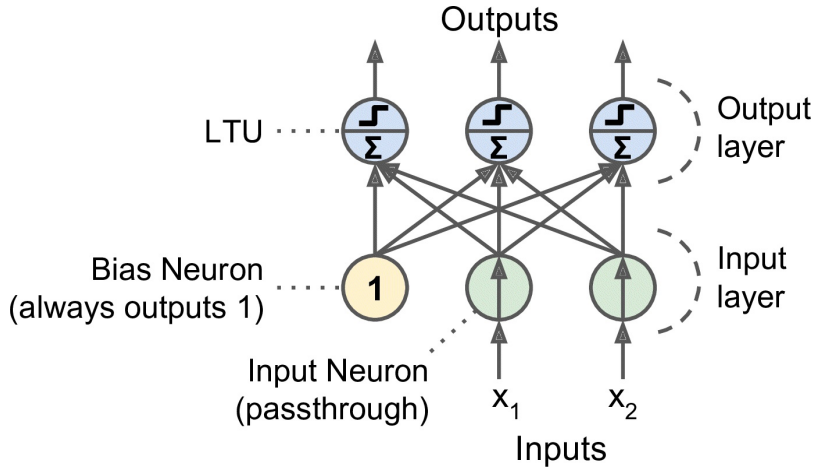

Figure 10-5. Perceptron diagram. This Perceptron can classify instances simultaneously into three different binary classes, which makes it a multioutput classifier.

How is a Perceptron trained?

Hebb's rule (Hebbian learning): the connection weight between two neurons is increased whenever they have the same output.

Perceptrons are trained using a variant of this rule that takes into account the error made by the network; it does not reinforce connections that lead to the wrong output. More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown as

Perceptron learning rule (weight update)

$$w_{i,j}^{next\_step} = w_{i,j} + \eta(\hat{y}_j - y_j)x_i$$

Perceptron convergence Theorem: if the training instances are linearly separable, this algorithm would converge to a solution.

Prefer Logistic Regression over Perceptrons, because instead of outputting a class probability, Perceptrons just make predictions based on a hard threshold.

Multi-Layer Perceptron (MLP) can eliminate some of the limitations of Perceptrons, while single-layer perceptrons are incapable of solving some trivial problems.

Multi_layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) input layer, one or more layers of LTUs, called *hidden layers*, and one final layer of LTUs called the *output layer*. Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

Deep neural network (DNN): ANN has two or more hidden layers.

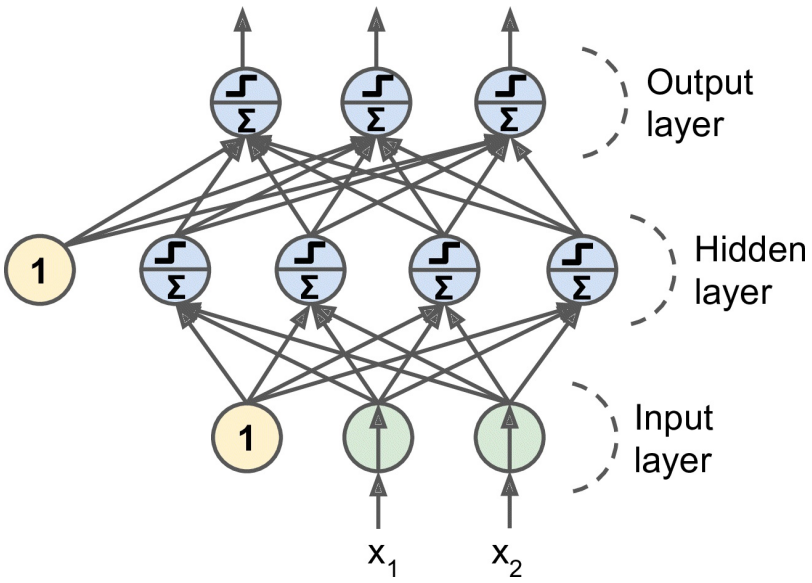

Figure 10-7. Multi-Layer Perceptron

Backpropagation training algorithm, same as Gradient Descent using reverse-mode autodiff: For each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally slightly tweaks the connection weights to reduce the error (Gradient Descent step).

Activation (Step) function with well-defined nonzero derivative everywhere:

- *Logistic function*, $\sigma(z) = 1/(1 + \exp(z)) \quad \in [0, 1]$
- *Hyperbolic tangent function*, $\tanh(z) = 2\sigma(2z) - 1 \quad \in [-1, 1]$, make each layer's output normalized (i.e., centered around 0) at the beginning of training. Speed up convergence.
- *ReLU function*, $ReLU(z) = \max(0, z) \quad \in [0, \infty)$, fast to compute gradient.

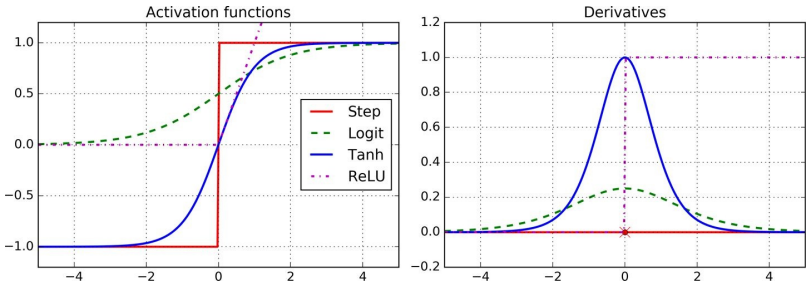

Figure 10-8. Activation functions ans their derivatives

An MLP is often used for classification, with each output corresponding to a different binary class. When the classes are exclusive, the output layer is typically modified by replacing the individual activation functions by a shared *softmax* function. The output of each neuron corresponds to the estimated probability of the corresponding class. Note that the signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

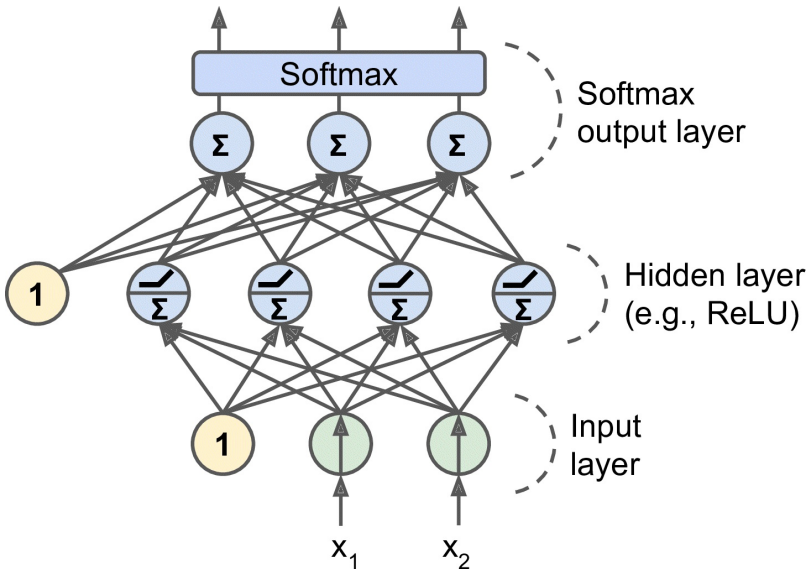

Figure 10-9. A modern MLP (including ReLU and softmax) for classification

NOTE

> Biological neurons seem to implement a roughly sigmoid (S-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that the ReLU activation function generally works better in ANNs.

Training an MLP with TensorFlow's High-Level API

Trains a DNN for classification with two hidden layers (one with 300 neurons, and the other with 100 neurons) and a softmax output layer with 10 neurons:

In [2]:
```
import numpy as np
import tensorflow as tf

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
X_train = X_train.astype(np.float32).reshape(-1, 28*28) / 255.0
X_test = X_test.astype(np.float32).reshape(-1, 28*28) / 255.0
y_train = y_train.astype(np.int32)
y_test = y_test.astype(np.int32)
X_valid, X_train = X_train[:5000], X_train[5000:]
y_valid, y_train = y_train[:5000], y_train[5000:]
```

In [3]:
```
feature_cols = [tf.feature_column.numeric_column("X", shape=[28 * 28])]
dnn_clf = tf.estimator.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                     feature_columns=feature_cols)

input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"X": X_train}, y=y_train, num_epochs=40, batch_size=50, shuffle=True)
dnn_clf.train(input_fn=input_fn)
```

```
INFO:tensorflow:Using default config.
WARNING:tensorflow:Using temporary folder as model directory: /tmp/tmpj1_i_7
bd
INFO:tensorflow:Using config: {'_evaluation_master': '', '_global_id_in_clus
ter': 0, '_num_ps_replicas': 0, '_master': '', '_keep_checkpoint_every_n_hou
rs': 10000, '_task_type': 'worker', '_save_summary_steps': 100, '_is_chief':
True, '_num_worker_replicas': 1, '_save_checkpoints_steps': None, '_model_di
r': '/tmp/tmpj1_i_7bd', '_tf_random_seed': None, '_task_id': 0, '_save_check
points_secs': 600, '_service': None, '_session_config': None, '_train_distri
bute': None, '_cluster_spec': <tensorflow.python.training.server_lib.Cluster
Spec object at 0x7fd876fa0f60>, '_log_step_count_steps': 100, '_keep_checkpo
int_max': 5}
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 1 into /tmp/tmpj1_i_7bd/model.ckpt.
```

In [4]:
```
test_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"X": X_test}, y=y_test, shuffle=False)
eval_results = dnn_clf.evaluate(input_fn=test_input_fn)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-07-12-15:25:47
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /tmp/tmpj1_i_7bd/model.ckpt-44000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-07-12-15:25:48
INFO:tensorflow:Saving dict for global step 44000: accuracy = 0.9784, average_l
oss = 0.11250929, global_step = 44000, loss = 14.241682
```

In [5]:
```
eval_results
```

Out[5]:
```
{'accuracy': 0.9784,
 'average_loss': 0.11250929,
 'global_step': 44000,
 'loss': 14.241682}
```

In [6]:
```
y_pred_iter = dnn_clf.predict(input_fn=test_input_fn)
y_pred = list(y_pred_iter)
y_pred[0]
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /tmp/tmpj1_i_7bd/model.ckpt-44000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
```

Out[6]:
```
{'class_ids': array([7]),
 'classes': array([b'7'], dtype=object),
 'logits': array([-11.688817  ,  -8.078765  ,  -4.9356127 ,   2.606026  ,
        -12.357452  , -14.665477  , -19.108204  ,  18.850456  ,
         -5.2496376 ,  -0.21223307], dtype=float32),
 'probabilities': array([5.4571077e-14, 2.0173785e-12, 4.6756488e-11, 8.8131955
e-08,
        2.7962592e-14, 2.7810878e-15, 3.2716234e-17, 9.9999988e-01,
        3.4155623e-11, 5.2623457e-09], dtype=float32)}
```

`DNNClassifier` class creates all the neuron layers, based on the ReLU activation function (change `activation_fn` hyperparameter). The output layer relies on the softmax function, and the cost function is cross entropy.

Training a DNN Using Plain TensorFlow

Mini-batch Gradient Descent to train it on the MNIST dataset. 1st step - construction phase, building the TensorFlow graph. 2nd step - the execution phase, where you actually run the graph to train the model.

Construction Phase

In [10]:
```
import tensorflow as tf

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

In [11]:
```
tf.reset_default_graph()

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int32, shape=(None), name="y")
```

In [12]:
```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)    # helps the algorithm converge much fas
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

In [13]:
```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1", activation=tf.nn.relu)
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2", activation=tf.nn.r
    logits = neuron_layer(hidden2, n_outputs, name="outputs")
    y_proba = tf.nn.softmax(logits)
```

In [15]:
```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=lo
    loss = tf.reduce_mean(xentropy, name="loss")
```

In [16]:
```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

In [17]:
```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

In [18]:
```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Note that `logits` is the output of the neural network *before* going through the `softmax` activation function: for optimization reasons, we will handle the softmax computation later.

You can use TensorFlow's `tf.layers.dense()` function to create a fully connected layer. methods: `name`, `activation`, `kernel_initializer`, the default `activation` is now `None`.

The `sparse_softmax_cross_entropy_with_logits()` function is equivalent to applying the softmax activation function and then computing the cross entropy, but it is more efficient, and it properly takes care of corner cases like logits equal to 0. There is also another function called `softmax_cross_entropy_with_logits()`, which takes labels in the form of one-hot vectors.

Execution Phase

In [26]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")

n_epochs = 20
batch_size = 50

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                            y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./logs/my_model_final.ckpt")
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
0 Train accuracy: 0.98 Test accuracy: 0.9131
1 Train accuracy: 0.96 Test accuracy: 0.929
2 Train accuracy: 0.9 Test accuracy: 0.9371
3 Train accuracy: 0.98 Test accuracy: 0.943
4 Train accuracy: 0.96 Test accuracy: 0.9482
5 Train accuracy: 0.98 Test accuracy: 0.9506
6 Train accuracy: 0.94 Test accuracy: 0.9547
7 Train accuracy: 1.0 Test accuracy: 0.958
8 Train accuracy: 0.94 Test accuracy: 0.9586
9 Train accuracy: 0.98 Test accuracy: 0.9616
10 Train accuracy: 1.0 Test accuracy: 0.9628
11 Train accuracy: 0.96 Test accuracy: 0.9643
12 Train accuracy: 1.0 Test accuracy: 0.9663
13 Train accuracy: 0.96 Test accuracy: 0.967
14 Train accuracy: 0.94 Test accuracy: 0.9667
15 Train accuracy: 0.98 Test accuracy: 0.9681
16 Train accuracy: 0.98 Test accuracy: 0.9693
17 Train accuracy: 1.0 Test accuracy: 0.9703
18 Train accuracy: 0.98 Test accuracy: 0.9697
19 Train accuracy: 1.0 Test accuracy: 0.9706
```

Using the Neural Network

First the code loads the model parameters from disk. Then it loads some new images that you want to classify. Remember to apply the same feature scaling as for the training data (in this case, scale it from 0 to 1). Then the code evaluates the `logits` node. If you wanted to know all the estimated class probabilities, you would need to apply the `softmax()` function to the logits, but if you just want to predict a class, you can simply pick the class that has the highest logit value (using the `argmax()` function does the trick).

In [ ]:
```
with tf.Session() as sess:
    saver.restore(sess, "./logs/my_model_final.ckpt")
    X_new_scaled = [...] # some new images (scaled from 0 to 1)
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Any *network topology*, number of layers, number of neurons per layer, type of activation function, weight initialization logic.

It is much better to use randomized search over grid search. Another option is to use a tool such as Oscar (http://oscar.calldesk.ai/) (http://oscar.calldesk.ai/)), which implements more complex algorithms to help you find a good set of hyperparameters quickly.

Number of Hidden Layers

MLP with just one hidden layer can model even the most complex functions provided it has enough neurons. However, deep networks have a much higher parameter efficiency than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, making them much faster to train.

Real-world data is often structured in such a hierarchical way and DNNs automatically take advantage of this fact.

Not only does this hierarchical architecture help DNNs converge faster to a good solution, it also improves their ability to generalize to new datasets.

In summary, for many problems you can start with just one or two hidden layers and it will work just fine. For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, typically require networks with dozens of layers (or even hundreds, but not fully connected ones), and they need a huge amount of training data. However, you will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will be a lot faster and require much less data.

Number of Neurons per Hidden Layer

As for the hidden layers, a common practice is to size them to form a funnel, with fewer and fewer neurons at each layer - the rationale being that many low-level features can coalesce into far fewer high-level features. However, this practice is not as common now, and you may simply use the same size for all hidden layers. Just like for the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. In general you will get more bang for the buck by increasing the number of layers than the number of neurons per layer.

A simpler approach is to pick a model with more layers and neurons than you actually need, then use *early stopping* to prevent it from overfitting (and other regularization techniques, especially *dropout*). This has been dubbed the “stretch pants” approach: instead of wasting time looking for

pants that perfectly match your size, just use large stretch pants that will shrink down to the right size.

Activation Functions

In most cases you can use the ReLU activation function in the hidden layers (or one of its variants). It is a bit faster to compute than other activation functions, and Gradient Descent does not get stuck as much on plateaus, thanks to the fact that it does not saturate for large input values (as opposed to the logistic function or the hyperbolic tangent function, which saturate at 1).

For the output layer, the softmax activation function is generally a good choice for classification tasks (when the classes are mutually exclusive). For regression tasks, you can simply use no activation function at all.