Texas A&M University

Survey on Deep Learning

Aryan Sharma
UIN: 326006767 — aryans@tamu.edu

October 4, 2018

Contents

1 Introduction

This is a survey of Neural Networks and Deep Learning.

2 Basics of Machine Learning

2.1 Validation

- Need to tune the hyperparameters involved
- Cannot use the test data for this as it will overfit
- Use cross-validation

3 Linear Classifiers

Linear mapping: $f(x_i, W, b) = Wx_i + b$. Set W and b in such way that the computed scores match the ground truth labels across the whole training set x.

3.1 Multiclass SVM loss

The SVM loss is set up so that the SVM wants the correct class for each image to a have a score higher than the incorrect classes by some fixed margin Δ. For the i^{th} example we are given the input x_i and the label y_i that specifies the index of the correct class. The score function takes the input and computes the vector $f(x_i, W)$ of class scores, which we will abbreviate to s (short for scores). For example, the score for the j^{th} class is the j^{th} element: $s_j = f(x_i, W)_j$. The SVM loss is then:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_i + \Delta)$$

.

L2 Regularization is added to the above equation that discourages large weights through an elementwise quadratic penalty over all parameters.

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

.

The margin Δ can be safely kept as 1 when regularization is added. The whole loss is then defined as:

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} [max(0, (f(x_i, W))_j - (f(x_i, W))_{y_i} + \Delta)] + \lambda \sum_k \sum_l W_{k,l}^2$$

3.2 Softmax Classifiers

Interpret the scores as the unnormalized log probabilities for each class and replace the hinge loss with a cross-entropy loss that has the form:

$$L_i = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}})$$

This provides probabilities for each class.

4 Neurons

4.1 Backpropagation

Every gate in a circuit diagram gets some inputs and can right away compute two things: 1. its output value and 2. the local gradient of its inputs with respect to its output value. Notice that the gates can do this completely independently without being aware of any of the details of the full circuit that they are embedded in. However, once the forward pass is over, during backpropagation the gate will eventually learn about the gradient of its output value on the final output of the entire circuit. Chain rule says that the gate should take that gradient and multiply it into every gradient it normally computes for all of its inputs. Backpropagation can thus be thought of as gates communicating to each other (through the gradient signal) whether they want their outputs to increase or decrease (and how strongly), so as to make the final output value higher.

4.1.1 Practical Aspects

1. Cache forward pass variables as they are needed to compute gradient in backprop.
2. Gradients add up at forks.
3. The add gate always takes the gradient on its output and distributes it equally to all of its inputs, regardless of what their values were during the forward pass.
4. The max gate routes the gradient during backprop.

4.2 Single neuron as a classifier

The neuron comes with an activation function like sigmoid. Optimizing the cross entropy loss can lead to binary softmax classifier or binary SVM classifier. The commonly used activation functions are:

4.2.1 Sigmoid

Sigmoid, $\sigma(x) = 1/(1 + e^{-x})$ takes a real-valued number and squashes it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1.

The drawbacks are:

- Sigmoids saturate and kill gradients. During backpropagation, the local gradient will be multiplied to the gradient of the gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively 'kill' the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. The network will barely learn.
- Sigmoid outputs are not zero-centered. This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = w^T x + b$), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights.

4.2.2 tanh

Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity.

4.2.3 ReLU

$f(x) = max(0, x)$

- It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold.

4.2.4 Leaky ReLU

Instead of the function being zero when x ¡ 0, a leaky ReLU will instead have a small negative slope. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons.

4.2.5 Maxout

Generalizes the ReLU and its leaky version. The Maxout neuron computes the function $max(w_1^T x + b1, w_2^T x + b2)$.

5 Neural Networks

Neural Network models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the fully-connected layer in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function. This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).

Neural Networks work well in practice because they compactly express nice, smooth functions that fit well with the statistical properties of data we encounter in practice, and are also easy to learn using our optimization algorithms (e.g. gradient descent). Similarly, the fact that deeper networks (with multiple hidden layers) can work better than a single-hidden-layer networks is an empirical observation, despite the fact that their representational power is equal.

5.1 Data Preprocessing

5.1.1 Mean substraction

Mean subtraction involves subtracting the mean across every individual feature in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension.

5.1.2 Normalization

Normalization refers to normalizing the data dimensions so that they are of approximately the same scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered. Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively.

5.1.3 PCA

In this process, the data is first centered as described above. Then, we can compute the covariance matrix that tells us about the correlation structure in the data. A nice property of `np.linalg.svd` is that in its returned value `U`, the eigenvector columns are sorted by their eigenvalues. We can use this to reduce the dimensionality of the data by only using the top few eigenvectors, and discarding the dimensions along which the data has no variance. This gives us the reduced dataset. This transformations is however, not used with Convolutional Networks. It is very important, though, to zero-center the data, and it is common to see normalization of every pixel as well.

5.2 Weight Initialization

5.2.1 Pitfall: All zero initialization

Shouldn't be done because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

5.2.2 Small random numbers

Initialize the weights of the neurons to small random numbers and refer to doing so as symmetry breaking. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network. One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. We can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in (i.e. its number of inputs). **Therefore**

$$W = np.random.randn(n) * \sqrt{\frac{1}{n}}$$

, where n is the number of its inputs. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence.

Consider the inner product $s = \sum_i^n w_i x_i$ between the weights w and input x, which gives the raw activation of a neuron before the non-linearity. Assuming zero mean inputs and weights and *iid* x_i and w_i, the variance of s is

$$Var(s) = Var(\sum_i^n w_i x_i) = (nVar(w))Var(x)$$

From this derivation we can see that if we want s to have the same variance as all of its inputs x, then during initialization we should make sure that the

variance of every weight w is $1/n$. And since $Var(aX) = a^2Var(X)$ for a random variable X and a scalar a, this implies that we should draw from unit gaussian and then scale it by $a = \frac{1}{\sqrt{n}}$, to make its variance $\frac{1}{\sqrt{n}}$.

Note that **in case of ReLU**, we use

$$W = np.random.randn(n) * \sqrt{\frac{2}{n}}$$

Batch Normalization: To properly initializing neural networks we explicitly force the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. BatchNorm layers are inserted immediately after fully connected layers (or convolutional layers), and before non-linearities. It improves gradient flow through the network, allows higher learning rates, and reduces the strong dependence on initialization.

5.3 Things to monitor while learning

5.3.1 Loss Function

The x-axis of the plots below are always in units of epochs, which measure how many times every example has been seen during training in expectation (e.g. one epoch means that every example has been seen once). The amount of 'wiggle' in the loss is related to the batch size. When the batch size is 1, the wiggle will be relatively high. When the batch size is the full dataset, the wiggle will be minimal because every gradient update should be improving the loss function monotonically (unless the learning rate is set too high).

5.3.2 Train/Val accuracy

The gap between the training and validation accuracy indicates the amount of overfitting. When there is a large gap, there is an overfitting. When we see this in practice we should increase regularization (stronger L2 weight penalty, more dropout, etc.) or collect more data.

The other possible case is when the validation accuracy tracks the training accuracy fairly well. This case indicates that the model capacity is not high enough: make the model larger by increasing the number of parameters.

5.4 Parameter Updates

5.4.1 Vanilla SGD

Change the parameters along the negative gradient direction (since the gradient indicates the direction of increase, but we usually wish to minimize a loss function): `x += -learning_rate * dx`

5.4.2 Momentum Update SGD

Here we see an introduction of a `v` variable that is initialized at zero, and an additional hyperparameter (`mu`). With Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient.

```
v = mu * v - learning_rate * dx ......# integrate velocity
x += v ..............................# integrate position
```

5.4.3 Nesterov Momentum SGD

The core idea behind Nesterov momentum is that when the current parameter vector is at some position x, then looking at the momentum update above, we know that the momentum term alone (i.e. ignoring the second term with the gradient) is about to nudge the parameter vector by $mu * v$. Therefore, if we are about to compute the gradient, we can treat the future approximate position $x + mu * v$ as a "lookahead" - this is a point in the vicinity of where we are soon going to end up. Hence, it makes sense to compute the gradient at $x+mu*v$ instead of at the "old/stale" position x. This is what we want to do.

```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

We can make thee updates similar to SGD by manipulating the update above with a variable transform $x\_ahead = x+mu*v$, and then expressing the update in terms of x_ahead instead of x. That is, the parameter vector we are actually storing is always the ahead version. The equations in terms of x_ahead (but renaming it back to x) then become:

```
v_prev = v # back this up
v = mu * v - learning_rate * dx # velocity update stays the same
x += -mu * v_prev + (1 + mu) * v # position update changes form
```

5.4.4 Annealing the learning rate over time

Knowing when to decay the learning rate can be tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. There are three common types of implementing the learning rate decay:

- **Step Decay:** Reduce the learning rate by some factor every few epochs.
- **Exponential decay**: $\alpha = \alpha_0 e^{-kt}$ where t is iteration number

- **1/t decay**: $\alpha = \alpha_0/(1 + kt)$ where t is iteration number

5.4.5 Second Order Updates

Newton's method which updates by calculating the Hessian matrix, which is a square matrix of second-order partial derivatives of the function. Multiplying by the inverse Hessian leads the optimization to take more aggressive steps in directions of shallow curvature and shorter steps in directions of steep curvature. There is no hyperparameters as well. However, the update above is impractical for most deep learning applications because computing (and inverting) the Hessian in its explicit form is a very costly process in both space and time.

$$x \leftarrow x - [H(f(x))]^{-1} \Delta f(x)$$

5.4.6 Adagrad

Adagrad is a per-parameter adaptive learning rate method. The variable `cache` has size equal to the size of the gradient, and keeps track of per-parameter sum of squared gradients. This is then used to normalize the parameter update step, element-wise. Notice that the weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased.

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

5.4.7 RMSprop

The RMSProp update adjusts the Adagrad method to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead. The cache here is leaky and decay_rate is a hyperparameter (typically 0.9, 0.99, 0.999).

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

5.4.8 Adam

RMSProp with momentum.

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

RMSprop with momentum and bias correction mechanism.

```
# t is your iteration counter going from 1 to infinity
m = beta1*m + (1-beta1)*dx
mt = m / (1-beta1**t)
v = beta2*v + (1-beta2)*(dx**2)
vt = v / (1-beta2**t)
x += - learning_rate * mt / (np.sqrt(vt) + eps)
```

5.5 Model Ensemble

Train multiple independent models, and at test time average their predictions. As the number of models in the ensemble increases, the performance typically monotonically improves (though with diminishing returns). Moreover, the improvements are more dramatic with higher model variety in the ensemble. Few approaches are:

- **Same model, different initializations.** Use cross-validation to determine the best hyperparameters, then train multiple models with the best set of hyperparameters but with different random initialization.
- **Top models discovered during cross-validation.** Pick the top few (e.g. 10) models to form the ensemble after finding hyperparamters from cross-validation.
- **Different checkpoints of a single model.**
- **Running average of parameters during training.** Maintain a second copy of the network's weights in memory that maintains an exponentially decaying sum of previous weights during training.

5.6 Transfer Learning

TODO

6 Convolutional Neural Networks

So what does change from the vanilla Neural Network? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

6.1 Architecture Overview

Neural Networks receive an input (a single vector), and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where

neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the "output layer" and in classification settings it represents the class scores. Regular Neural Nets don't scale well to full images and huge number of parameters quickly leads to overfitting.

A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters. Layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.

6.2 Layers used to build ConvNets

We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer.

- A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.
- RELU layer will apply an elementwise activation function, such as the max(0,x) thresholding at zero. No change in volume.
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height).
- FC (i.e. fully-connected) layer will compute the class scores
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

6.2.1 Convolutional Layer

The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume.

During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer.

We will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

Another view is that every entry in the 3D output volume can also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially.

We will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume.

Three hyperparameters control the size of the output volume: the **depth, stride and zero-padding**. First, the depth of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. Sometimes it will be convenient to pad the input volume with zeros around the border. It ensures that the input volume and output volume will have the same size spatially.The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly we use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

For input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border, the number of neurons is given by $(W - F + 2P)/S + 1$

A real example is that of ALexNet where they used neurons with receptive field size F=11, stride S=4 and no zero padding P=0. Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of K=96, the Conv layer output volume had size [55x55x96]. Each of the 55*55*96 neurons in this volume was connected to a region of size [11x11x3] in the input volume. Moreover, all 96 neurons in each depth column are connected to the same [11x11x3] region of the input, but of course with different weights.

Parameter sharing scheme is used in Convolutional Layers to control the

number of parameters. Using the real-world example above, we see that there are 55*55*96 = 290,400 neurons in the first Conv Layer, and each has 11*11*3 = 363 weights and 1 bias. Together, this adds up to 290400 * 364 = 105,705,600 parameters on the first layer of the ConvNet alone. Clearly, this number is very high.

Hence, there is a **Parameter sharing scheme** is used in Convolutional Layers to control the number of parameters. It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: That if one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x2,y2). In other words, denoting a single 2-dimensional slice of depth as a depth slice (e.g. a volume of size [55x55x96] has 96 depth slices, each of size [55x55]), **we are going to constrain the neurons in each depth slice to use the same weights and bias**. With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of 96*11*11*3 = 34,848 unique weights, or 34,944 parameters (+96 biases). Alternatively, all 55*55 neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

Backpropagation. The backward pass for a convolution operation (for both the data and the weights) is also a convolution (but with spatially-flipped filters).

6.2.2 Pooling Layer

Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting.

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1\text{-}F)/S + 1$
  - $H_2 = (H_1\text{-}F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

Backpropagation. The backward pass for a max(x, y) operation has a simple interpretation as only routing the gradient to the input that had the highest value in the forward pass. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called the switches) so that gradient routing is efficient during backpropagation.

Sometimes, pooling is disregarded as in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs)

6.2.3 Fully Connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks.

Any FC layer can be converted to a CONV layer. For example, an FC layer with $K = 4096$ that is looking at some input volume of size $7 \times 7 \times 512$ can be equivalently expressed as a CONV layer with $F = 7, P = 0, S = 1, K = 4096$. In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be $1 \times 1 \times 4096$ since only a single depth column 'fits' across the input volume, giving identical result as the initial FC layer.

Appendices

A A simple neural network model for recognizing digits on MNIST data

The following code sets up few layers of fully connected neural network. The accuracy given by this model over MNIST data is 97.56%.

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
from keras.utils import np_utils
import numpy as np

tf.logging.set_verbosity(tf.logging.ERROR)

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

learning_rate = 0.5
epochs = 10
batch_size = 100

# input x - for 28 x 28 pixels = 784, output: 10
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, 10])

W1 = tf.Variable(tf.random_normal([784, 300], stddev=0.03), name=
    ↪ 'W1')
b1 = tf.Variable(tf.random_normal([300]), name='b1')
W2 = tf.Variable(tf.random_normal([300, 10], stddev=0.03), name='
    ↪ W2')
b2 = tf.Variable(tf.random_normal([10]), name='b2')

# calculate the output of the hidden layer
h1 = tf.add(tf.matmul(x,W1), b1)
a1 = tf.nn.relu(h1)

# output layer
# y_ = tf.nn.softmax(tf.add(tf.matmul(a12, W2), b2))
y_ = tf.nn.softmax(tf.add(tf.matmul(a1, W2), b2))

y_clipped = tf.clip_by_value(y_, 1e-10, 0.9999999)
cross_entropy = -tf.reduce_mean(tf.reduce_sum(y * tf.log(
    ↪ y_clipped)
```

```
+ (1 - y) * tf.log(1 - y_clipped), axis=1))

optimiser = tf.train.GradientDescentOptimizer(learning_rate=
    ↪ learning_rate).minimize(cross_entropy)

init_op = tf.global_variables_initializer()

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)
    ↪ )

with tf.Session() as sess:
        # initialise the variables
        sess.run(init_op)
        total_batch = int(len(mnist.train.labels) / batch_size)
        for epoch in range(epochs):
                avg_cost = 0
                for i in range(total_batch):
                        batch_x, batch_y = mnist.train.next_batch(
                            ↪ batch_size=batch_size)
                        _, c = sess.run([optimiser, cross_entropy],
                                feed_dict={x: batch_x, y: batch_y})
                        avg_cost += c / total_batch
                print("Epoch:", (epoch + 1), "cost␣=", "{:.3f}".
                    ↪ format(avg_cost))
        print(sess.run(accuracy, feed_dict={x: mnist.test.images,
            ↪ y: mnist.test.labels}))
```

B ConvNet for CIFAR-10 Image recognition

To classify the images in the MNIST dataset we used the following CNN architecture. The summary after run was: 'loss': 1.4672284, 'global_step': 20000, 'accuracy': 0.4837

- Convolutional Layer #1: Applies 64 3x3 filters, with ReLU activation function
- Pooling Layer #2: Performs max pooling with a 2x2 filter and stride of 2
- Convolutional Layer #1: Applies 64 3x3 filters, with ReLU activation function
- Pooling Layer #2: Performs max pooling with a 2x2 filter and stride of 2
- Dense Layer #1: 512 neurons, with dropout regularization rate of 0.4
- Dense Layer #2: (Logits Layer): 10 neurons, one for each image class.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from keras.datasets import cifar10

import numpy as np
import tensorflow as tf

tf.logging.set_verbosity(tf.logging.INFO)

def cnn_model_fn(features, labels, mode):
        # Reshape X to 4-D tensor: [batch_size, width, height,
            ↪ channels]
        input_layer = tf.reshape(features["x"], [-1, 32, 32, 3])

        # Input Tensor Shape: [batch_size, 32, 32, 3]
        # Output Tensor Shape: [batch_size, 32, 32, 32]
        conv1 = tf.layers.conv2d(
                                inputs=input_layer,
                                filters=32,
                                kernel_size=[3, 3],
                                padding="same",
                                activation=tf.nn.relu)

        # Input Tensor Shape: [batch_size, 32, 32, 32]
        # Output Tensor Shape: [batch_size, 16, 16, 32]
        pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size
            ↪ =[2, 2], strides=2)
```

```
# Input Tensor Shape: [batch_size, 32, 32, 32]
# Output Tensor Shape: [batch_size, 32, 32, 64]
conv2 = tf.layers.conv2d(
                    inputs=pool1,
                    filters=64,
                    kernel_size=[3, 3],
                    padding="same",
                    activation=tf.nn.relu)

# Input Tensor Shape: [batch_size, 16, 16, 64]
# Output Tensor Shape: [batch_size, 8, 8, 64]
pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size
    ↪ =[2, 2], strides=2)

# Flatten tensor into a batch of vectors
# Input Tensor Shape: [batch_size, 8, 8, 64]
# Output Tensor Shape: [batch_size, 8 * 8 * 64]
pool2_flat = tf.reshape(pool2, [-1, 8 * 8 * 64])

# Input Tensor Shape: [batch_size, 8 * 8 * 64]
# Output Tensor Shape: [batch_size, 512]
dense = tf.layers.dense(inputs=pool2_flat, units=512,
    ↪ activation=tf.nn.relu)

dropout = tf.layers.dropout(
       inputs=dense, rate=0.4, training=mode == tf.
           ↪ estimator.ModeKeys.TRAIN)

# Input Tensor Shape: [batch_size, 512]
# Output Tensor Shape: [batch_size, 10]
logits = tf.layers.dense(inputs=dropout, units=10)

predictions = {
       # Generate predictions (for PREDICT and EVAL mode)
       "classes": tf.argmax(input=logits, axis=1),
       "probabilities": tf.nn.softmax(logits, name="
           ↪ softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
       return tf.estimator.EstimatorSpec(mode=mode,
           ↪ predictions=predictions)

loss = tf.losses.sparse_softmax_cross_entropy(labels=
    ↪ labels, logits=logits)
```

```
        # Configure the Training Op (for TRAIN mode)
        if mode == tf.estimator.ModeKeys.TRAIN:
                optimizer = tf.train.GradientDescentOptimizer(
                    ↪ learning_rate=0.001)
                train_op = optimizer.minimize(
                        loss=loss,
                        global_step=tf.train.get_global_step())
                return tf.estimator.EstimatorSpec(mode=mode, loss=
                    ↪ loss, train_op=train_op)

        # Add evaluation metrics (for EVAL mode)
        eval_metric_ops = {
                "accuracy": tf.metrics.accuracy(
                labels=labels, predictions=predictions["classes"])}
        return tf.estimator.EstimatorSpec(
                mode=mode, loss=loss, eval_metric_ops=
                    ↪ eval_metric_ops)


def main():
        (X_train, y_train), (X_test, y_test) = cifar10.load_data()
        print('X_train␣shape:', X_train.shape)
        print(X_train.shape[0], 'train␣samples')
        print(X_test.shape[0], 'test␣samples')

        Y_train = np.asarray(y_train, dtype=np.int32)
        Y_test = np.asarray(y_test, dtype=np.int32)

        X_train = X_train.astype('float32')
        X_test = X_test.astype('float32')
        X_train /= 255
        X_test /= 255

        # Create the Estimator
        mnist_classifier = tf.estimator.Estimator(
        model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model
            ↪ ")

        # Train the model
        train_input_fn = tf.estimator.inputs.numpy_input_fn(
                x={"x": X_train},
                y=Y_train,
                batch_size=100,
                num_epochs=None,
                shuffle=True)
        mnist_classifier.train(
```

```
                input_fn=train_input_fn,
                steps=20000)

        # Evaluate the model and print results
        eval_input_fn = tf.estimator.inputs.numpy_input_fn(
                x={"x": X_test},
                y=Y_test,
                num_epochs=1,
                shuffle=False)
        eval_results = mnist_classifier.evaluate(input_fn=
            ↪ eval_input_fn)
        print(eval_results)

if __name__ == "__main__":
        tf.app.run()
```

References

[1] CS231n: Convolutional Neural Networks for Visual Recognition

[2] Goodfellow et al, Deep Learning, MIT Press, 2016

[3] "Python TensorFlow Tutorial - Build a Neural Network." Adventures in Machine Learning, 11 Aug. 2018, adventuresinmachinelearning.com/python-tensorflow-tutorial/.

Appendices

as