Hands-On Machine Learning: summary

June 1, 2020

Chapter 10

Artificial Neural Networks

Artificial neural networks (ANNs): Machine learning model inspired by networks of biological neurons. They are at the very core of Deep Learning: versatile, powerful and scalable.

10.1 From Biological to Artificial Neurons

Reasons to believe this wave of interest in ANNs is more profound:

- Huge quantity of data available to train & ANNs outperform other ML technologies in large & complex problems
- Tremendous increase in computing power since the 90s + GPU & cloud platforms
- Training algorithms have improved (by small tweaks with huge impact)
- Theoretical limitations have been mostly rare in practice
- ANNs seem to have entered a virtuous cycle of funding & progress

10.1.1 Biological neurons

Individual biological neurons seem to behave in a simple way, but are organized in a network of billions, each connected to thousands of others. Highly complex computations can emerge from their combines efforts. It appears that biological neurons are organized in consecutive layers, especially in the cerebral cortex.

10.1.2 Logical computations with neurons

The first artificial neurons had only binary inputs & one binary output. Each neuron's output is active if more than a certain number of inputs are. A network like this can compute complex logical expressions.

10.1.3 The Perceptron

The *perceptron* is a simple ANN architecture composed by a single layer of threshold logic unit neurons. Their inputs and output are now real numbers; each input connection is associated with a

weight. The TLU computes a weighted sum of its inputs $\left(z = \boldsymbol{x}^{\intercal}\boldsymbol{w}\right)$ and then applies a *step function*. Its output is thus $h_{\boldsymbol{w}}\left(\boldsymbol{x}\right) = \text{step}\left(z\right)$.

A single TLU can be used for simple linear binary classification. The perceptron contains a layer of TLUs. each *fully connected* to all inputs and also containing a *bias* term. Ir can thus perform multi-output binary classification: $h_{\boldsymbol{w},i} = \phi\left(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\right)$, where ϕ is the activation function (step for TLUs), $\boldsymbol{X}$ is the input matrix $(n_{\text{instances}} \times n_{\text{features}})$, $\boldsymbol{W}$ is the weights matrix $(n_{\text{features}} \times n_{\text{neurons}})$, and $\boldsymbol{b}$ is the bias vector (n_{neurons}).

The training is done in a way to reinforce connections that help reduce the error. The perceptron recieves one training instance at a time, and for each wrong output neuron, reinforces weights from inputs that would have contributed to the correct prediction:

$$w_{i,j}^{\text{next}} = w_{i,j} + \underbrace{\eta}_{\text{learning rate}} \underbrace{\left(y_j - \hat{y}_j\right)}_{\text{0 or 1}} x_i$$

If the training instances are *linearly separable*, the algorithm is proven to converge to a (non unique) solution. Contrary to logistic regression, these perceptrons do not output a class probability.

Limitations of perceptrons (such as the XOR problem) can be eliminated by stacking multiple layers

10.1.4 The Multilayer Perceptron and Backpropagation

A MLP is comprised of an input layer, an output layer and several *hidden* layers. Each layer except for the output) is fully connected (layers close to the input are called *lower*).

The training algorithm is known as **backpropagation**: gradient descent with an efficient technique for computing gradients automatically (*autodiff*). In a forward and a backward pass, the algorithm is able to compute the gradient with respect to every model parameter.

Algorithm breakdown

1. Handles one mini-batch at a time, and goes through full training set multiple times (each pass is called an *epoch*).
2. For each instance in the mini-batch, the output is computed in a *forward-pass* (intermediate results from all layers are preserved since they are needed for the backward pass).
3. The output error is measured using the loss function
4. Then it computes each output connection's contribution to the error. This is done analytically using the chain rule.
5. It now calculates the error contributions from the weights on the layer below (also with the chain rule); this is repeated working backwards to the input layer
6. Finally, a gradient step is performed to update all weights

It is important that all weights be initialized randomly, or else the training will fail (a *symmetry breaking* is required).

In order for the algorithm to work, the step function must be smoothed to have a well-defined non-zero derivative, thus allowing GD to make some progress at every step. A first replacement is the sigmoid function $\sigma\left(z\right) = \dfrac{1}{1 + \exp\left(-z\right)}$

Other choices:

- $\tanh(z) = 2\sigma(2z) - 1$; similar to the logistic, but with output in $[-1, 1]$. This tends to make each output mostly centered around 0 at the beginning of training, which helps speed up the convergence.
- ReLU $(z) = \max(0, z)$; derivative jumps at $z = 0$, which can make GD bounce; but works well and is very fast to compute. Its unbounded image helps reduce some issues during GD.

Nonlinearities introduced by activation functions are essential to the complexity of the model. A large enough DNN can theoretically approximate any continuous function.

10.1.5 Regression MLPs

One output neuron per output dimension (*e.g.:* 2 for locating the center of an object in an image; another 2 for a bounding box (height and width)). Usually no activation at the output, except for ReLU/softplus $(z) = \log[1 + \exp(z)]$ to restrict it to positive images; or σ/tanh to bound it.

Training loss is typically MSE, or MAE/Huber (a combination of both) if there are many training outliers. The number of hidden layers is usually $\sim 1 - 5$, and the neurons per layer are $\sim 10 - 100$.

10.1.6 Classification MLPs

For each binary classification, a single output neuron with logistic activation is used, which can be interpreted as the estimated probability of the positive class.

For single output multiclass classification, an output neuron per class is needed, and a softmax activation for the whole layer (which estimates each class' probability)

The training loss function is generally multiclass cross-entropy. The rest of the architecture is broadly the same as with regression.

10.2 Implementing MLPs with Keras

Keras is a high-level Deep learning API with several backends. TensorFlow comes bundled with its own implementation, `tf.keras`, which has many advantages (*e.g.* TF's Data API to load and preprocess data efficiently).

10.2.1 Building an image classifier using the `Sequential` API

We use Fashion MNIST (70k grayscale 28×28 images, 10 classes). Since we will be using GD, we scale the input features.

Creating the model `model = keras.models.Sequential()` is the simplest model, for a single stack of layers connected sequentially
`model.add(keras.layers.Flatten(input_shape = [28,28]))` converts the image to a 1D array. As is the first layer, we must pass the `input_shape` (for a single instance)
`model.add(keras.layers.Dense(#neurons, activation = 'relu'))`

Instead of adding layers by one, it is possible to pass them as a list to the `Sequential` constructor.

10.2.1.1 Inspection

`model.summary()` displays the layers (by name), along with their parameters. `model.layers` returns them as a list, but they can also be called by name as `model.get_layer('dense')`. Its parameters can be retrieved by `weights, biases = layer.get_weights()`. For a custom initialization, when creating the layer we can set `kernel_initializer` or `bias_initializer`.

10.2.1.2 Compiling the model

The method must be called to specify the loss function and optimizer:
`model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'sgd', metrics = ['accuracy'])`. The `metrics` arg corresponds the list of metrics to be computed during training and evalutation. The loss function here is due to exclusive classes and the class given by a single index (this is considered *sparse*). If class were given by a one-hot vector instead, we use `categorical_crossentropy`. To convert sparse to one-hot, `keras.utils.to_categorical()`

10.2.1.3 Training and evaluating

`history = model.fit(X_train, y_train, epochs, validation_data = (X_val,y_val))`
With class imbalance, we can set the `class_weigths` (and even `sample_weight`) arguments in the `fit` method.

The `fit` method returns a `History` object with the training parameters (`history.params`) and a dictrionary `history.history` containing the loss and metrics computed during training. This can be easily put in a dataframe toplot the learning curve (the training curve should be shifted to the left by half an epoch). The `fit` method resumes training from last state if called multiple times.

To increase performance, the first hyperparameter to be tuned should be the learning rate; if that doesn't help, changing the optimizer. After that, the architecture and activation functions. Finally, we can evaluate the model with `model.evaluate(X_test,y_test)`

10.2.2 Building complex models using the `Functional` API

To build models with more complex topologies, multiple inputs or outputs, keras offers the `Functional` API. Each layer must be defined as a separate object, specifying its input
`input_ = keras.layers.Input(shape=[...])` an `Input` object must be defined (even more than one if necessary)
`hidden = Dense(neurons, activation = 'relu')(input_)` the input to this layer is passed by calling it as a function
`concat = keras.layers.Concatenate()([input_, hidden])` layer which concatenates inputs
`model = keras.Model(inputs = [input_], outputs = [output])` we create the model, specifying input(s) and output(s). The model can then be compiled and trained.
With multiple inputs, the `fit` method must be supplied with a tuple (or dictionary) of input matrices. Multiple outputs may be needed in many cases:

- The task may demand it; as in object location and identification (regression for a bounding box, classification to identify).
- Multiple independent tasks on the same data; the network can learn features in the data that are useful across tasks.

- It can be used as a regularization technique, for example, by adding an auxiliary output from middle layers, to ensure the underlying part learn something useful.
- Each output needs its own loss function. These will be added to obtain the total loss used in training, The losses will be passed as a list to the `fit` method, along with a list of `loss_weights` to perform a weighted sum instead. As with inputs, a tuple of targets must be supplied.

10.2.3 Using the `Subclassing` API to build dynamic models

To create models with even greater flexibility (*e.g.* loops, varying shapes, dynamical behaviours), we may use the subclassing API:

```
class CustomModel(keras.Model):          #inherit from base class

    def __init__(self, ..., **kwargs):
        super().__init__(**kwargs)    #handles standard args(e.g. name)
        # all layers should be created in the constructor
        self.hidden = ...

    def call(self, inputs):
    # all computations performed here (input need not be created,
    # just passed to the call method)
        ...
    return outputs
```

The extra flexibility's cost is that the model architecture is hidden in `call`, so keras cannot easily inspect it, save it or clone it.
Keras models can be used like regular layers to combine them.

10.2.4 Saving and restoring a model

With the `Sequential` or `Functional` API, a model can be saved to HDF5 using `model.save(fname)`. This saves the architecture, parameters for every layer and optimizer.
It can be loaded using `keras.models.load_model`
This won't work with subclassing, but model parameters can be saved with `save_weights` (also loaded with `load_weights`).

10.2.5 Using Callbacks

The `fit` method accepts a list of objects in the `callbacks` argument, which will be called at the start and end of training, or each epoch or batch.

10.2.5.1 Model checkpoint

`checkpoint_cb = keras.callbacks.ModelCheckpoint(fname, save_best_only=True)` the callback saves the model at regular intervals during training (by default at the end of each epoch).
`save_best_only=True` will only make the checkpoint when the performance on the validation set is the best so far.

10.2.5.2 Early Stopping

`early_stopping_cb = keras.callbacks.EarlyStopping(patience)` will interrupt training when it measures no progress on validation during a number of epochs (given by `patience`). The best model can be restored enabling `restore_best_weights=True`.

10.2.5.3 Custom Callbacks

We can create callbacks by inheriting from the base class

```
class CustomCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print(logs['val_loss'])
```

options include `on_train_end`, `on_batch_end`, etc. Also ones only with evaluate, predict.

10.2.6 Using TensorBoard for visualization

TensorBoard is an interactive visualization tool to view learning curves during training, compare them between runs, visualize the computation graph, training statistics & multidimensional data projected to 3D (and more).
To use it, the model must generate binary *event files*, The TB server monitors a log directory; it is useful to have a different subdir for each run. The TB callback must be used to generate the files:
`tensorboard_cb = keras.callback.TensorBoard(run_logdir)`
The server can be called with
`%load_ext tensorboard`
`%tensorboard -logdir=./my_logs -port=6006`
TF also has a low-level API to manually write logs (`tf.summary`)

10.3 Fine-tuning neural network hyperparameters

The main drawback of the flexibility of ANNs is the number of hyperparameters to tweak. There are some options to find the best combination.
One option is to try many combinations using grid or random search and measuring on the validation set (or using k-fold cross validation). To work with the scikit-learn API, we use a wrapper to treat models as sklearn estimators:
`keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)` where `build_model` is function that takes a set og hyperparameters and returns a compiled model.
Random search works well for fairly simple problems, but when training is slow, it will only explore a tiny portion of hyperparameter space.

Other approaches "zoom in" when a region in the space turns out to be good; there are several libraries for this purpose

10.3.1 Number of hidden layers

For simple problems, a single hidden layer may be enough; but for complex problems, deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially fewer neurons.

Deep networks can model the hierarchical structure of data: lower layers encode low-level features, and higher layers combine them to model high-level, more abstract structures (*e.g.* from line segments to faces). This can also help the model generalize to other datasets. For example, we may reuse the lower layers from a model already trained to recognize faces to train a new model to recognize hairstyles

Chapter 11

Training Deep Neural Networks

Training a DNN (10+ layers), each with 100s of neurons and 100000s of connections can incur many problems:

- *Vanishing/exploding gradients* problem, which make lower layers very hard to train.
- Not enough training data for such a large network, or too costly to label.
- Training may be extremely slow
- A model with millions of parameters would severely risk overfitting, if there are not enough training instances

11.1 The Vanishing/exploding gradients problems

BP works by propagating the error gradient from the output to the input layer. However, gradients often get smaller as the algorithm progressed to the lower layers: thus the GD update is not significant and training cannot converge to a good solution. Likewise, the gradients can grow bigger and the algorithm diverges.
This behaviour was addressed in [Glorot and Bengio, 2010]. Using a sigmoid activation and $\mathcal{N}(0,1)$ weight initialization causes weight variance to grow with layer depth. causing the activations to saturate. This implies the gradient is close to 0 dilutes further with BP.

11.1.1 Glorot and He initialization

In [Glorot and Bengio, 2010], they argue that the variance of the outputs of each layer needs to be equal to the variance of its inputs, and the same condition is needed for the variance of the gradients before and after backflowing through the layer, for the signal to flow properly in both directions. This cannot be guaranteed when #inputs (*fan-in*) is not equal to the #outputs (*fan-out*), but a good compromise can be made

11.1.1.1 Glorot/Xavier initialization (when using logistic activation)

Weights to be sampled from a normal distribution $\mathcal{N}(0,\sigma^2)$, with $\sigma^2 = \dfrac{1}{\text{fan}_{\text{avg}}}$; or from a uniform distribution $U[-r,r]$, with $r = \sqrt{\dfrac{3}{\text{fan}_{\text{avg}}}}$

11.1.1.2 He initialization (when using ReLU & variants)

Same as Glorot, but using $\sigma^2 = \frac{2}{\text{fan}_{\text{in}}}$, or uniform using $r = \sqrt{3\sigma^2}$

11.1.1.3 LeCun initialization (when using SELU):

Only normal distribution with $\sigma^2 = \frac{1}{\text{fan}_{\text{in}}}$

By default. keras uses uniform Glorot. When creating a layer, the initializer can be set with `kernel_initializ` for instance. A custom initialization can be done with (e.g. He average)

```
he_avg_init=keras.initializers.VarianceScaling(scale=2.,
                                               mode='fan_avg',
                                               distribution='uniform')
```

and this object must be passed to the `kernel_initializer`.

11.1.2 Nonsaturating activation functions

Due to the lack of saturation, ReLUs work better than sigmoid in DNNs. However, during training some neurons may "die", outputting nothing than 0, anulling their gradients too. This may be solved using a *leaky* ReLU, with a small slope α for $z < 0$. This ensures the unitrs never "die".

[Xu et al., 2015] concluded leaky ReLUs always outperform strict ones, even working better with $\alpha \sim 0.2$ instead of ~ 0.01. The paper also evaluated the *randomized leaky* ReLU (setting a random α during training and fixing to an average during test), which performed fairly well and seemed to act as a regularizer.
The paper also evaluated the PReLU, where α is an hyperparameter. It is reported to outperform ReLUs on large image datasets, but runs the risk of overfitting on smaller ones.

[Clevert et al., 2015] proposed the *exponential linear unit* (ELU) that outperformed ReLU in both training time and test score. ELU:

- Takes on negative values with $z < 0$, which allows the unit to have an average output closer to 0, and helps with vanishing gradients. The asympote negative value is an hyperparameter.
- It has nonzero gradient for $z < 0$, which helps avoid dead neurons.
- With the hyperparameter α it is smooth everywhere, helps speed up GD

The main drawback is the slower computation than ReLU; but its faster convergence rate compensates for that during training (not so during evaluation).

[Klambauer et al., 2017] introduced the scaled ELU (SELU). The authors showed that a nn built only from a stack of dense layers, using only SELU will *self-normalize*: the output of each layer will tend to preserve a mean of 0 and a variance of 1 during training, which solves both gradient problems. SELU often outperforms other activation functions, but there are a few conditions for self-normalization:

1. Input features must be standarized (mean 0, std 1)
2. Every hidden layer initialized with LeCun normal

3. Network architecture must be sequential. In RNN or other architectures, self-normalization cannot be guaranteed.

The paper only guarantees self-normalization if all layers are dense. but it has been noted that SELU can improve performance in CNNs.

In general, SELU > ELU > leaky ReLU > tanh > sigmoid. When caring about runtime, leaky ReLU may be preferred, even with default $\alpha = 0.3$ (keras). ELU may perform better than SELU if self-normalization is not possible. RReLU may be used when overfitting or PReLU with a huge training set.

In keras, leaky ReLU and PReLU must be added as layers:
`keras.layers.LeakyReLU(alpha)` or `PReLU()`. SELU can be set as
`Dense(units, activation='selu', kernel_initializer='lecun_normal')`

11.1.3 Batch normalization

The former strategies can reduce the danger of vanishing/exploding gradients at the beginning of training, there is no guarantee they won't come back during training.

[Ioffe and Szegedy, 2015] proposed **batch normalization** to address these problems. The technique adds a new operation as a layer, to be used before or after the activation on each hidden layer. Each input feature is zero-centered and normalized, then re-scaled and centered using two learnable parameters (per input per layer). The model can thus learn the optimal scale and mean of each of the layer's inputs.

The algorithm estimates the input's mean and std on each mini-batch:

$$
\begin{aligned}
1\ \boldsymbol{\mu}_B &= \frac{1}{m_B}\sum_{i=1}^{m_b}\boldsymbol{x}^{(i)} \\
2\ \boldsymbol{\sigma}_B^2 &= \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\boldsymbol{x}^{(i)}-\boldsymbol{\mu}_B\right)^2 \\
3\ \hat{\boldsymbol{x}}^{(i)} &= \frac{\boldsymbol{x}^{(i)}-\boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2+\varepsilon}} \\
4\ \boldsymbol{z}^{(i)} &= \boldsymbol{\gamma}\otimes\boldsymbol{x}^{(i)}+\boldsymbol{\beta}
\end{aligned}
$$

m_B is the mini-batch size, ε is a smoothing term $\sim 10^{-5}$, $\boldsymbol{z}^{(i)}$ is the BN output and $\boldsymbol{\gamma}$ $\boldsymbol{\beta}$ are the output scale and shift parameters.

$\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ are estimated for each batch furing training. At test time, an estimation for the whole training set (accumulated during training) is used. Thus, four parameter vectors are learned: $\boldsymbol{\gamma},\ \boldsymbol{\beta},\ \boldsymbol{\mu},\ \boldsymbol{\sigma}^2$ (the latter two used only after training).

BN also acts as a regularizer, reducing the need for other techniques. However, it does add some complexity to the model and has longer runtime. Though after training it is possible to merge the BN with the previous layer (by combining the parameters into new weights and biases).

Training will be slower, as each epoch takes much mote time; but this is counterbalanced by faster convergence, sot it will take fewer epochs to reach the same performance.

11.1.3.1 Implementation in keras

BN can be added as a layer, before or after each hidden layer's activation function (as `keras.layers.BatchNormaliztion()`).

There is some debate whether the layer should be added before or after the activation. To add them before, the activation must be unspecified on the hidden layers and added as a separate layer. This allows us to remove bias from those layers with `use_bias=False`, as BN already contains a shift.

The layer contains a few hyperparameters; namely the **momentum (p)**, used when updating the exponential moving averages (for μ and σ)

$$\hat{\boldsymbol{v}} \leftarrow \hat{\boldsymbol{v}} \times p + \boldsymbol{v} \times (1 - p)$$

where $\hat{\boldsymbol{v}}$ is the running average and $\boldsymbol{v}$ is the new value. p is usually very close to 1 (0.9 or 0.99) (closer for larger datasets).

Another hyperparameter is `axis`, which indicates which axis to normalize. Default is -1 (last axis), which works when input is 2D `[batch_size,features]`. Care should be taken when feeding unflattened 2D images, for instance.

The different beahaviour when training and predicting is given by the `training` arg in the `call` method. It is set to 1 when calling `fit`.

BN is near ubiquitous now, but [Zhang et al., 2019] managed to train a very deep (10k layers) net without BN by using a *fixed-update* weight initialization scheme.

11.1.4 Gradient clipping

Another technique to prevent exploding gradient is to limit the gradient value below a threshold during BP. This is most often used on RNNs, where BN is tricky, With keras, `keras.optimizers.SGD(clipvalue=1.0)` each gradient component will be limited to ± 1.0 (which may change the direction). To scale by the ℓ^2norm and preserve the direction, `clipnorm` may be used instead.

11.2 Reusing pretrained layers

Instead of training a large DNN from scratch, it is usually a better idea to find an existing one which accomplishes a similar task and reuse the lower layers (***transfer learning***). This will speed up training and require significantly less training data.

Transfer learning will work best when the inputs have similar low-level features. The upper hidden layers of the original model are less likely to be useful, as the high-level features may differ significantly between tasks. The more similar the tasks are, the more layers will be reusable.

A first attemts would be to freeze (make their weights non trainable) all reused layers and train & evaluate the model. Then, unfreezing layers from the top, retraining and checking if performance improves. The more data available, the more layers we can unfreeze, It is useful to reduce learning rate when unfreezing, as to avoid wrecking the fine-tuned weights.

11.2.1 Transfer learning with keras

Cloning a model to reuse:

```
model_A_clone = keras.models.clone_model(model_A)
```

`model_A_clone.set_weights(model_A.get_weights())` (weights must be set manually). Reusing all but the last layer:
`model_B = Sequential(model_A.layers[:-1])`. Freezing layers:
`for layer in model_B.layers: layer.trainable=False`
Model must be always compiled after freezing/unfreezing.

Transfer learning does not work very well with small dense networks, presumably because they learn few, very specific patterns. TL works best with deep CNN, which tend to learn more general feature detectors.

11.2.2 Unsupervised pretraining

When tackling a complex task with few training data and no pretrained similar model, we may be able to perform **unsupervised pretraining:** With a large amount of unlabeled training data, an autoencoder or GAN can be trained, and its lower layers reused for a supervised model (using an output layer on top), fine-tuned with available data.

11.2.3 Pretraining on an auxiliary task

Also with little training data, another option is to first train a nn on an auxiliary task for which training data can be easily obtained or generated, and then reuse that model's lower layers for the actual task. This is done in NLP by training a model to predict missing words ins a large corpus (*i.e.* word2vec).

Self-supervised learning consists on automatically generating labels from the data itself, thus counted as unsupervised.

11.3 Faster optimizers

A further way to speed up training is by a faster optimizer than SGD.

11.3.1 Momentum optimization

Regular GD updates by $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, responding only to the local; when that gradient is tiny, it goes slowly. Momentum optimization updates weights with a *momentum* vector $\boldsymbol{m}$, and uses the gradient for acceleration. It introduces the hyperparameter β (the *momentum*) as a friction, to prevent the momentum vector from growing too large (usually $\beta \sim 0.9$). The whole technique allows the optimizer to speed up and also escape plateaus much faster. It is implemented in keras by
`optimizer=keras.optimizers.SGD(r=0.001,momentum=0.9)`
Due to the momentum, the optimizer may overshoot a bit and oscillate by the minimum. The friction helps get rid of them and converge.

11.3.2 Nesterov accelerated gradient

A small variant almost always faster: to compute the gradient not at current position $\boldsymbol{\theta}$ but slightly ahead $\boldsymbol{\theta} + \beta \boldsymbol{m}$:

$$1\ \boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta \boldsymbol{m})$$
$$2\ \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$$

This small tweak works because in general the momentum vector will be pointing on the right direction, so it will be slightly more accurate to measure the gradient a bit farther in that direction. It can be enabled by passing `nesterov=True.` to SGD constructor.

11.3.3 AdaGrad

As with the elongated bowl cost function, the local gradient may not point directly at the global minimum. This algorithm corrects its direction earlier by scaling down the gradient along the steepest dimensions.

$$
\begin{aligned}
&1\ \boldsymbol{s} \leftarrow \boldsymbol{s} + \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
&2\ \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \varepsilon}
\end{aligned}
$$

The algorithm decays the learning rate, but faster for dimensions with steeper slopes ***(adaptive learning rate).*** AdaGrad works well for simple quadratic problems, but often stops too early when training nn (the η gets scaled down too much too soon).

11.3.4 RMSProp

This algorithm fixes the problem ins AdaGrad by only accumulating the gradients from recent iterations (by using and exponential running average).

$$
\begin{aligned}
&1\ \boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1-\beta)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
&2\ \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \varepsilon}
\end{aligned}
$$

This always performs much better than AdaGrad. In keras,
`keras.optimizers.RMSProp(lr, rho)` (`rho` is β here).

11.3.5 Adam and Nadam optimization

Adam (*adaptive moment estimation*) combines the ideas of momentum and RMSProp. It tracks and exponential running average of past gradients and their squares; an estimation of their first and second moments.

$$
\begin{aligned}
&1\ \boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1-\beta_1)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
&2\ \boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1-\beta_2)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\
&3\ \hat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^t} \\
&4\ \hat{\boldsymbol{s}} \leftarrow \frac{s}{1-\beta_2^t} \\
&5\ \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \boldsymbol{m} \oslash \sqrt{\boldsymbol{s} + \varepsilon}
\end{aligned}
$$

3&4 help boost $\boldsymbol{m}$ & $\boldsymbol{s}$ at the beginning of training, since both are initialized to 0 and will be biased

towards it. In keras,
`keras.optimizers.Adam(lr,beta_1,beta_2)`. As an adaptive learning rate, it requires less tuning of the hyperparameter η.

Two variations of Adam:

- **AdaMax:** Adam scales down the parameter updates by the ℓ^2 norm (scaled) between each component's current and past gradients. AdaMax uses instead the ℓ^∞ norm; *i.e.* replaces step **2** with $\boldsymbol{s} \leftarrow \max\left(\beta_2 \boldsymbol{s}, \boldsymbol{\nabla}_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}\right)\right)$ (component-wise) and drops step **4**. This can make the algorithm more stable but it really depends on the dataset.

- **Nadam:** Adam + Nesterov; will often converge slightly faster. [Dozat, 2016] finds that Nadam generally outperforms Adam, but is sometimes outperformed by RMSProp.

Adaptive optimization methods often converge fast to a good solution, but [Wilson et al., 2017] showed they can lead to solutions that generalize poorly on some datasets.

Bibliography

[Clevert et al., 2015] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus).

[Dozat, 2016] Dozat, T. (2016). Incorporating nesterov momentum into adam.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[Klambauer et al., 2017] Klambauer, G., Unterthiner, T., Mayr, A., and Hochreiter, S. (2017). Self-normalizing neural networks.

[Wilson et al., 2017] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning.

[Xu et al., 2015] Xu, B., Wang, N., Chen, T., and Li, M. (2015). Empirical evaluation of rectified activations in convolutional network.

[Zhang et al., 2019] Zhang, H., Dauphin, Y. N., and Ma, T. (2019). Fixup initialization: Residual learning without normalization.