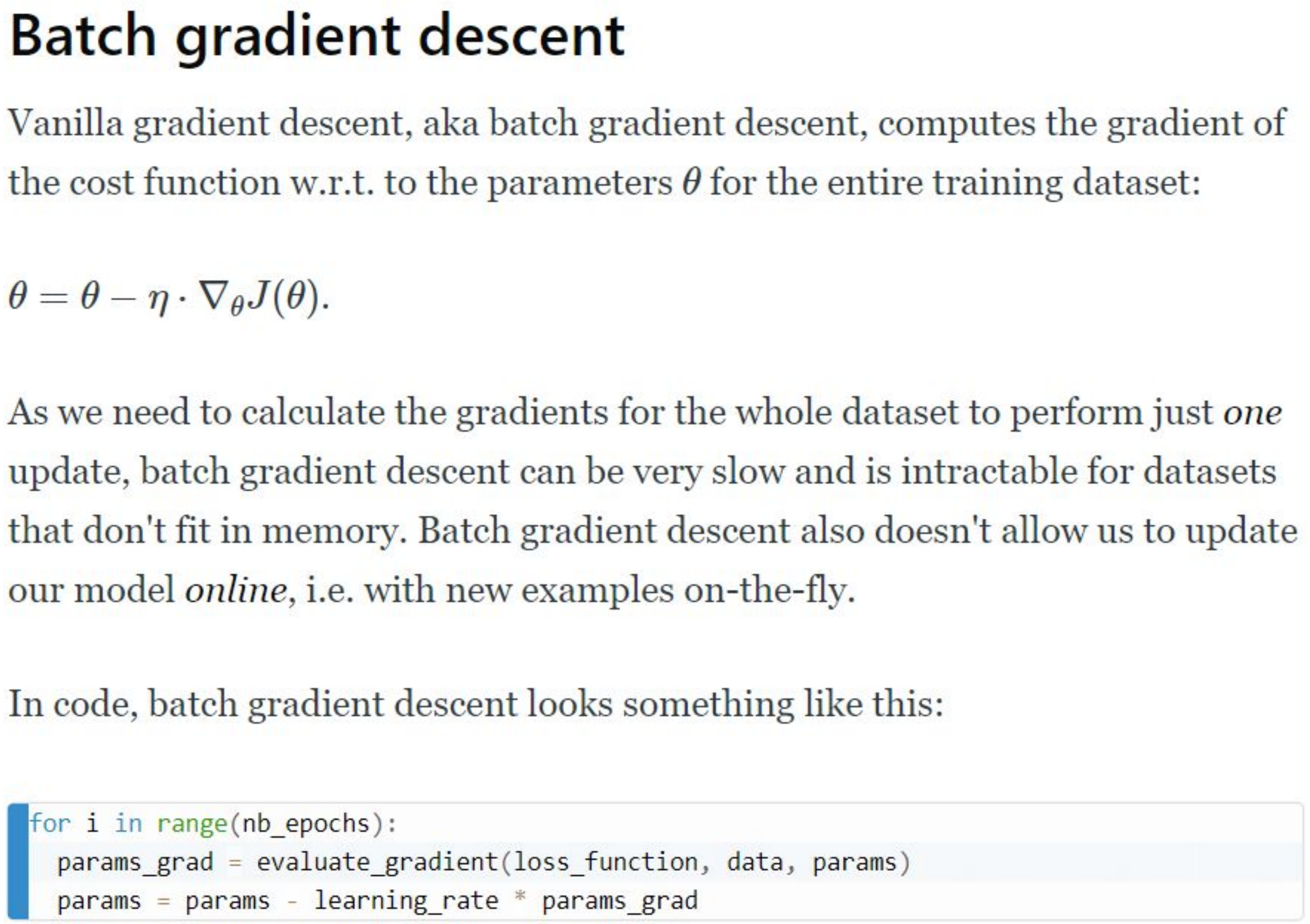
Deep Learning Lesson 5

Lecturer: Paolo Russo
Dip. Ingegneria Informatica, Automatica e Gestionale, Roma

Credits: solvers part taken from https://ruder.io/optimizing-gradient-descent/

SAPIENZA
UNIVERSITÀ DI ROMA

Solvers



Batch gradient descent

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

As we need to calculate the gradients for the whole dataset to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Batch gradient descent also doesn't allow us to update our model *online*, i.e. with new examples on-the-fly.

In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

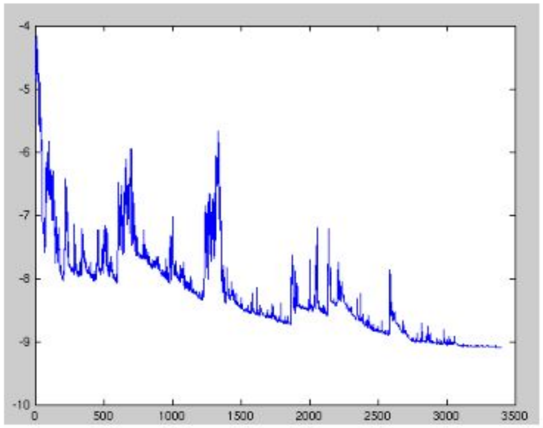
Solvers

Stochastic gradient descent

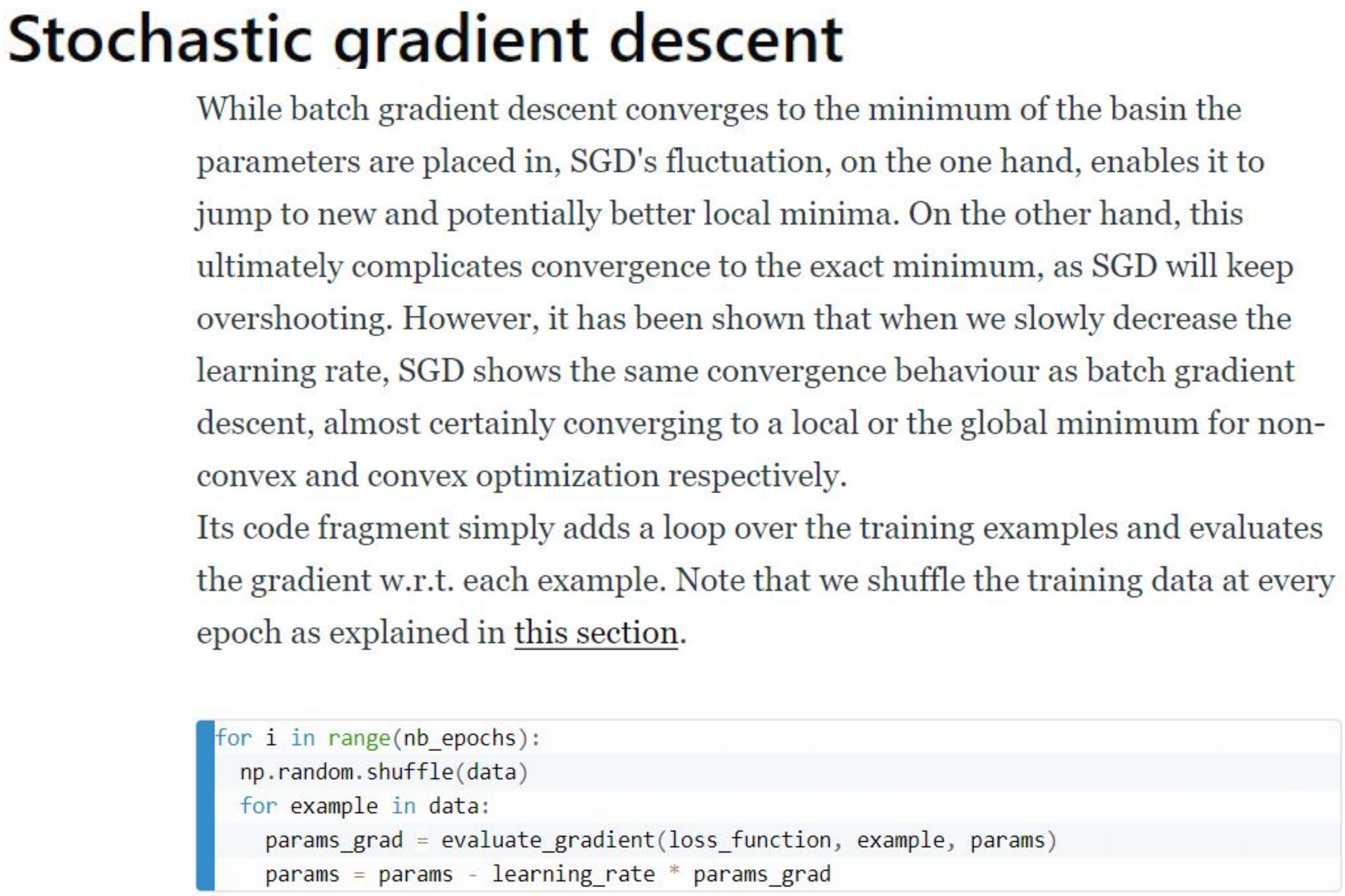
Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Image 1.



Solvers

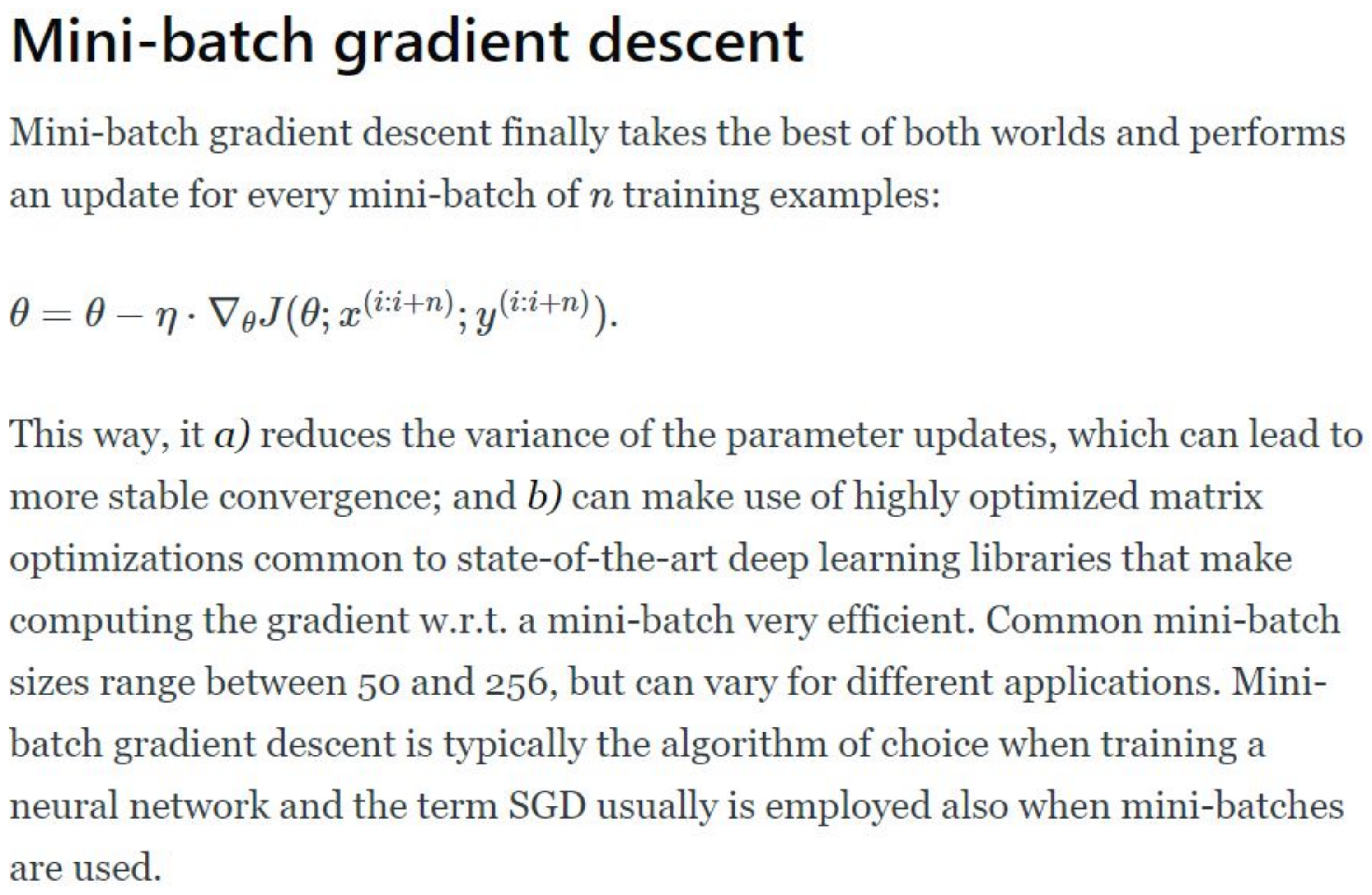


Stochastic gradient descent

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.
Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example. Note that we shuffle the training data at every epoch as explained in this section.

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Solvers



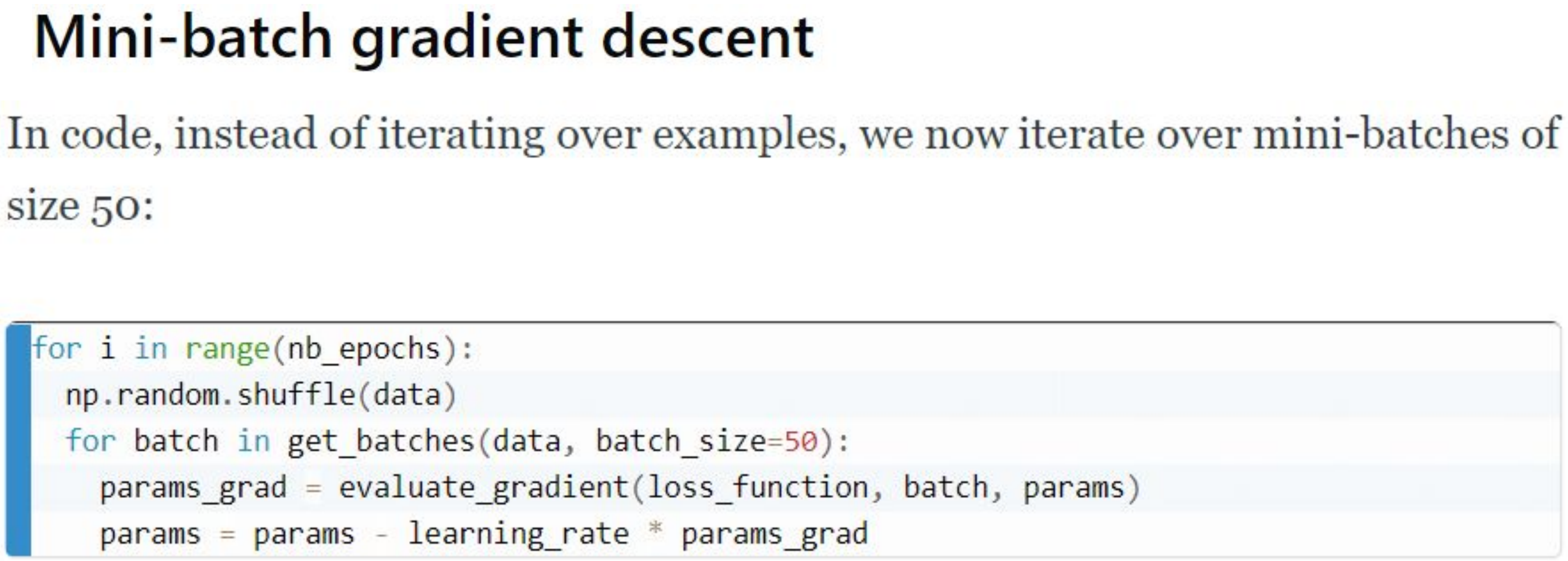
Mini-batch gradient descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

This way, it *a)* reduces the variance of the parameter updates, which can lead to more stable convergence; and *b)* can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

Solvers



Mini-batch gradient descent

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Solvers - SGD (mini-batch) challenges

- Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- Learning rate schedules [1] try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics.

Solvers - SGD (mini-batch) challenges

- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. The difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

Solvers

Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another [4], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Image 2.

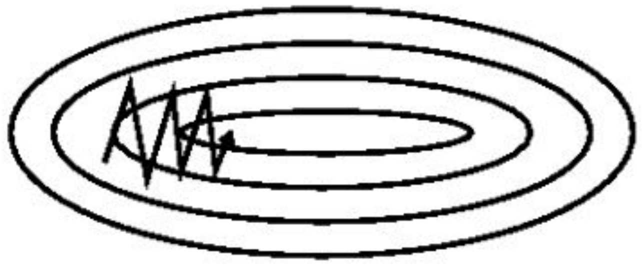


Image 2: SGD without momentum

Solvers

Momentum

Momentum [5] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Image 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Some implementations exchange the signs in the equations. The momentum term γ is usually set to 0.9 or a similar value.

Solvers

Momentum

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way. The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

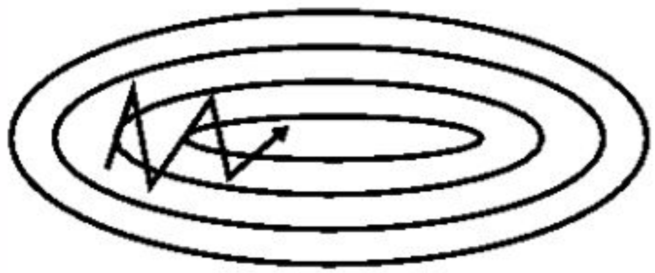
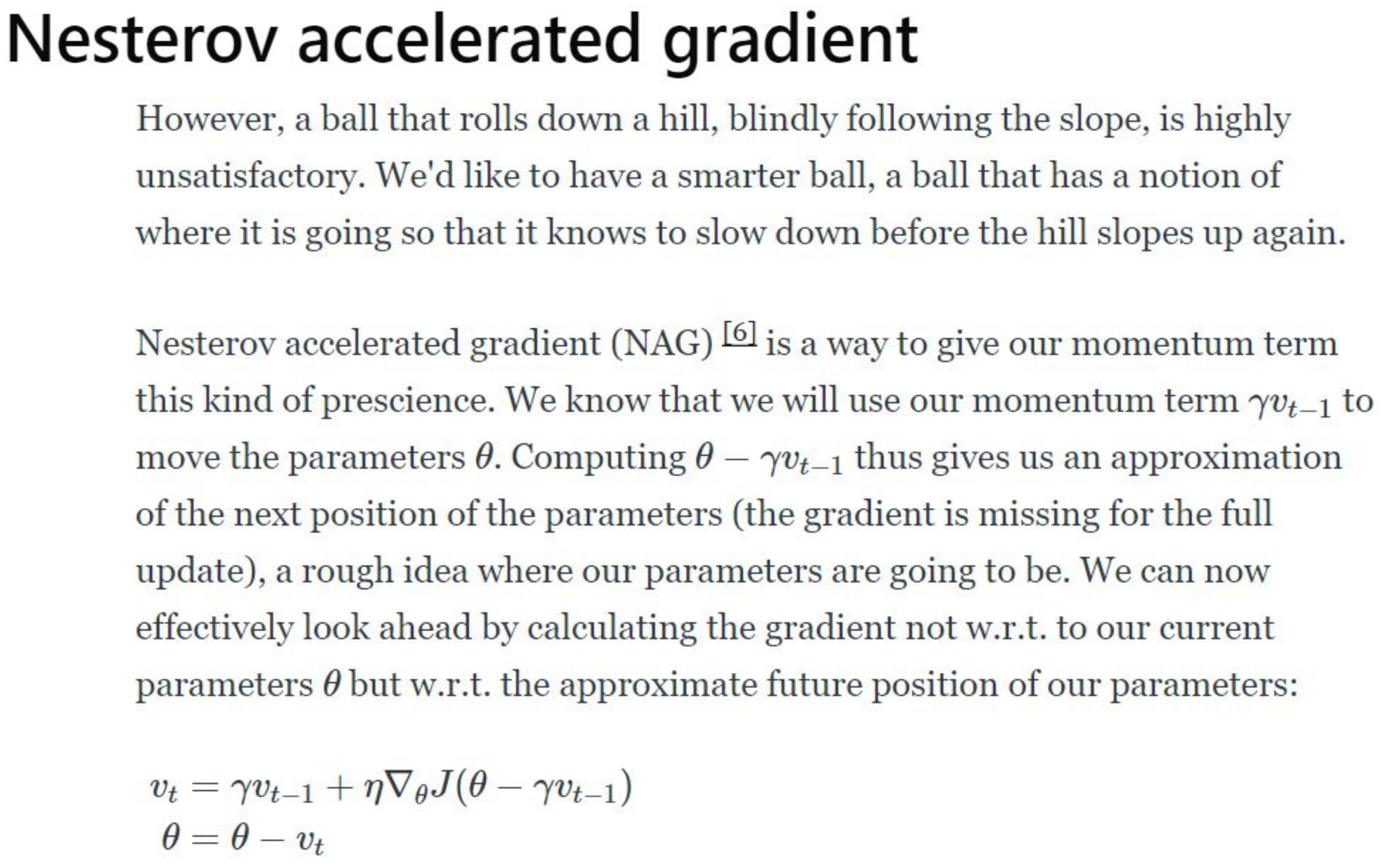


Image 3: SGD with momentum

Solvers



Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

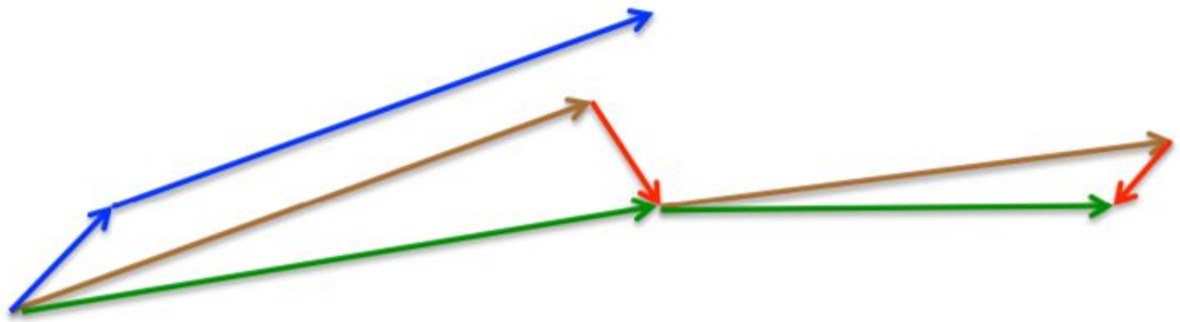
Nesterov accelerated gradient (NAG) [6] is a way to give our momentum term this kind of prescience. We know that we will use our momentum term γv_{t-1} to move the parameters θ. Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$
\begin{aligned}
v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\
\theta &= \theta - v_t
\end{aligned}
$$

Solvers

A picture of the Nesterov method

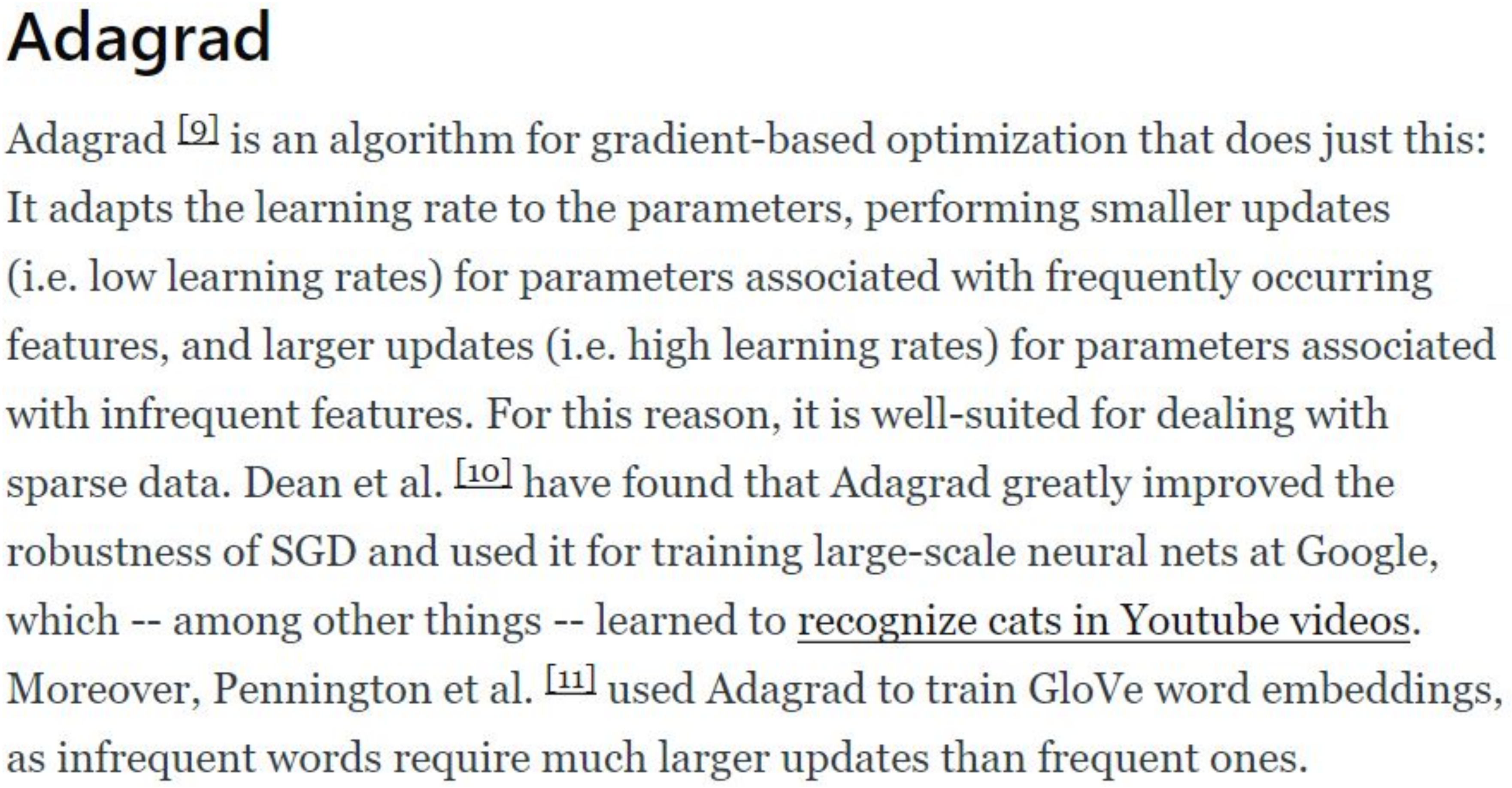
- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

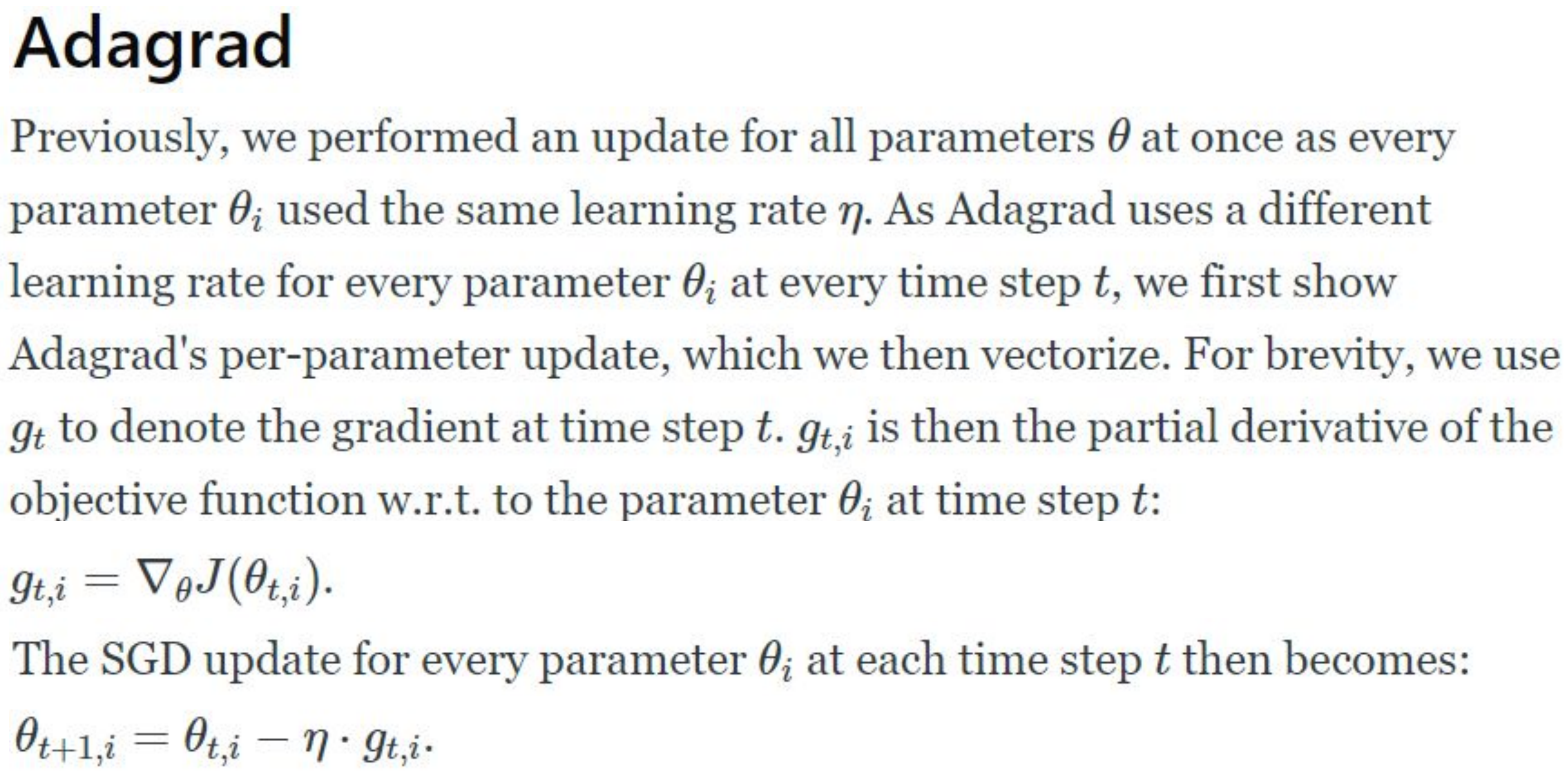
Solvers - adaptive methods



Adagrad

Adagrad [9] is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data. Dean et al. [10] have found that Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google, which -- among other things -- learned to recognize cats in Youtube videos. Moreover, Pennington et al. [11] used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Solvers - adaptive methods



Adagrad

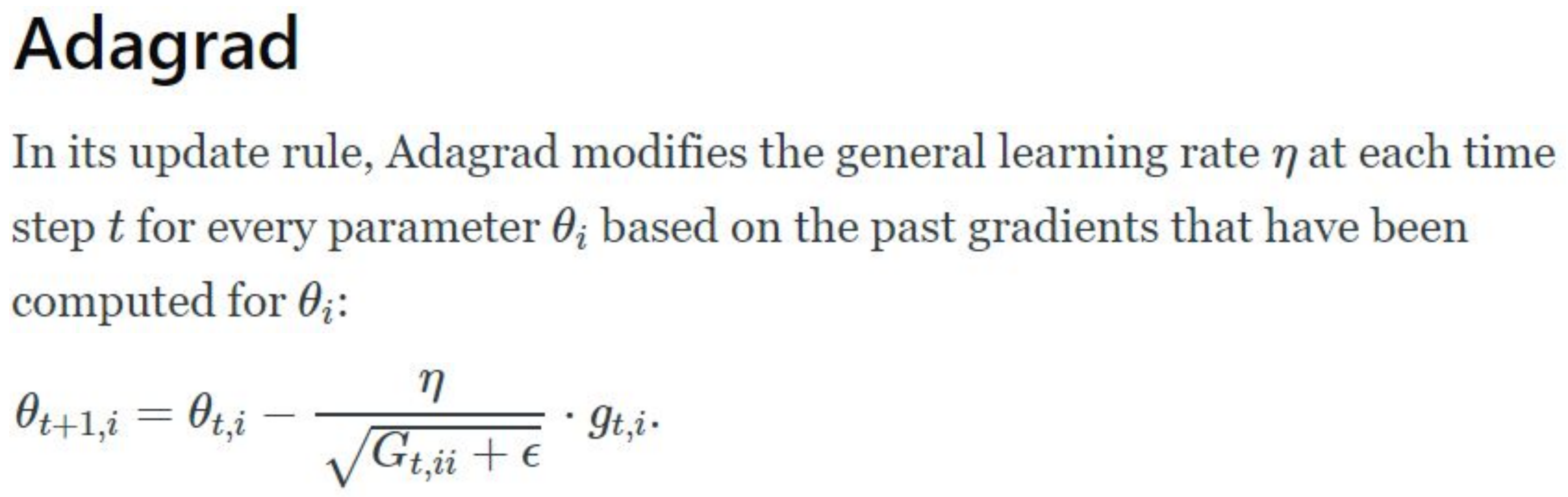
Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we use g_t to denote the gradient at time step t. $g_{t,i}$ is then the partial derivative of the objective function w.r.t. to the parameter θ_i at time step t:

$g_{t,i} = \nabla_\theta J(\theta_{t,i})$.

The SGD update for every parameter θ_i at each time step t then becomes:

$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$.

Solvers - adaptive methods



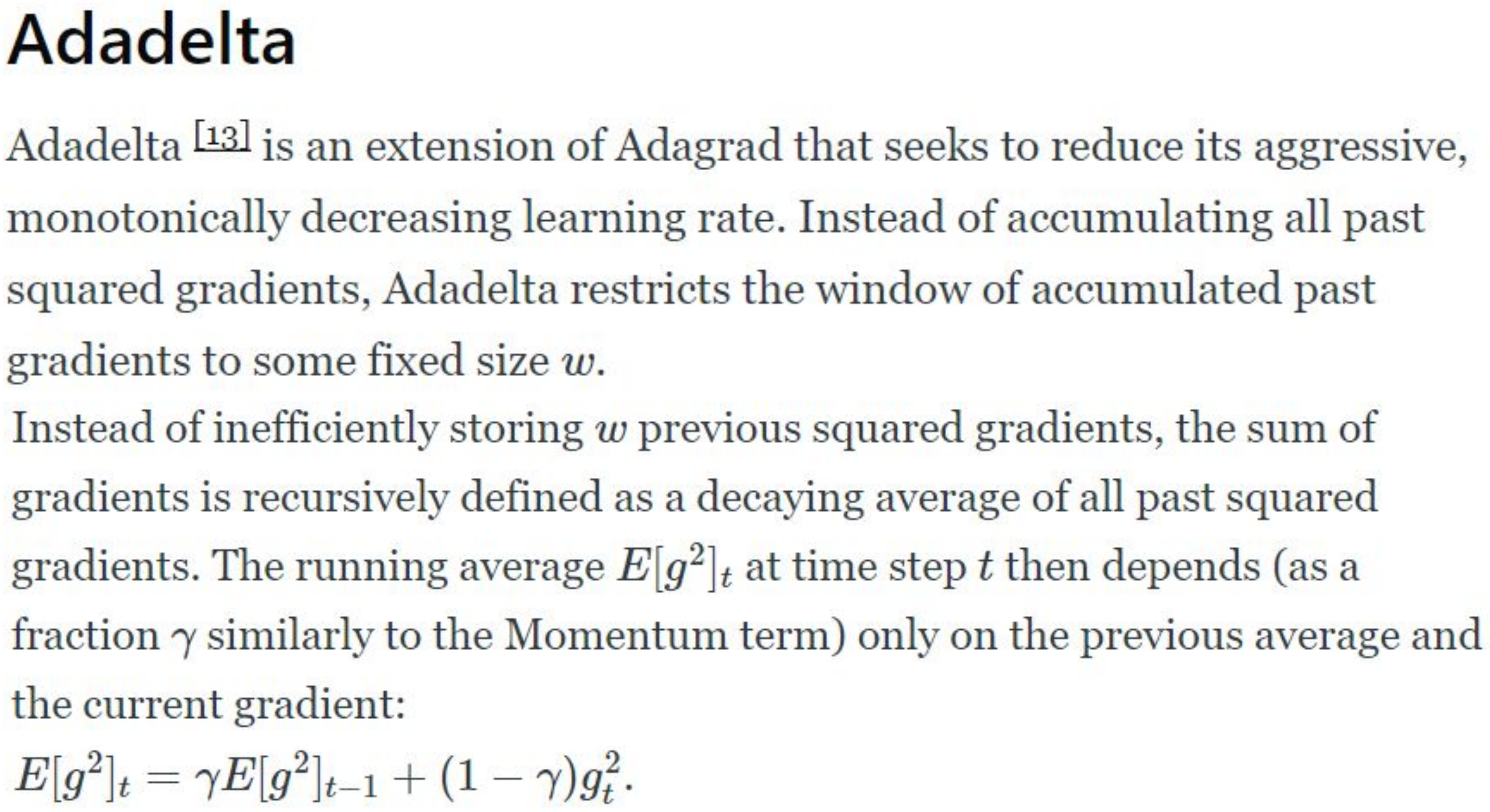
Adagrad

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

- Main strength: it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.
- Main weakness: the accumulated sum of squared gradients keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

Solvers - adaptive methods



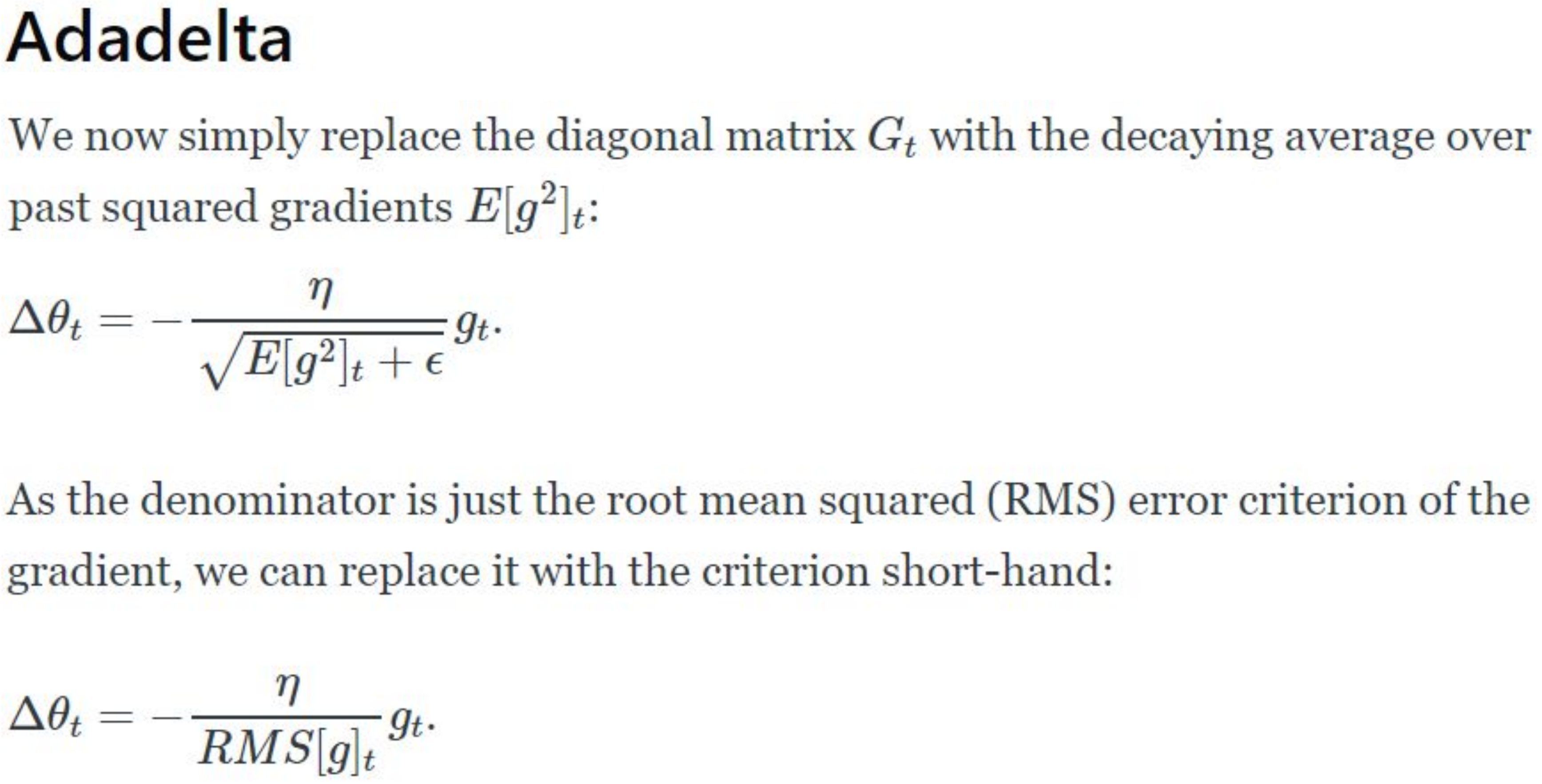
Adadelta

Adadelta [13] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2.$$

Solvers - adaptive methods



Adadelta

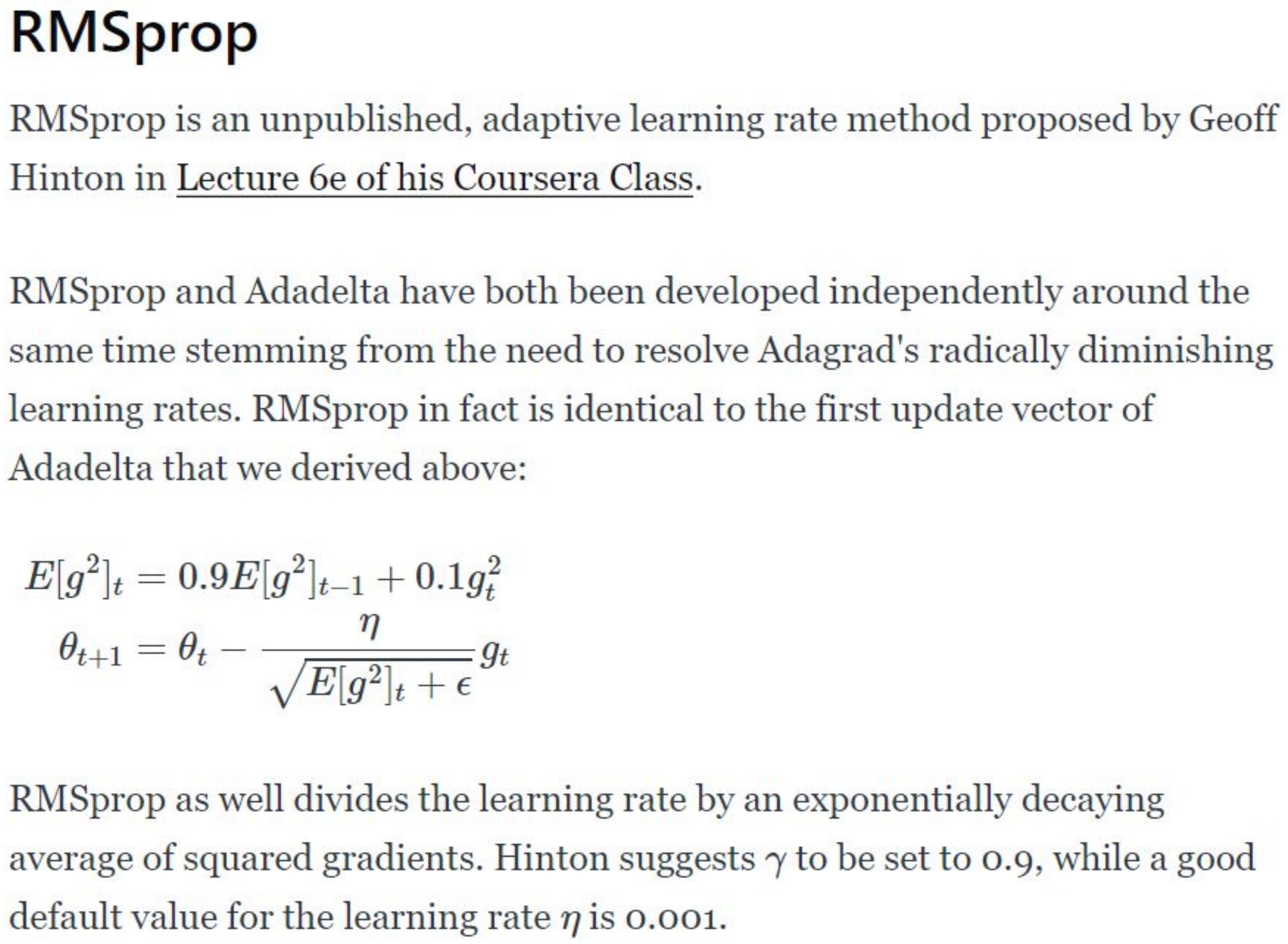
We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

Solvers - adaptive methods



RMSprop

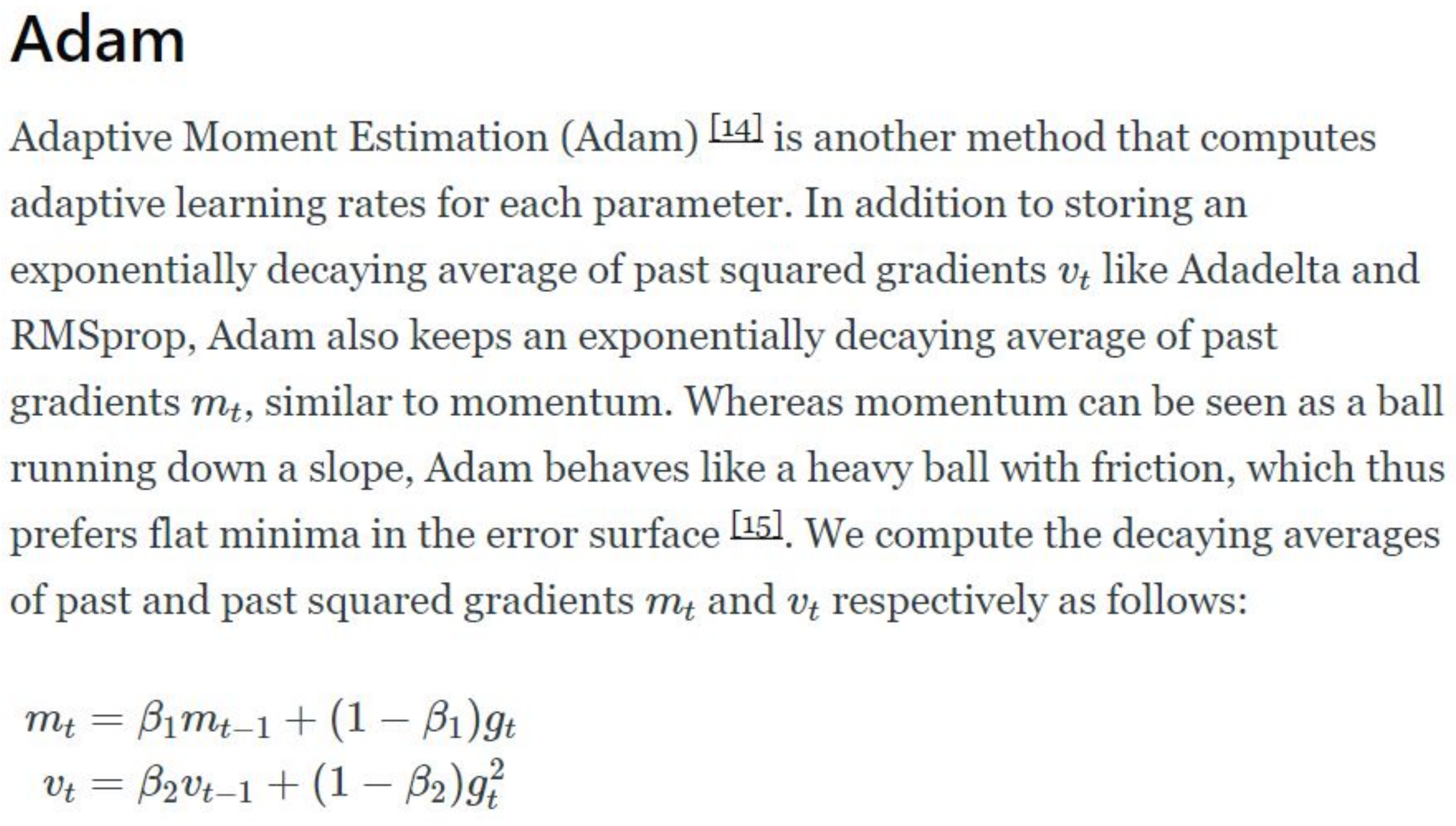
RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$
\begin{aligned}
E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t
\end{aligned}
$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

Solvers - adaptive methods



Adam

Adaptive Moment Estimation (Adam) [14] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface [15]. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

Solvers - adaptive methods

Adam

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

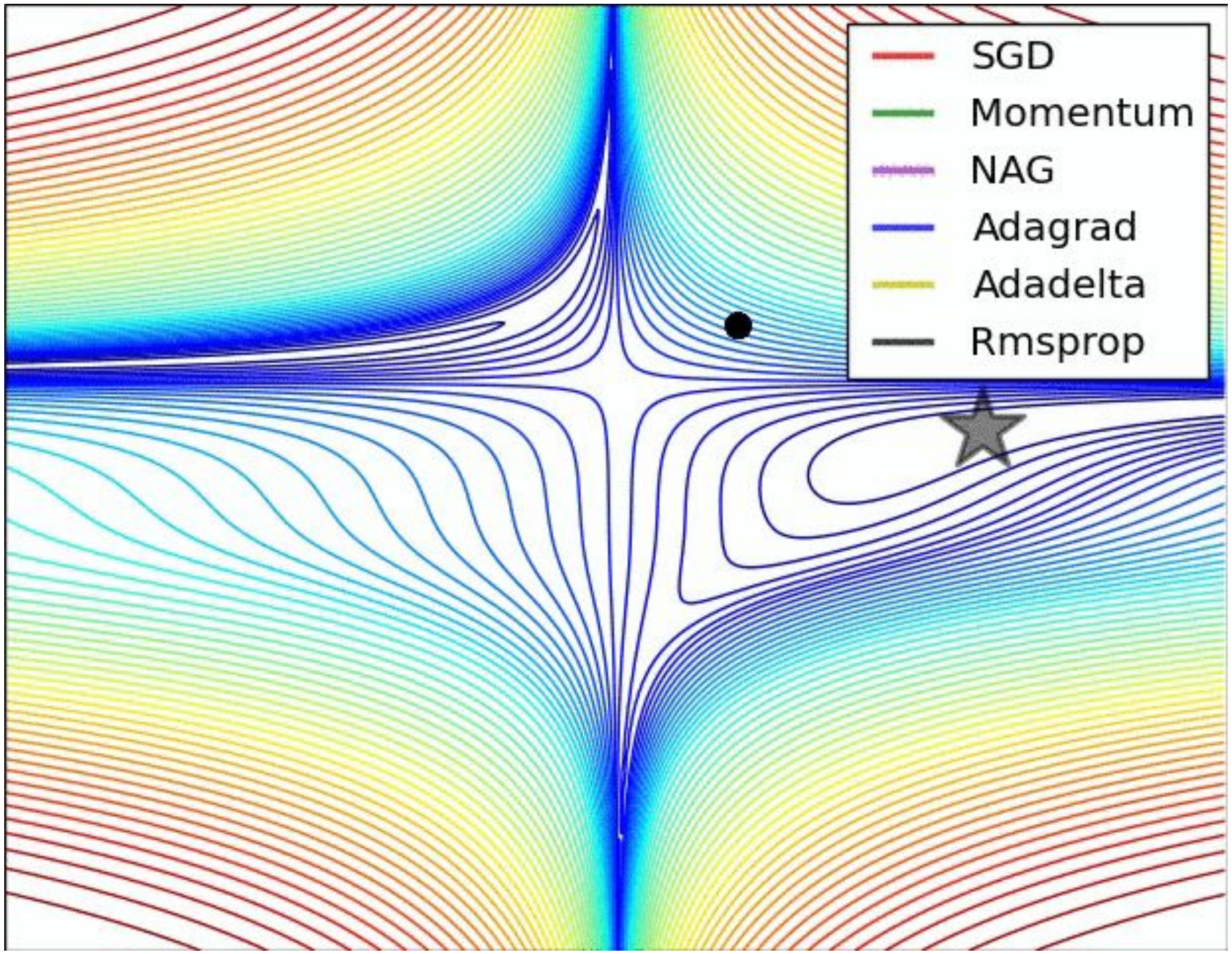
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

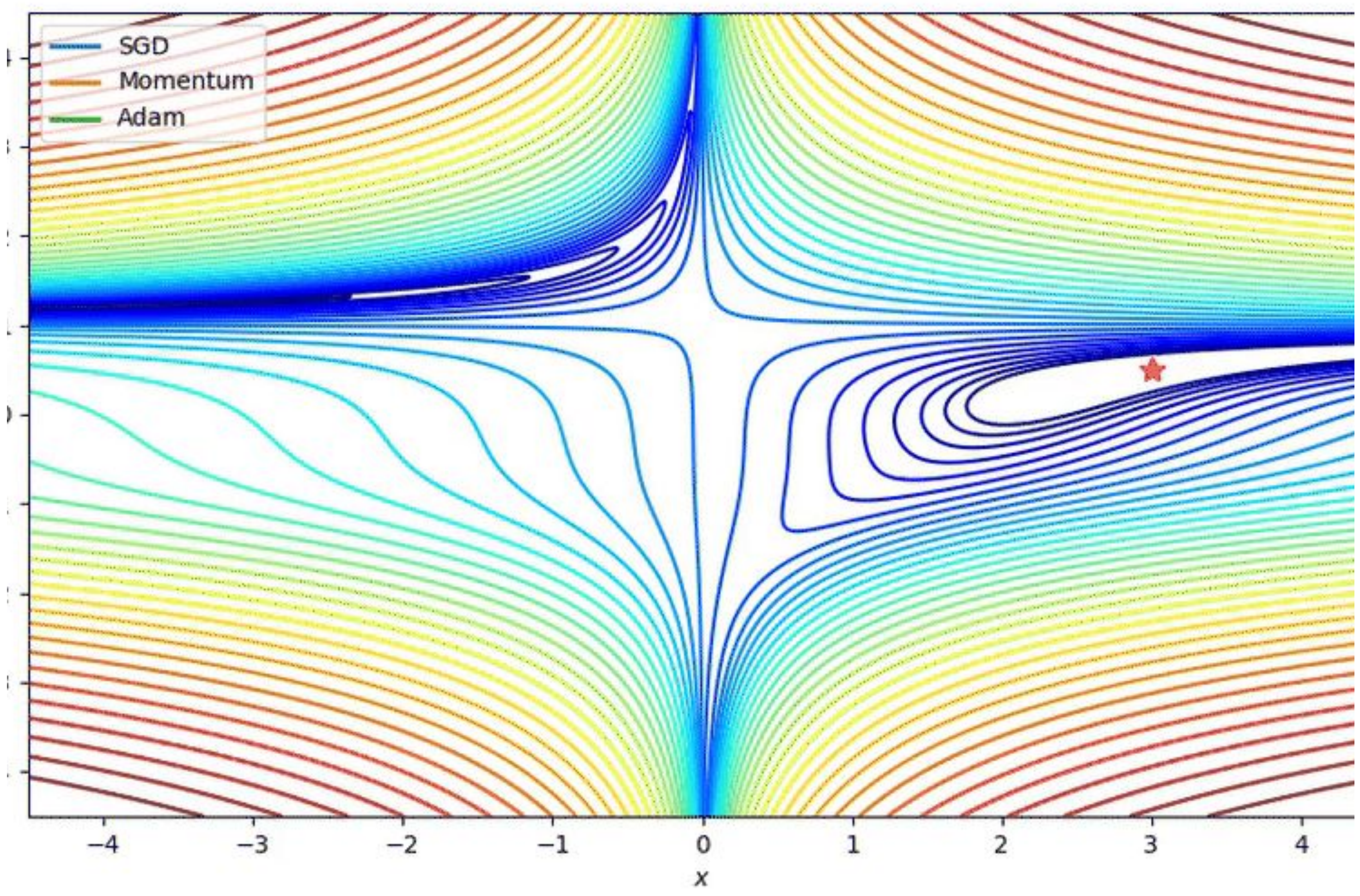
Solvers - which one to choose?

- If your input data is sparse (or your architecture is fragile!), then you likely achieve the best results using one of the adaptive learning-rate methods. An additional benefit is that you won't need to tune the learning rate but likely achieve the best results with the default value.
- In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, **Adam might be the best overall choice**.
- Interestingly, many recent papers use vanilla SGD without momentum and a simple learning rate annealing schedule.

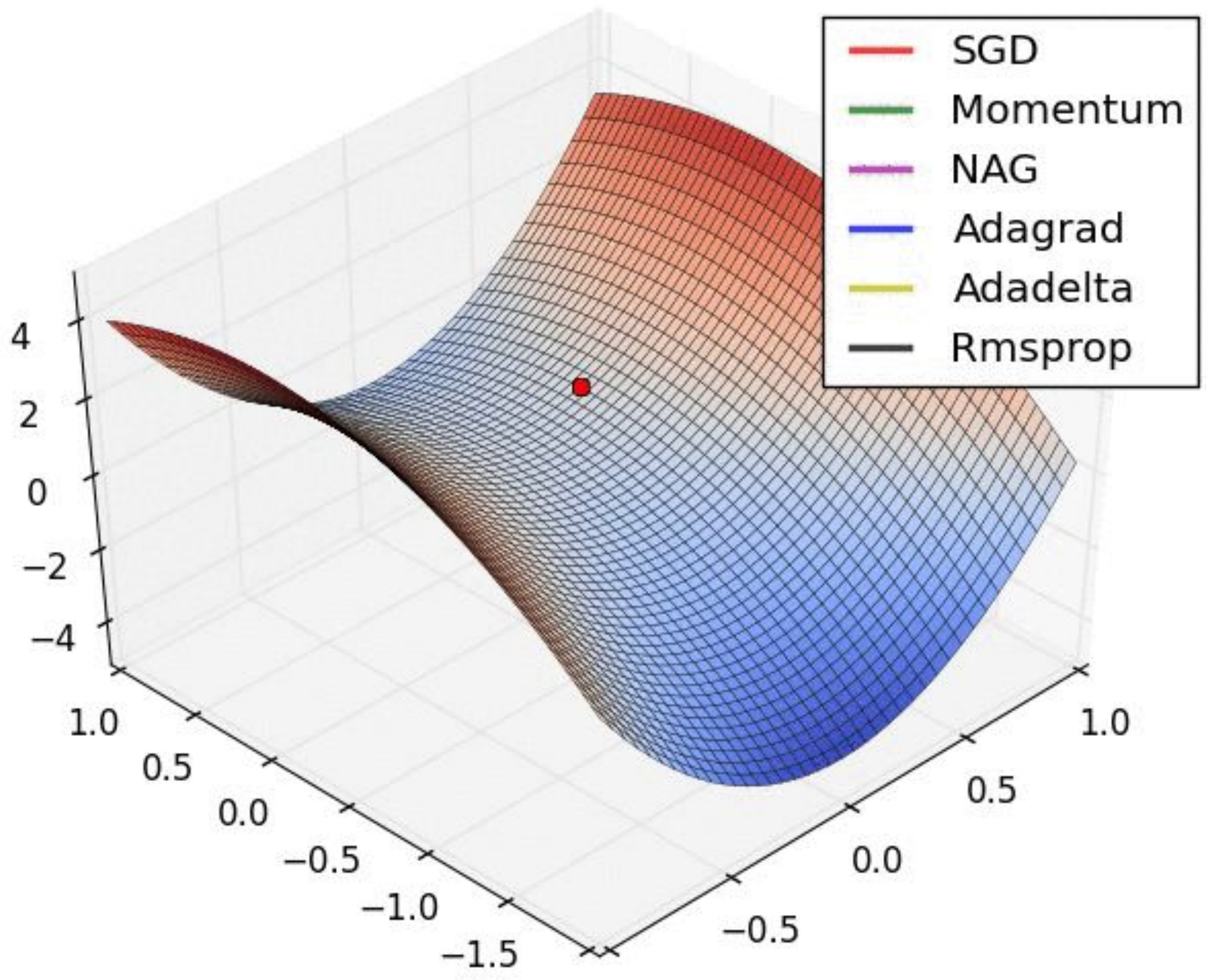
Solvers - visual comparison



Solvers - visual comparison



Solvers - visual comparison



PYTORCH

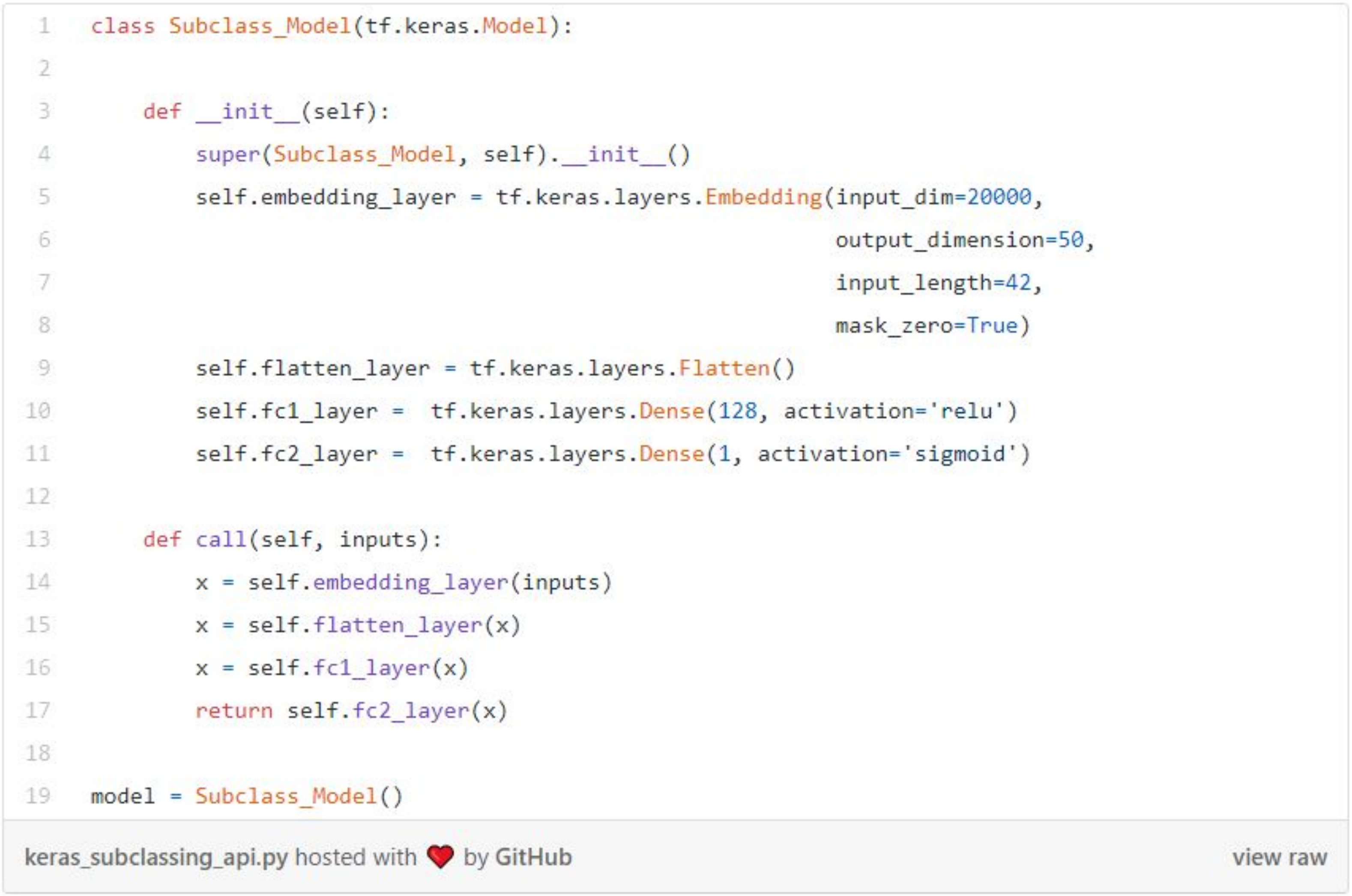
vs

TensorFlow

Pytorch vs Tensorflow

- Most popular frameworks
- Supported by big companies - Facebook / Google !
- Huge differences in the programming paradigm, until TF 2.0 !
- Now they share the same paradigm, a similar syntax, with only marginal differences
- Becoming expert in one of them means an easy switch to the other one

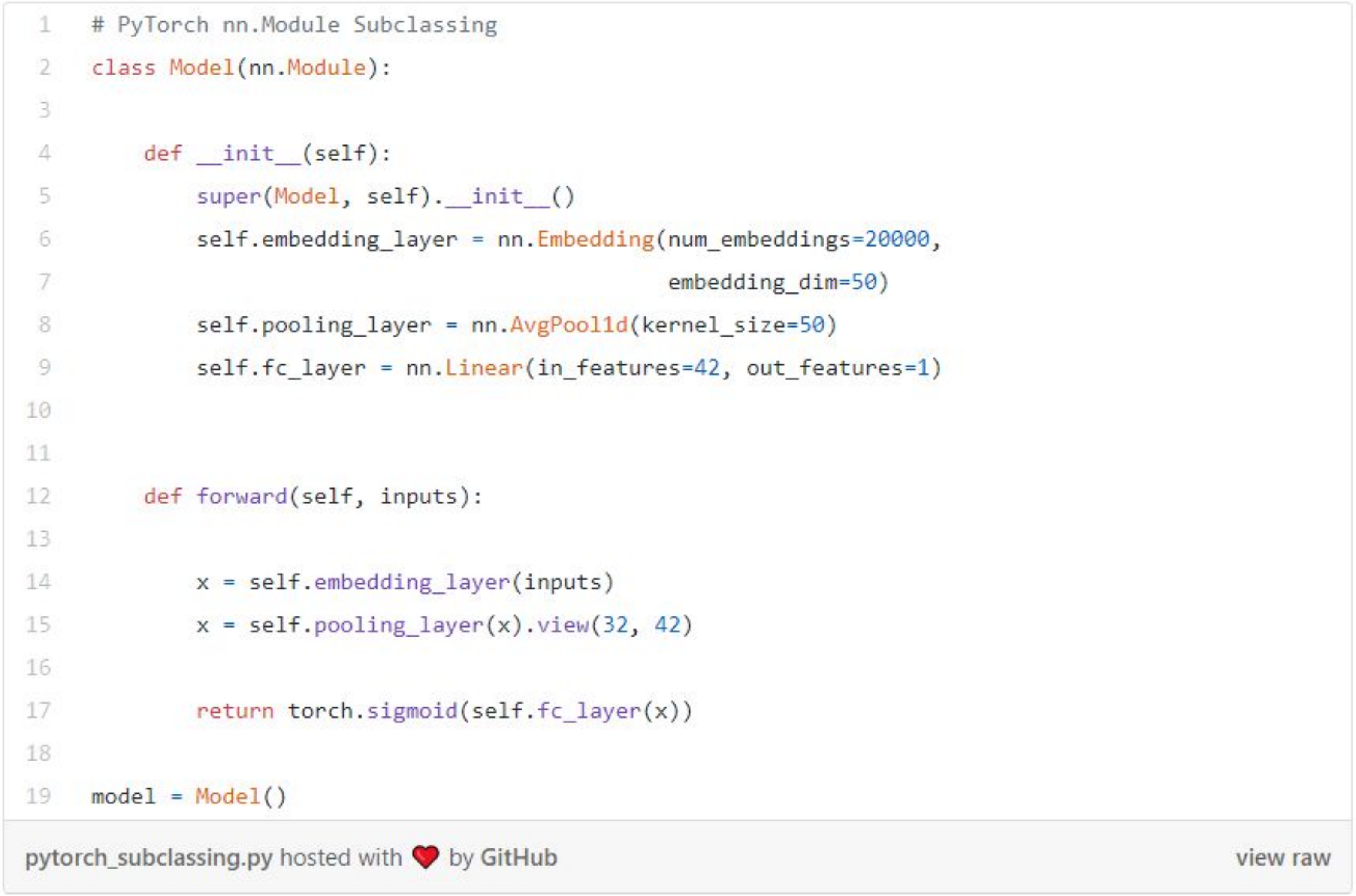
Tensorflow subclassing



```python
class Subclass_Model(tf.keras.Model):

    def __init__(self):
        super(Subclass_Model, self).__init__()
        self.embedding_layer = tf.keras.layers.Embedding(input_dim=20000,
                                                         output_dimension=50,
                                                         input_length=42,
                                                         mask_zero=True)
        self.flatten_layer = tf.keras.layers.Flatten()
        self.fc1_layer =  tf.keras.layers.Dense(128, activation='relu')
        self.fc2_layer =  tf.keras.layers.Dense(1, activation='sigmoid')

    def call(self, inputs):
        x = self.embedding_layer(inputs)
        x = self.flatten_layer(x)
        x = self.fc1_layer(x)
        return self.fc2_layer(x)

model = Subclass_Model()
```

keras_subclassing_api.py hosted with ❤ by GitHub   view raw

Pytorch subclassing



```python
# PyTorch nn.Module Subclassing
class Model(nn.Module):

    def __init__(self):
        super(Model, self).__init__()
        self.embedding_layer = nn.Embedding(num_embeddings=20000,
                                            embedding_dim=50)
        self.pooling_layer = nn.AvgPool1d(kernel_size=50)
        self.fc_layer = nn.Linear(in_features=42, out_features=1)


    def forward(self, inputs):

        x = self.embedding_layer(inputs)
        x = self.pooling_layer(x).view(32, 42)

        return torch.sigmoid(self.fc_layer(x))

model = Model()
```

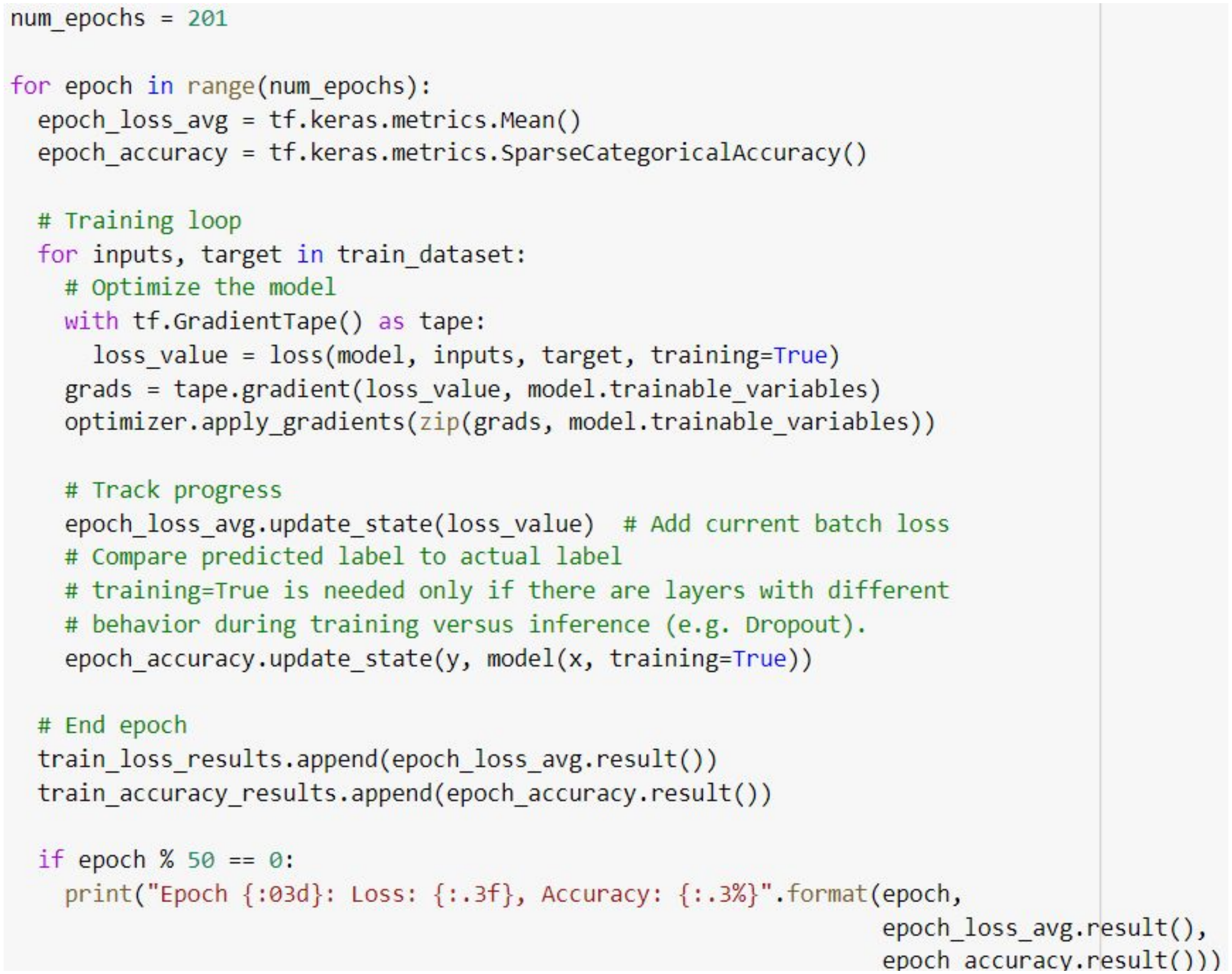
pytorch_subclassing.py hosted with ❤ by GitHub   view raw

Tensorflow training (1)

```
model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=
['accuracy'])
```

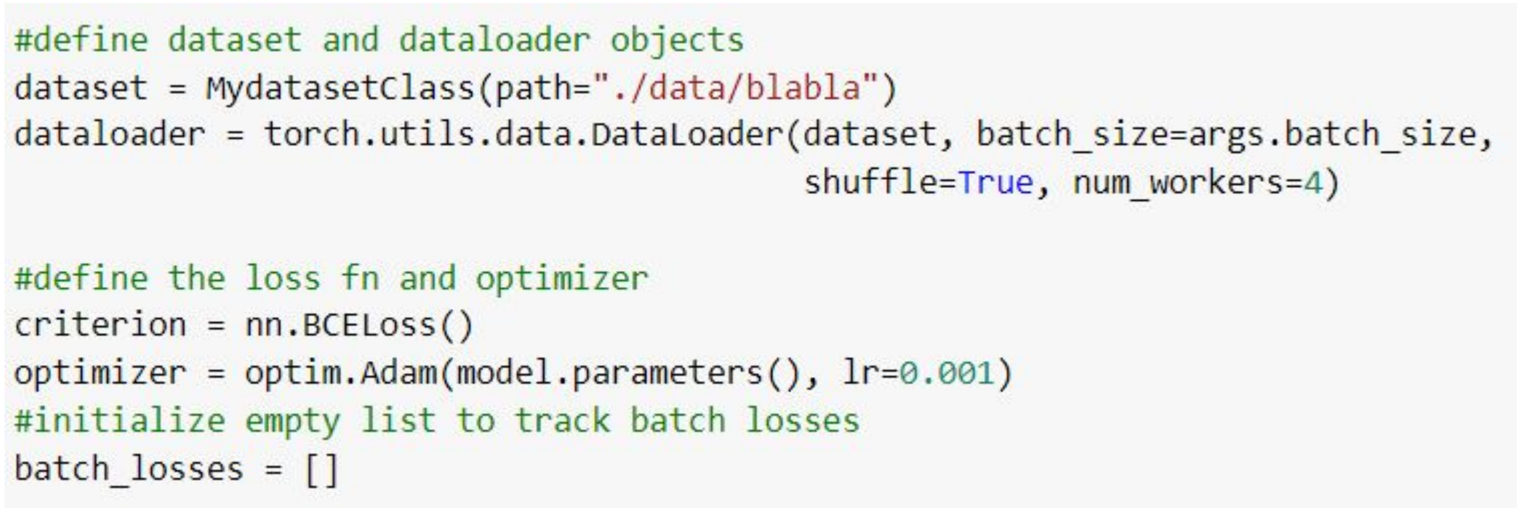
```
model.fit(x=X, y, batch_size=32, epochs=5, verbose=2,
validation_split=0.2)
```

Tensorflow training (2)



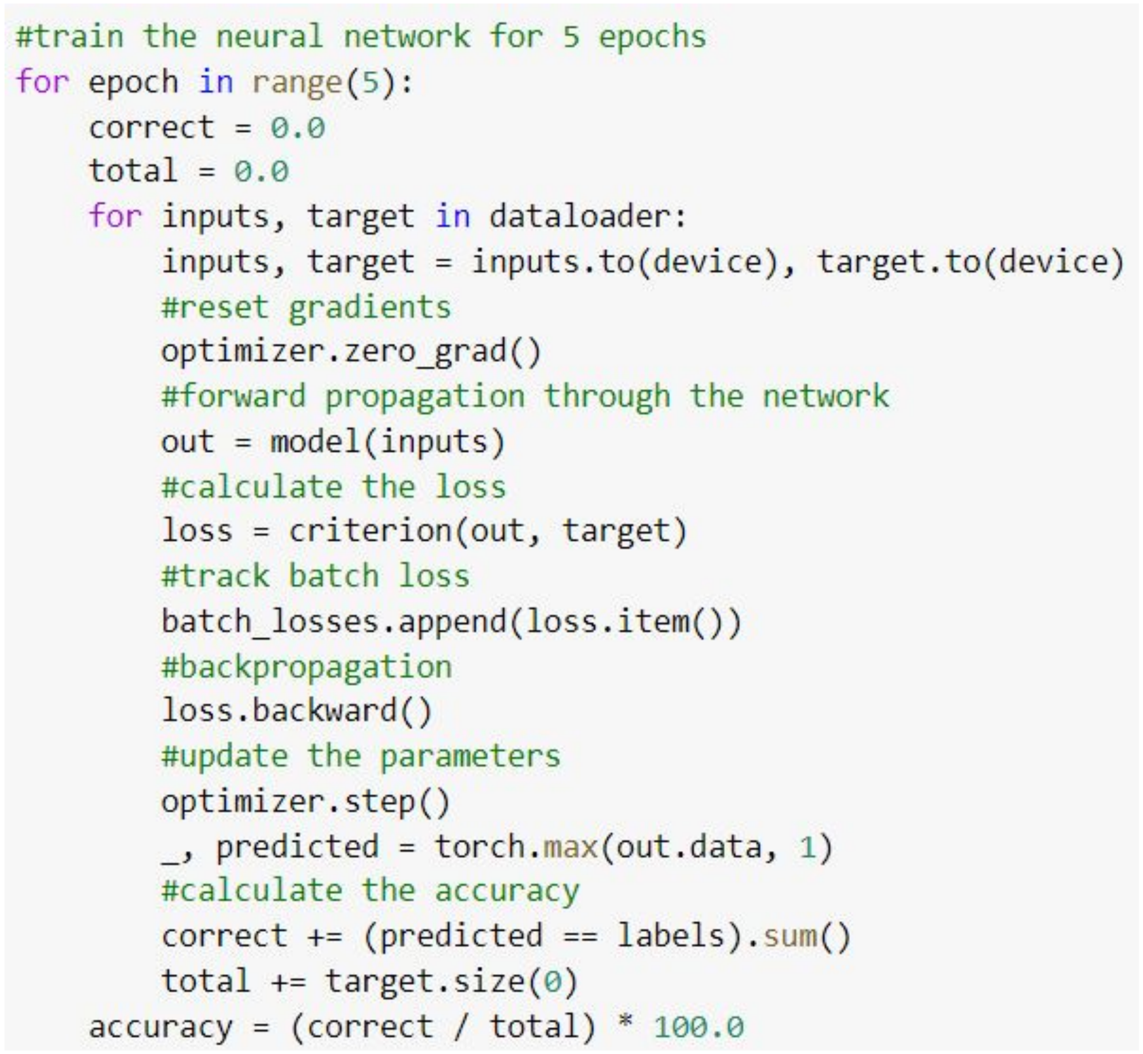
```python
num_epochs = 201

for epoch in range(num_epochs):
  epoch_loss_avg = tf.keras.metrics.Mean()
  epoch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

  # Training loop
  for inputs, target in train_dataset:
    # Optimize the model
    with tf.GradientTape() as tape:
      loss_value = loss(model, inputs, target, training=True)
    grads = tape.gradient(loss_value, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

    # Track progress
    epoch_loss_avg.update_state(loss_value)  # Add current batch loss
    # Compare predicted label to actual label
    # training=True is needed only if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    epoch_accuracy.update_state(y, model(x, training=True))

  # End epoch
  train_loss_results.append(epoch_loss_avg.result())
  train_accuracy_results.append(epoch_accuracy.result())

  if epoch % 50 == 0:
    print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,
                                                                epoch_loss_avg.result(),
                                                                epoch_accuracy.result()))
```

Pytorch training



```python
#define dataset and dataloader objects
dataset = MydatasetClass(path="./data/blabla")
dataloader = torch.utils.data.DataLoader(dataset, batch_size=args.batch_size,
                                         shuffle=True, num_workers=4)

#define the loss fn and optimizer
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
#initialize empty list to track batch losses
batch_losses = []
```

Pytorch training - continue

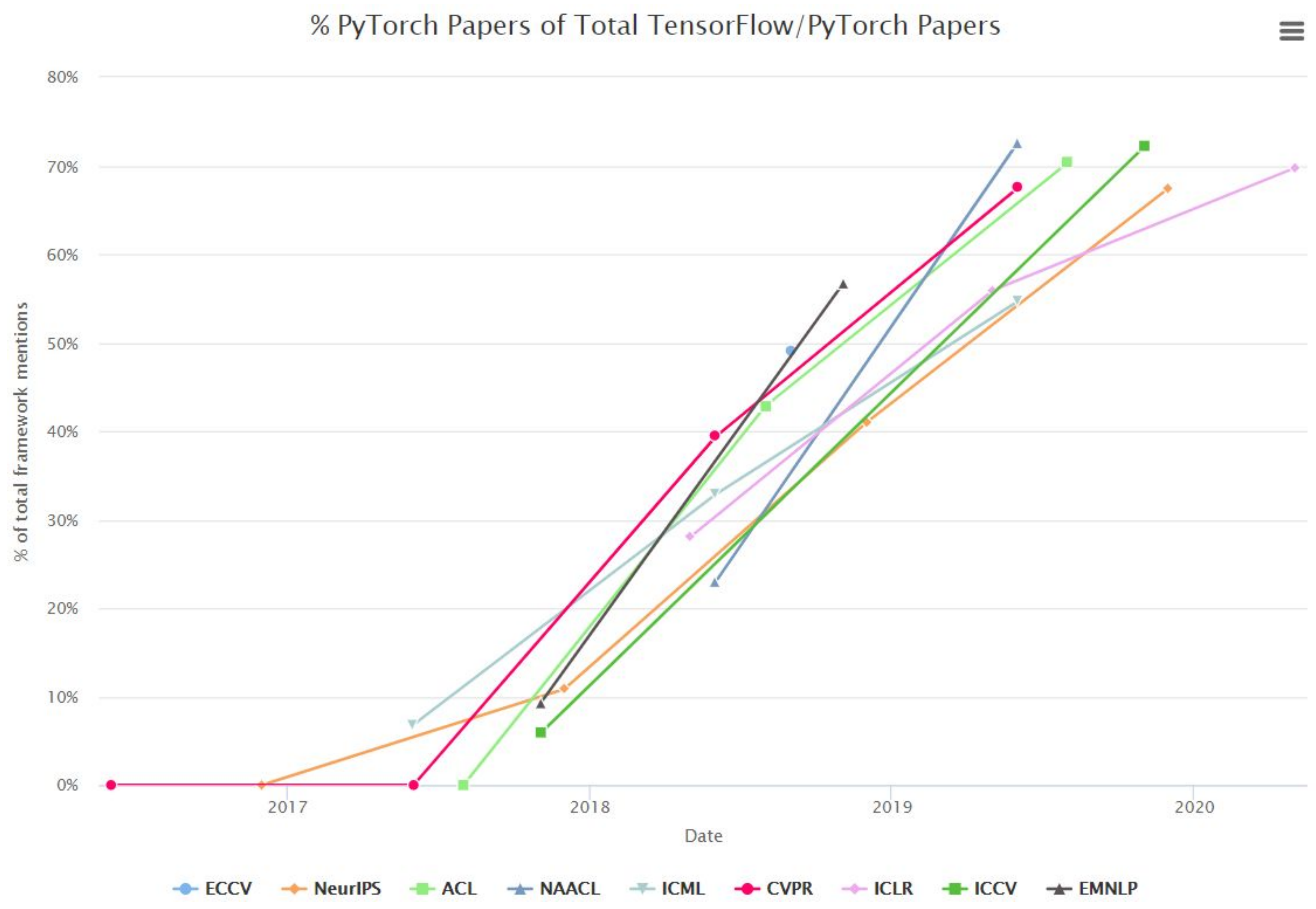


```
#train the neural network for 5 epochs
for epoch in range(5):
    correct = 0.0
    total = 0.0
    for inputs, target in dataloader:
        inputs, target = inputs.to(device), target.to(device)
        #reset gradients
        optimizer.zero_grad()
        #forward propagation through the network
        out = model(inputs)
        #calculate the loss
        loss = criterion(out, target)
        #track batch loss
        batch_losses.append(loss.item())
        #backpropagation
        loss.backward()
        #update the parameters
        optimizer.step()
        _, predicted = torch.max(out.data, 1)
        #calculate the accuracy
        correct += (predicted == labels).sum()
        total += target.size(0)
    accuracy = (correct / total) * 100.0
```

Pytorch vs Tensorflow

- Tensorflow meant for production, Pytorch meant for research
- Nowadays both are viable for both fields
- Pytorch is the most popular choice on ML conferences

Pytorch vs Tensorflow



That’s all!