Optimization for Deep Models

Optimizers, Adaptive Learning Rates, and Second-Order Methods

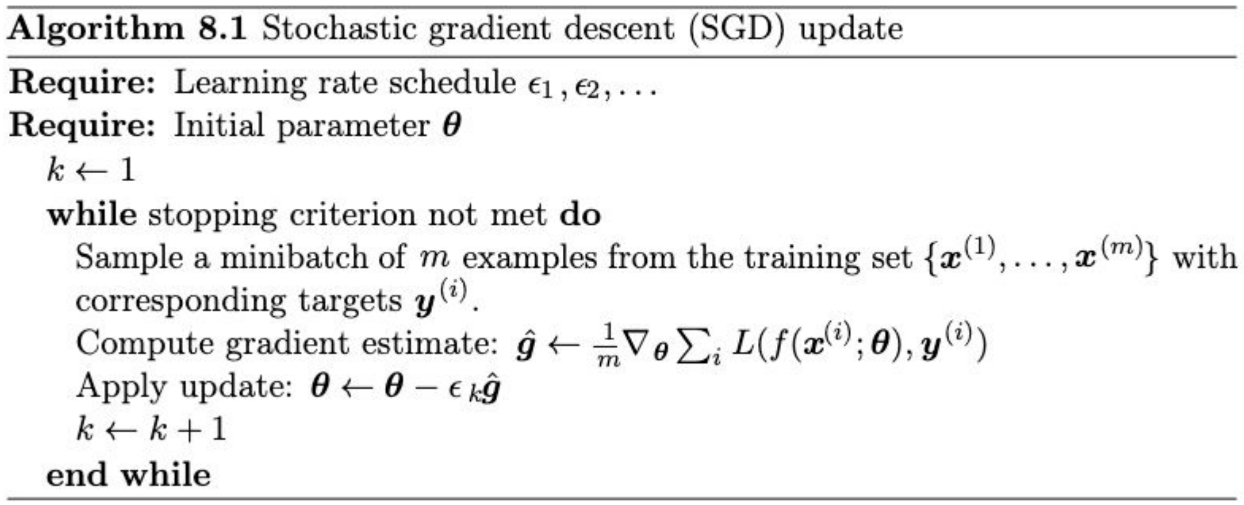
N. Rich Nguyen, PhD
SYS 6016

Optimization for Deep Neural Nets

1. **Initialization:** how to initialize the weights so that they do not saturate?
2. **Activation:** how to solve the vanishing gradient problem?
3. **Normalization:** how to get the model to learn the optimal scale?
4. **Optimizers:** when gradient descent was too slow or not good enough?
5. **Adaptive Learning Rate**: what if convergence is too slow or sub-optimal?
6. **Second-Order Training Methods**: can we make use of second derivatives?

4. Basic Optimizers

Stochastic Gradient Descent



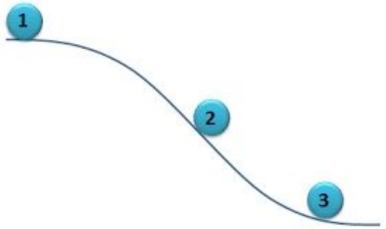
Algorithm 8.1 Stochastic gradient descent (SGD) update

```
Require: Learning rate schedule $\epsilon_1, \epsilon_2, \dots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while
```

- SGD is **the most widely used algorithm** in deep learning
- Training time does not grow with the number of training examples
- To study convergence speed, measure the **excess error** $J(\theta) - \min_{\theta} J(\theta)$
- When applied to a convex problem, excess error is $O(1/k)$ in k steps

Momentum Optimization

- Gradient Descent take **same small steps** down the slope → slow



$$\mathbf{g} \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), y^{(i)})$$
$$\theta \leftarrow \theta - \epsilon\mathbf{g}$$

- Imagine a bowling ball rolling down a slope on a smooth surface → accelerate
- Momentum **accumulates past gradients** and continues to move in their direction to **accelerate the learning**

$$\mathbf{m} \leftarrow \boxed{\alpha}\mathbf{m} - \epsilon\mathbf{g}$$
$$\theta \leftarrow \theta + \mathbf{m}$$

momentum hyperparameter (0.9 means to multiply the max speed by 10 relative to GD)

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Nesterov Momentum

A variant of Momentum that was inspired by Nesterov's accelerated gradient:

$$\bar{\theta} \leftarrow \theta \boxed{+ \alpha \mathbf{m}} \longleftarrow \text{Interim update}$$

$$\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\bar{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \bar{\theta}), \mathbf{y}^{(i)})$$

$$\mathbf{m} \leftarrow \alpha \mathbf{m} - \epsilon \mathbf{g}$$

$$\theta \leftarrow \theta + \mathbf{m}$$

The difference between this and Momentum is **when** the gradient is evaluated. With this, the gradient is evaluated **after** the current momentum is applied.

- In convex **batch** gradient case, Nesterov brings the rate of convergence of excess error from `O(1/k)` to $O(1/k^2)$ after `k` steps as shown by Nesterov.
- Unfortunately, in stochastic gradient case, Nesterov does not improve this rate of convergence.

5. Optimizers with Adaptive Learning Rates

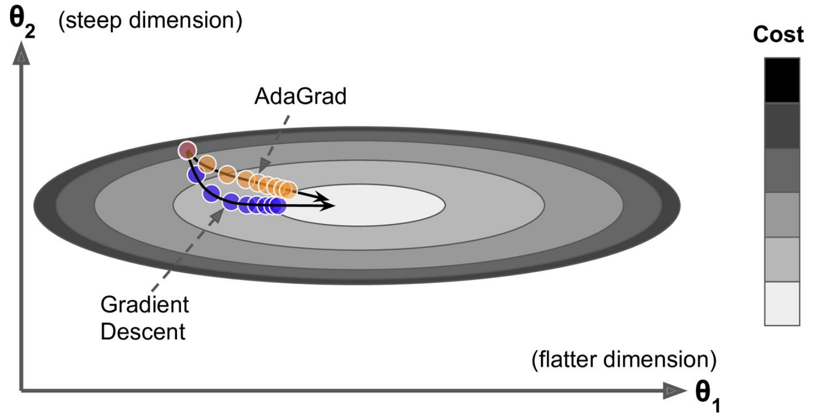
Adaptive Learning Rates

- **Learning rate ε** is one of the **most-difficult-to-set** hyperparameters
- The loss function is often highly **sensitive** to some direction of parameter space and insensitive to others
- Momentum can somewhat mitigate these issues, but it introduces another **momentum hyperparameter α**.
- *Can we use a separate learning rate for each parameter and adapt these learning rate throughout the course of training?*

→ There are a number of mini-batch based optimization methods that **adapt the learning rate of model parameters**. Let us visit these methods!

AdaGrad

- Consider a elongated bowl shaped loss function, SGD may start quickly by going down the steepest slope, but will slowly go down the *plateaus* bottom of the bowl.
- AdaGrad (Duchi, 2011) individually adapts the learning rates of all model parameters by **scaling** them **inversely proportional to the square root** of the **sum of all previous** **squared values of the gradient**.
- Param w/ **largest** partial derivative of loss → rapidly **decrease** their learning rate
- The **net effect** is greater progress in the **more gently sloped direction**.



scale

squared values of the gradient

element-wise division

$$\boxed{\mathbf{s}} \leftarrow \mathbf{s} + \boxed{\mathbf{g} \otimes \mathbf{g}}$$

$$\theta \leftarrow \theta - \epsilon \mathbf{g} \boxed{\oslash \sqrt{\mathbf{s}}}$$

RMSProp

- AdaGrad **scale down** the **learning rate too fast** before arriving at a **nonconvex** region that locally resembles a convex bowl.
- RMSProp (Hinton, 2012) fixes this by changing the gradient accumulation into **an exponentially decaying average** to discard history from extreme past
- RMSProp accumulates only the gradient from most recent iteration by using **exponential decay *β*** (typically set at 0.9) in the first step
- Empirically, RMSProp has been shown to be effective for deep neural nets

$$\mathbf{s} \leftarrow \boxed{\beta}\mathbf{s} + (1 - \beta)\mathbf{g} \otimes \mathbf{g}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\mathbf{g} \oslash \sqrt{\mathbf{s}}$$

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Adam (Adaptive Moment Estimation)

Adam (Kingma, 2014) combines ideas of Momentum and RMSProp scaling:

- Like Momentum, it keeps track of an exponentially decaying moving average of **past gradients**.
- Like RMSProp, it also keeps track of an exponentially decaying moving average of **past squared gradients**

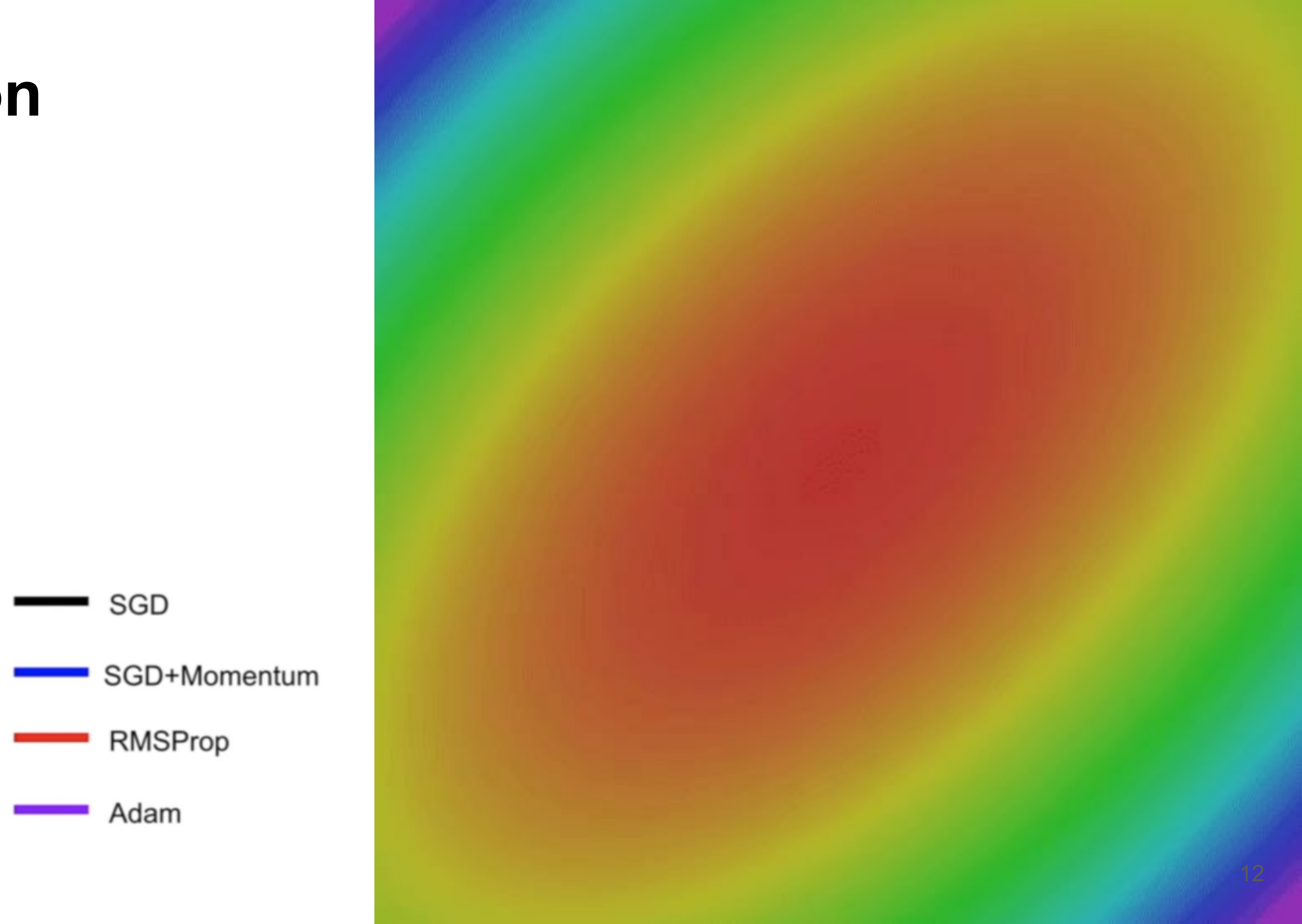
$$\text{momentum} \rightarrow \boxed{\mathbf{m}} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1)\mathbf{g}$$

$$\text{scale} \rightarrow \boxed{\mathbf{s}} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\mathbf{g} \otimes \mathbf{g}$$

$$\theta \leftarrow \theta + \epsilon \mathbf{m} \oslash \sqrt{\mathbf{s}}$$

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

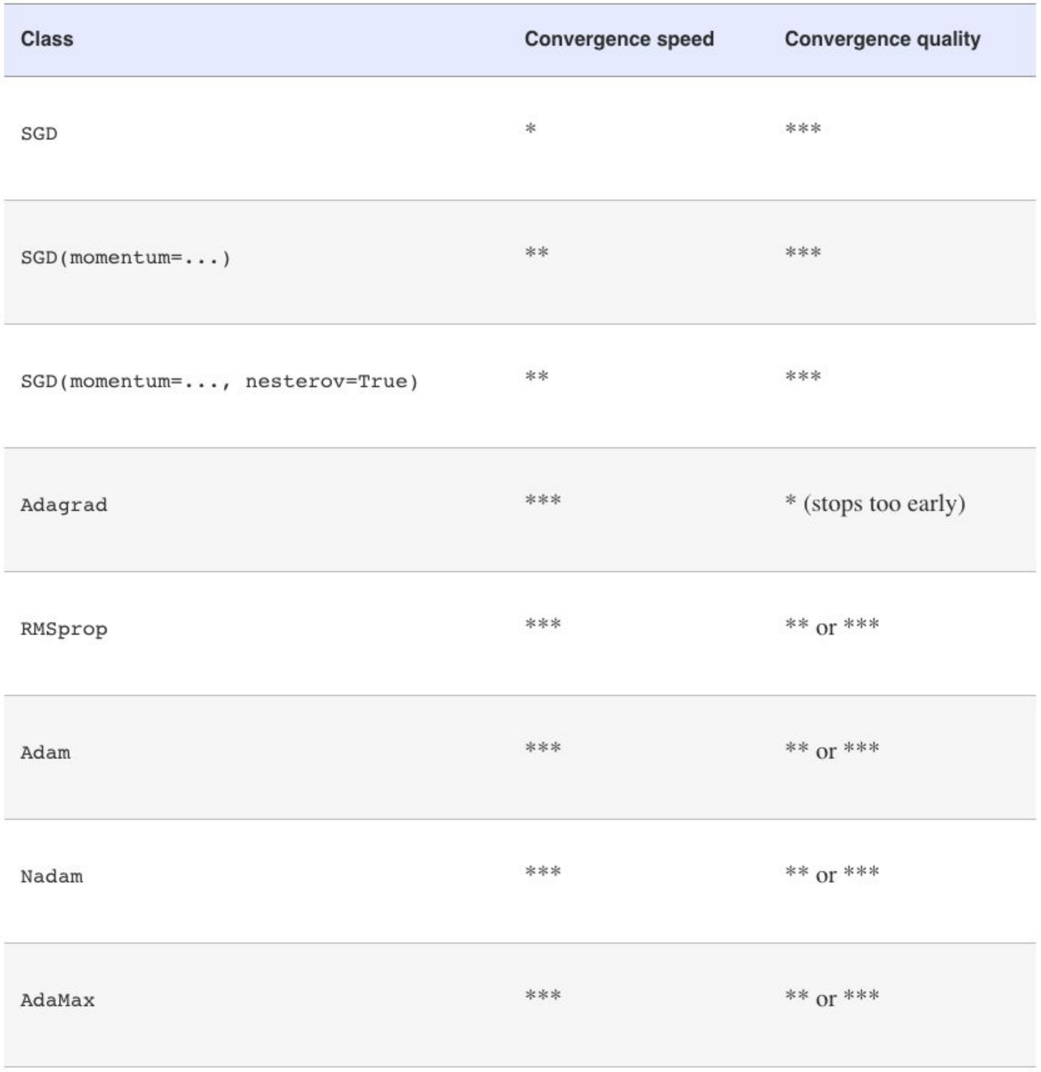
Animation



Comparison

Currently, **no consensus** on which algorithm one should choose

Seems to depend largely on the user’s **familiarity** with the algorithm!



Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

6. Second-Order Methods

Second-order Methods

- Optimization algorithms that use only the gradient (ie. gradient descent, momentum, adam) are called **First-Order optimization methods**
- **Second-Order methods** use of second derivatives to improve optimization
- **The second derivative** tells us how the first derivative will change as we vary the inputs (same as measuring the **curvature** of a function)
- Matrix contains second deriv. of multi-dimensional inputs is called a **Hessian**:

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial_{x_i}\partial_{x_j}} f(\mathbf{x})$$

- The second derivative can be used to determined whether a critical point is a *local maximum* (`f'=0` and `f''>0`), *local minimum* (`f'=0` and `f''< 0`), or a *saddle point* (in many cases `f'=0` and `f''=0`).

Newton's Method

- Newton's method is the most widely used second-order method.
- Based on a **Taylor series expansion** to approximate J(θ) near some θ_0:

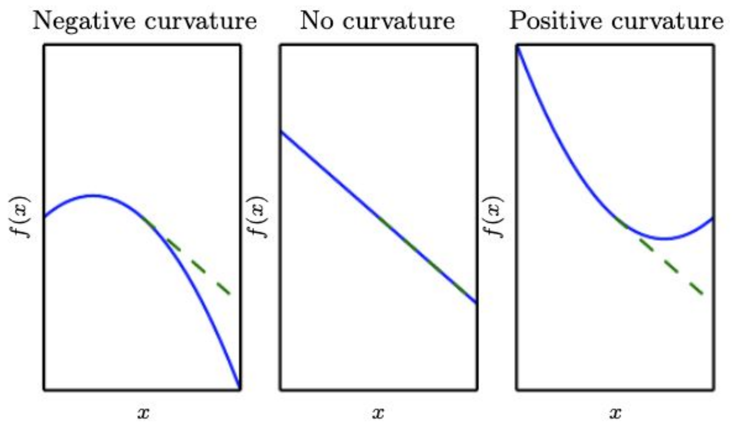
$$J(\theta) \approx \boxed{J(\theta_0)} + \boxed{(\theta - \theta_0)^\top \mathbf{g}} + \boxed{\frac{1}{2}(\theta - \theta_0)^\top \boxed{\mathbf{H}}(\theta - \theta_0)}$$

previous value of the loss function

expected improvement due to the slope

correction applied to account for curvature

the Hessian of J w.r.t. θ evaluated at θ_0



Newton Parameter Update

Solving for the critical point of J(θ), we obtain the Newton parameter update:

$$\theta^* = \theta_0 - \mathbf{H}^{-1}\mathbf{g}$$

- For a quadratic function, Newton's method **jumps directly** to the minimum
- For convex but not quadratic function, this update can be applied **iteratively**
- For non-convex function in deep learning, this can be problematic around *saddle points* where updates can move in the **wrong direction** → mitigate by adding a **regularization term** on matrix **H**

Computational Issue: Inversion of a `k x k` matrix **H** results in a complexity of $O(k^3)$ at every iteration → only practical on small networks.

Conjugate Gradients

- Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending **conjugate directions.**
- In a quadratic bowl, gradient descent progresses in a line search direction of **ineffective zig-zag** → need to find a direction that is **conjugate** to the previous line.
- At iteration `t`, current direction $\mathbf{p}_t$ takes the form:

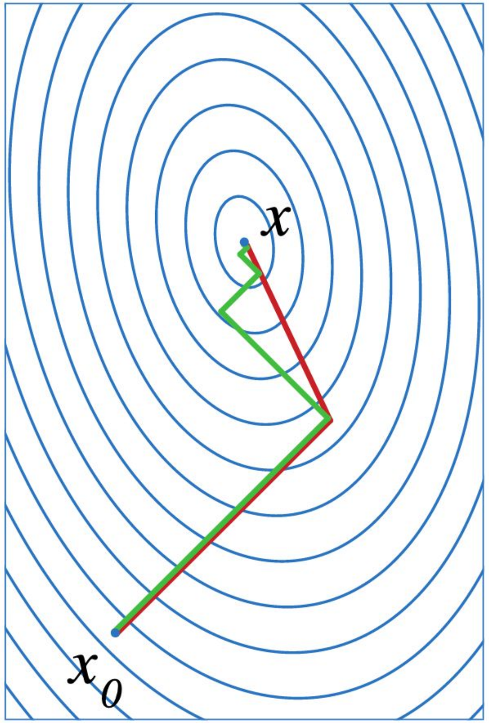
$$\mathbf{p}_t = -\mathbf{g}_t + \boxed{\beta_t}\mathbf{p}_{t-1}$$

how much of the previous direction should be added back

$$\theta_{t+1} = \theta_t + \epsilon \mathbf{p}_t$$

$$\beta_t = \frac{(\mathbf{g}_t - \mathbf{g}_{t-1})^\top \mathbf{g}_t}{\mathbf{g}_{t-1}^\top \mathbf{g}_{t-1}}$$

- In a k-dimensional parameter space, conjugate gradient requires at most `k` line searches to reach the minimum.



Broyden-Fletcher-Goldfarb-Shanno (BFGS) algo

- BFGS attempts to bring some advantages of Newton's method **without** inversing the Hessian **H** → It is a **quasi-Newton** method.
- Approximate the inverse with **M**`t` that is *iteratively refined by low-rank updates*
- Then, direction of descent **p**`t` is determined along with parameter updates:

$$\mathbf{p}_t = \boxed{\mathbf{M}_t}\mathbf{g}_t$$
$$\theta_{t+1} = \theta_t + \epsilon\mathbf{p}_t$$

approximation of H^{-1}

- Like conjugate gradients, BFGS iterates a **series of line searches**
- Unlike conjugate gradients, BFGS spends **less time refining** each line search.
- BFGS must store matrix **M** that requires $O(n^2)$ memory, making it impractical for most deep learning models with millions of parameters → Limited Memory or L-BFGS to have no storage y assuming **M**`t-1` is an identity matrix **or** store $O(n)$ per step by including more information about **Hessian** in a few vectors

Summary: Optimization for Deep Neural Nets

1. **Initialization:** how to initialize the weights so that they do not saturate?
2. **Activation:** how to solve the vanishing gradient problem?
3. **Normalization:** how to get the model to learn the optimal scale?
4. **Optimizers:** when gradient descent was too slow or not good enough?
5. **Adaptive Learning Rate**: what if convergence is too slow or sub-optimal?
6. **Second-Order Training Methods**: can we make use of second derivatives?

Next on Deep Learning

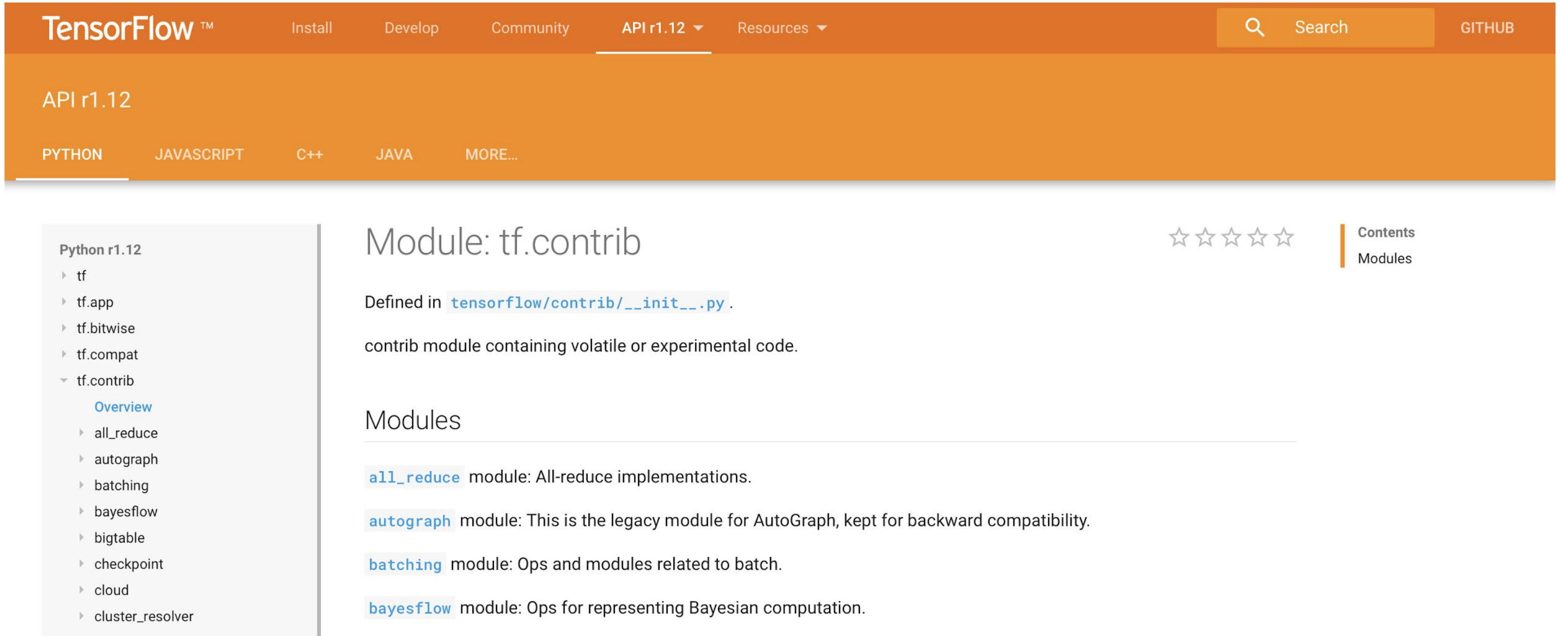
We are now familiar with the **basic family of neural network models** and how to **regularize** and **optimize** them, but there are still **a lot more:**

- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)
- Generative Adversarial Networks (GANs)
- Deep Q-Network (DQN)

Bonus Content

A note on the implementation details of DNN

In this lecture, we are mostly cover the concepts without diving too much into the implementation details. However, in the coding assignment, you will use what we are about to discuss to find the appropriate TensorFlow API.



Module: tf.contrib

Defined in `tensorflow/contrib/__init__.py`.

contrib module containing volatile or experimental code.

Modules

`all_reduce` module: All-reduce implementations.

`autograph` module: This is the legacy module for AutoGraph, kept for backward compatibility.

`batching` module: Ops and modules related to batch.

`bayesflow` module: Ops for representing Bayesian computation.

Fine-Tuning Hyperparameters

Many hyperparameters to tweak, still be considered as a **black art** to find the perfect hyperparameters:

- Number of layers
- Number of neurons per layer
- Type of activation function to use at each layer
- Weight initialization logic
- and much more...

How do you know which combination of hyperparameters is the best for your task?

Number of Hidden Layers

- One hidden layer can model complex functions if it has enough neurons
- Deep networks can model complex functions using exponentially fewer neurons than shallow nets, making them faster to train
- Real-world data often structured in such a hierarchical way:
  - Lower hidden layers model low-level structures (line segment, shape and orientation)
  - Intermediate layers combine low-level structures to intermediate structure (squares, circles)
  - Highest hidden layers combine intermediate structure to model high-level (faces)
- Better generalization if only changes the high hidden layer (ie. hairstyles)
- Start with two hidden layers (98% accuracy on MNIST). Careful for overfitting!
- Use other pretrained state-of-the-art network to perform a similar task

Number of Neurons per Layer

- Obviously, it depends on the type of input and output your task requires.
- MNIST task requires 28 x 28 = 784 input neuron and 10 output neurons
- For hidden layer, a common practice is to form a funnel: with fewer and fewer neurons at higher layers
- Rationale: Many low-level features can coalesce into far fewer high-level features (for MNIST, 300 neurons on the 1st hidden layer and 100 on 2nd layer)
- Simpler approach: pick model with more layer and neuron that needed, than use early stopping.

Most common strategies for learning schedule

Predetermined piecewise constant learning rate

For example, set the learning rate to $\eta_0 = 0.1$ at first, then to $\eta_1 = 0.001$ after 50 epochs. Although this solution can work very well, it often requires fiddling around to figure out the right learning rates and when to use them.

Performance scheduling

Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling

Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0\ 10^{-t/r}$. This works great, but it requires tuning η_0 and r. The learning rate will drop by a factor of 10 every r steps.

Power scheduling

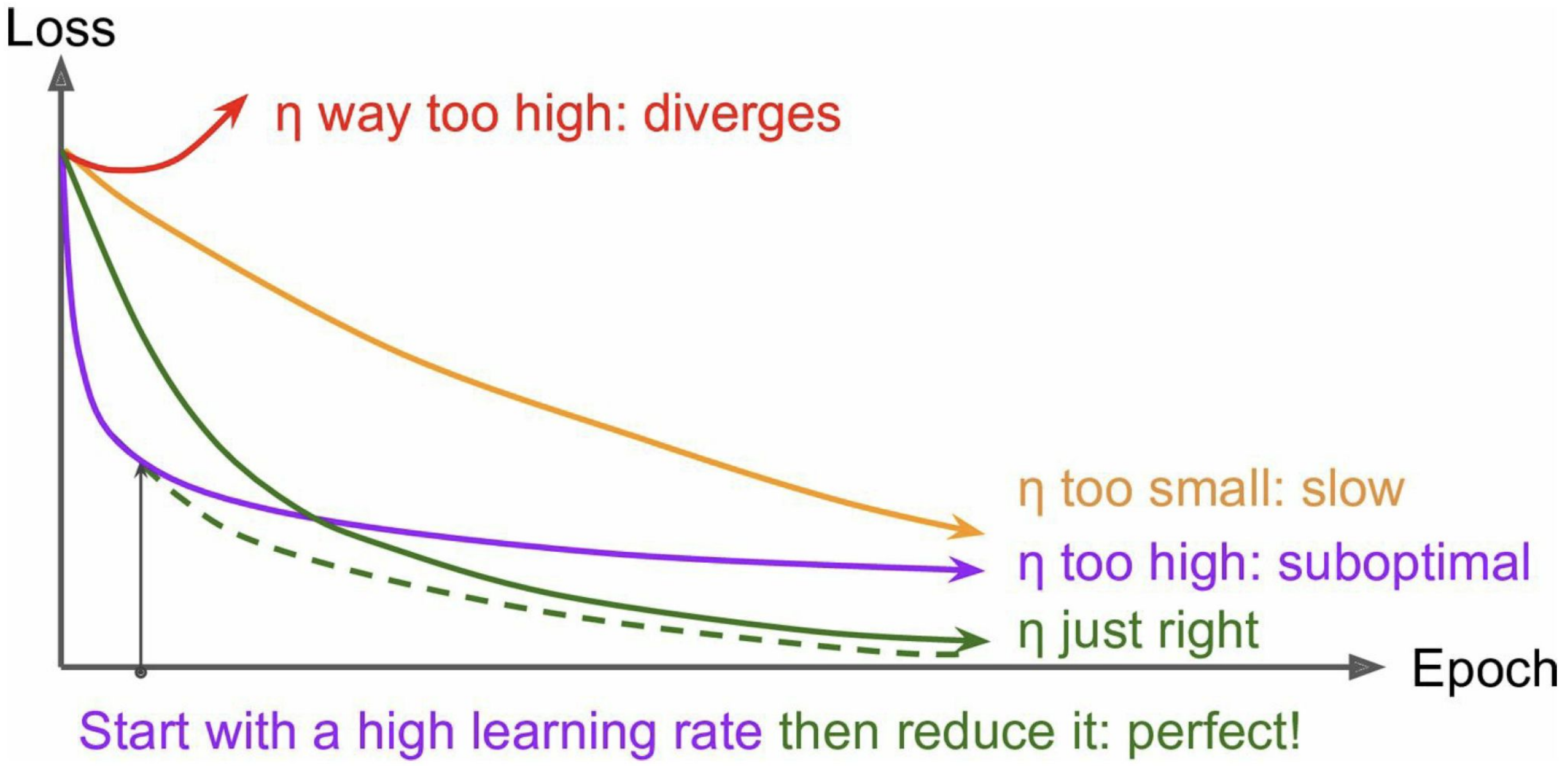
Set the learning rate to $\eta(t) = \eta_0\ (1 + t/r)^{-c}$. The hyperparameter c is typically set to 1. This is similar to exponential scheduling, but the learning rate drops much more slowly.

Implementing the Learning Rate

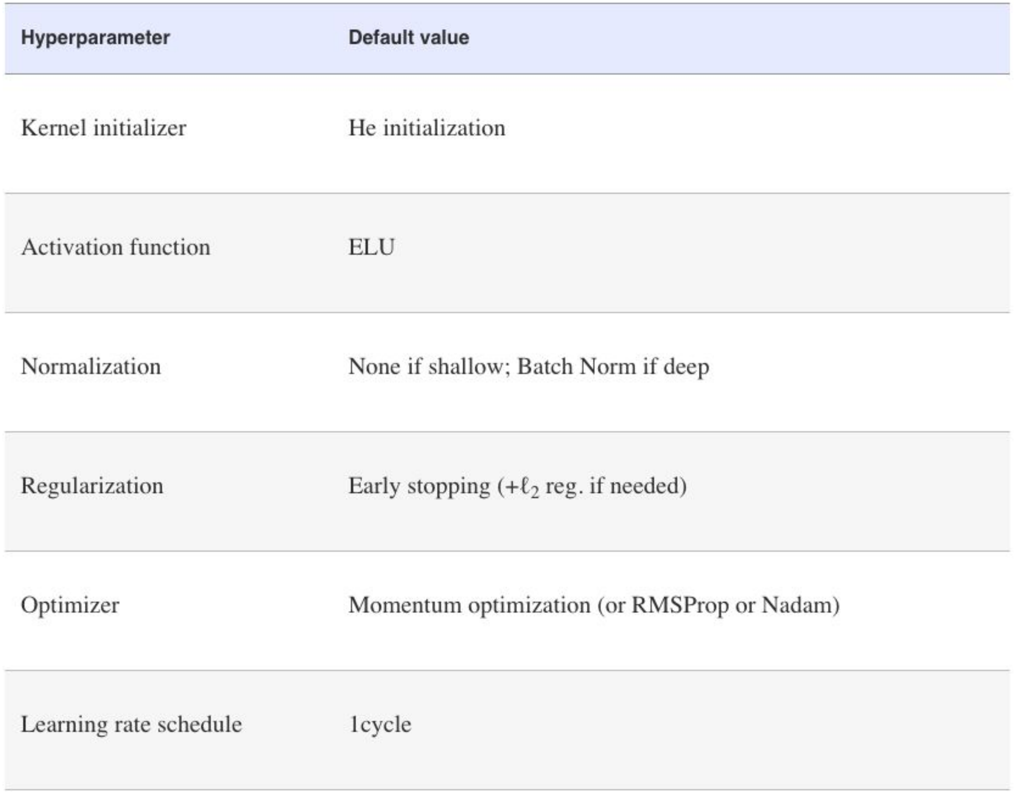
```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

Loss vs. Iteration Epoch



Summary: the Practical Guidelines



Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Regularizing Newton's Method

The *saddle point* situation can be avoided by regularizing the Hessian by adding a constant, α, along the diagonal of the Hessian:

$$\theta^* = \theta_0 - \left[\mathbf{H}(f(\theta_0)) + \alpha\mathbf{I}\right]^{-1}\mathbf{g}$$

- This works well as long as the negative eigenvalue of the Hessian close to 0
- In more extreme direction of curvature, the value of α must be sufficiently large to offset the negative eigenvalue
- In strong negative curvature, α must be so large that Newton's method would make smaller steps then gradient descent