Training Deep Neural Networks

Using Machine Learning Tools

2021

Reading: Géron Chapter 11

Previously …

Training: Minimise cost function
by adjusting model parameters

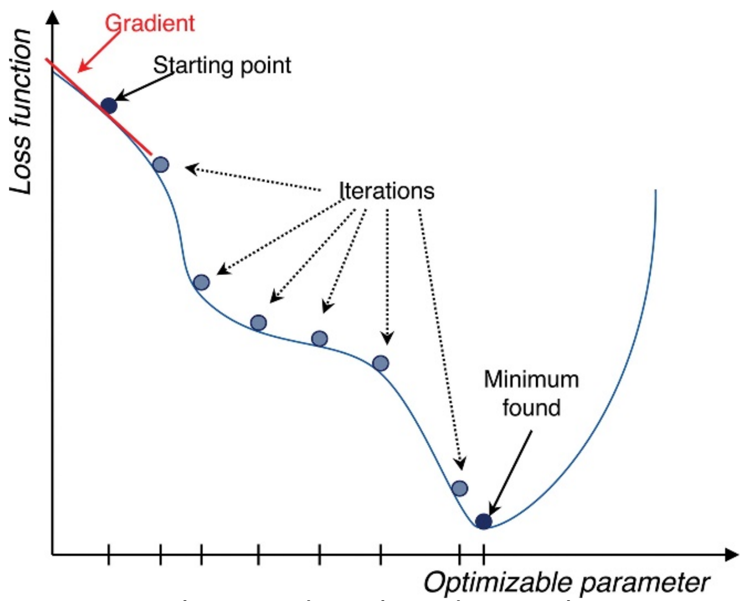
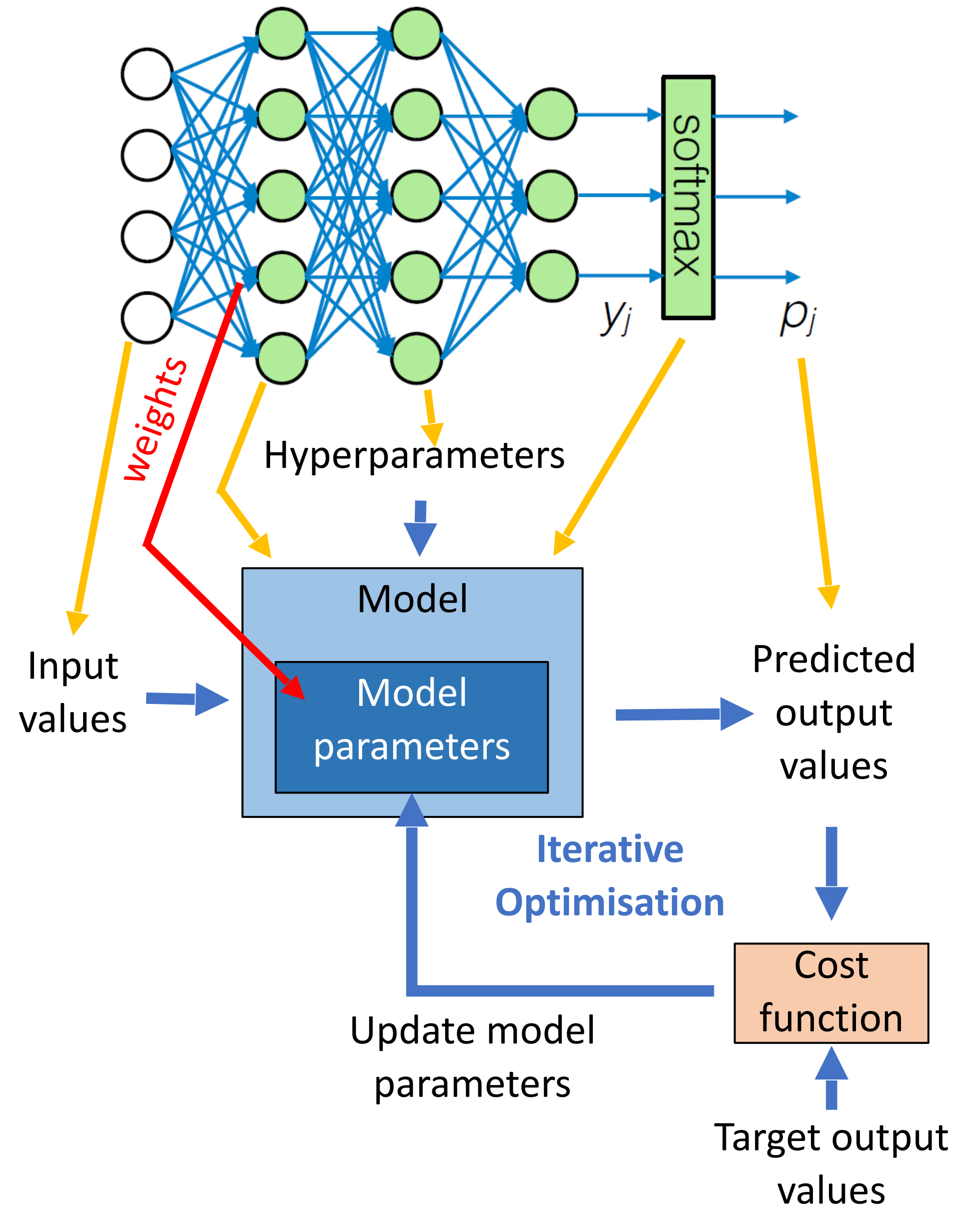


Image: Chartrand et al. RadioGraphics 2017



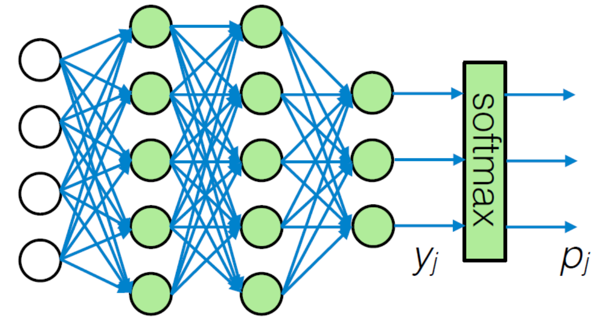
Today …

- Training deep NNs uses gradient descent
- Backpropagation (very briefly)
- Vanishing and exploding gradient problems
  - Four ways of reducing these problems
- Optimisers have been incrementally refined
  - A range of options are available
- Learning rate scheduling and 1cycle method

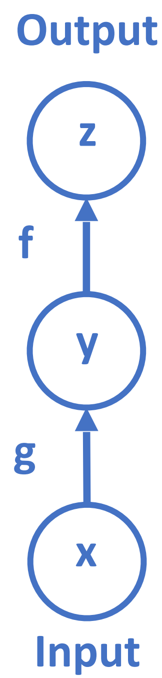
Training Algorithm

- Initialise weights randomly (break symmetry)
- Forward pass:
- Compute loss function
- Backward pass (Backpropagation)

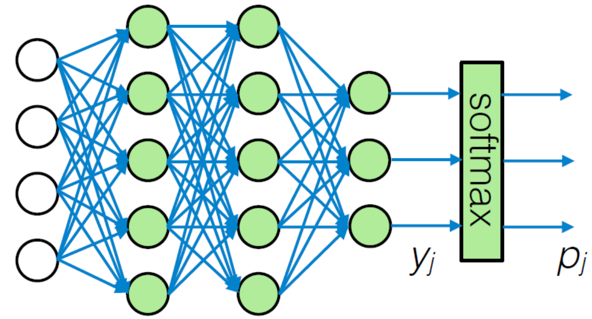
Training Algorithm



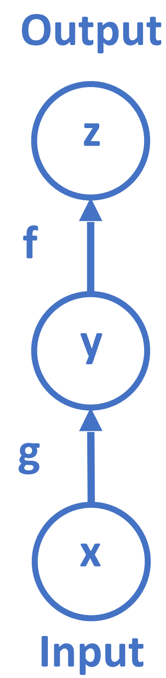
- Initialise weights randomly (break symmetry)
- Forward pass:
- Compute loss function
- Backward pass (Backpropagation)



Training Algorithm



- Initialise weights randomly (break symmetry)
- Forward pass:
- Compute loss function
- Backward pass (Backpropagation)



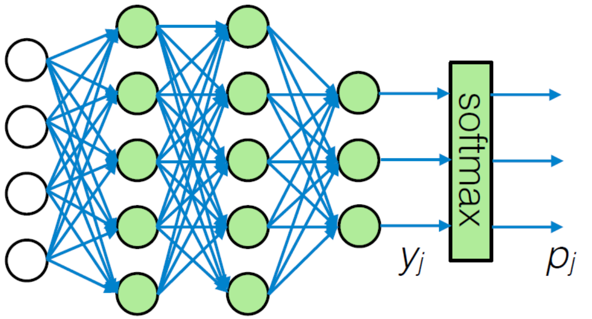
$z = f(y) = f(g(x))$

$\partial z/\partial y = f'(y)$

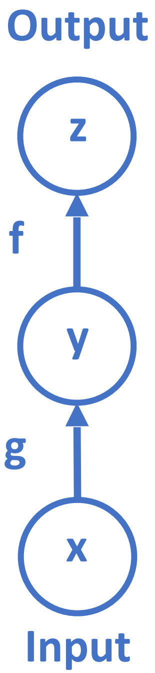
$\partial z/\partial x = \partial z/\partial y * \partial y/\partial x$

$= f'(y) * g'(x)$

Training Algorithm



- Initialise weights randomly (break symmetry)
- Forward pass:
- Compute loss function
- Backward pass (Backpropagation)
- Gradient descent step on weights
- Repeat for batches of data (mini-batch)
- Repeat for multiple epochs



Vanishing & Exploding Gradients

Backpropagation calculates gradients of loss for many, many weights

Vanishing gradients:

- Gradients can get small depending on where they are in activation function
- Many contributions => individual terms are small
- Small gradients => slow convergence

Exploding gradients:

- Chain rule can lead to increasing and opposing contributions
- Large gradients => instability

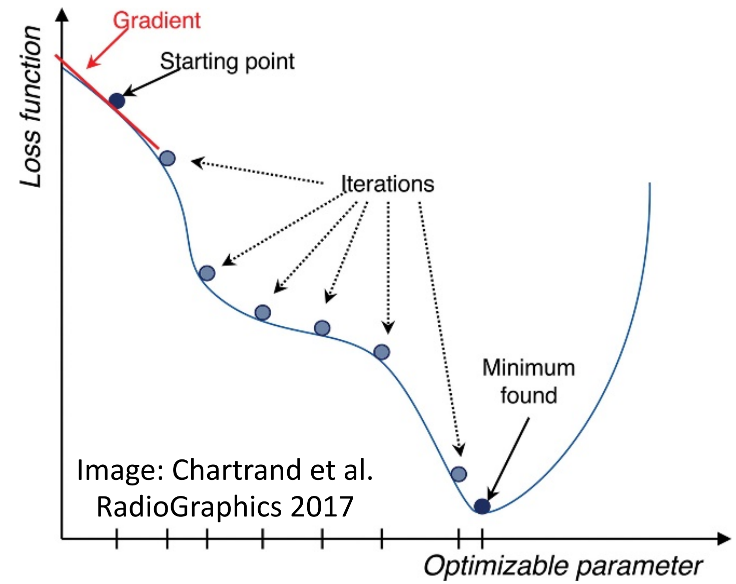


Image: Chartrand et al. RadioGraphics 2017

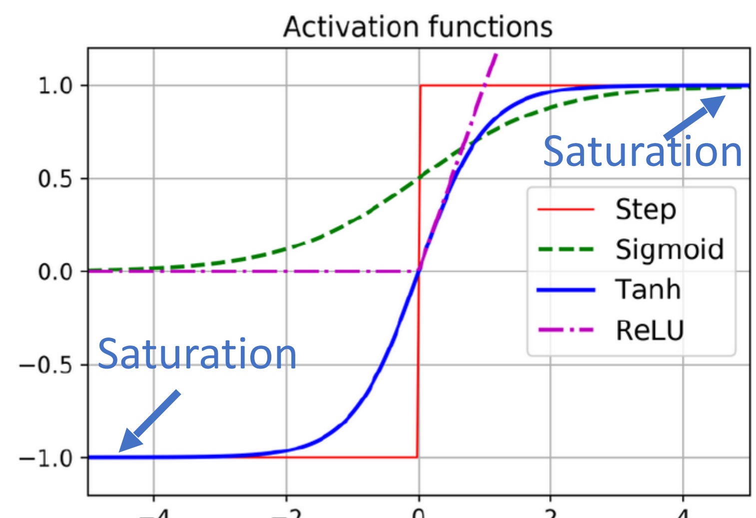
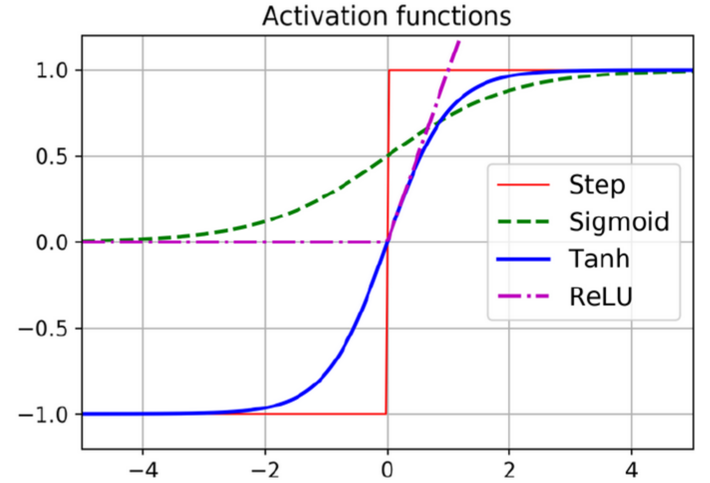


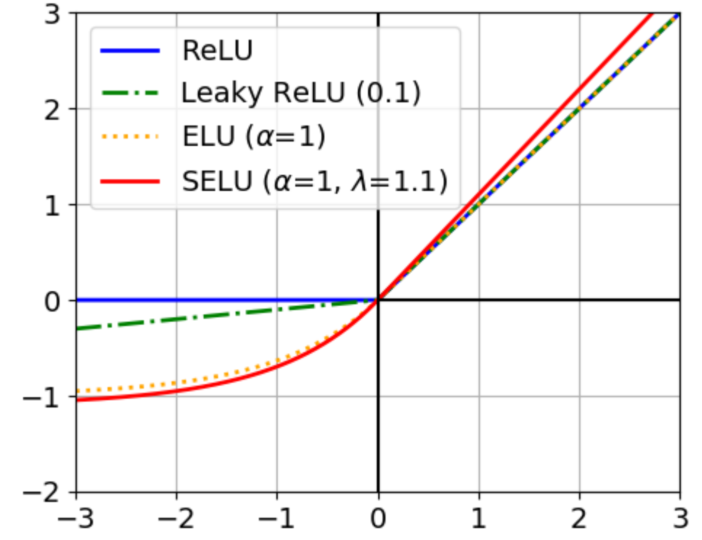
Image: Géron, Hands On ML

Non-saturating Activation Functions

Many options:

- Sigmoid saturates for low/high values
- ReLU: for negative values, becomes 0 and stays 0 (with zero gradient)
- Leaky ReLU: non-saturating, non-dying
- Exponential linear unit (ELU): average closer to 0, converges faster, slower to compute
- Scaled exponential linear unit (SELU): self-normalises dense sequential NNs





ReLU is still popular - as it is simple and encourages sparseness

Be wary of generalisations, as no single thing works best in all situations ==> No Free Lunch theorem

Batch Normalisation

To prevent growing or shrinking gradients through layers…

Add **normalisation layer** before or after each hidden layer that learns optimal mean and scale for each input of a layer

During training:

1. Standardise to mean 0 and standard deviation 1 across current training batch
2. Scale with adjustable parameter $\boldsymbol{\gamma}$
3. Shift with adjustable parameter $\boldsymbol{\beta}$
4. Create moving average across batches

$$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{{\boldsymbol{\sigma}_B}^2 + \varepsilon}}$$

$$\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

Very commonly used and can be highly useful at improving optimisation… but not always.

Gradient Clipping

- Another strategy for managing gradient problems - in particular exploding gradients
- During backpropagation, clip gradients at a threshold

- In recurrent neural networks as an alternative to batch normalisation

- Keras: hyperparameters of optimizer
  - “clipvalue”: absolute value per dimension, orientation changes
  - “clipnorm”: L2 norm clipped, orientation preserved

Initialisation Strategies

- Initialise connection weights randomly with **mean = 0**
- Aim to "statistically" have weights that keep signal variance the same

- Several options (with both normal or uniform distributions) with *recommended matches to activation functions*

- Keras default: Glorot (with uniform distribution)

fan_{in} = number inputs of layer

fan_{out} = number of neurons/outputs

$fan_{avg} = (fan_{in} + fan_{out})/2$

Initia-lisation	Activation functions	Variance σ^2 of normal distr.
Glorot	None, tanh, logistic, softmax	$1/fan_{avg}$
He	ReLU and variants	$2/fan_{in}$
LeCun	SELU	$1/fan_{in}$

Optimisers: Baseline (Stochastic) GD

Gradient Descent (from Lecture 5):

- Partial derivatives of cost function

$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)}\right) x_j^{(i)}$$

- Local gradier

$$\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\mathbf{X}^\top(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

- Iteratively step downhill

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta\,\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}(\boldsymbol{\theta})$$

Weight vector **θ**

Learning rate “eta” η

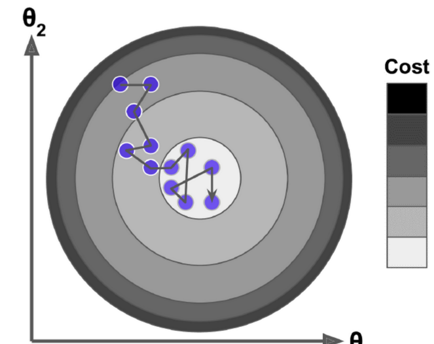


Image: Géron, Hands On ML

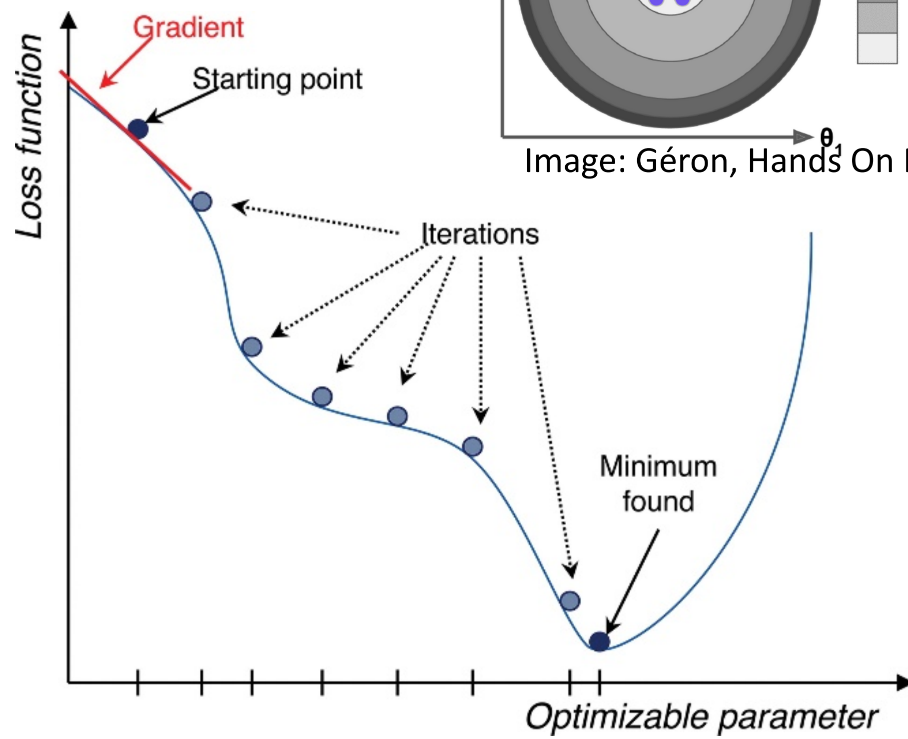


Image: Chartrand et al. RadioGraphics 2017

Optimisers: Baseline (Stochastic) GD

Gradient Descent (from Lecture 5):

- Partial derivatives of cost functi

$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

- Local gradier

$$\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\mathbf{X}^\mathsf{T}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

- Iteratively step downhill

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta\,\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}(\boldsymbol{\theta})$$

Weight vector θ

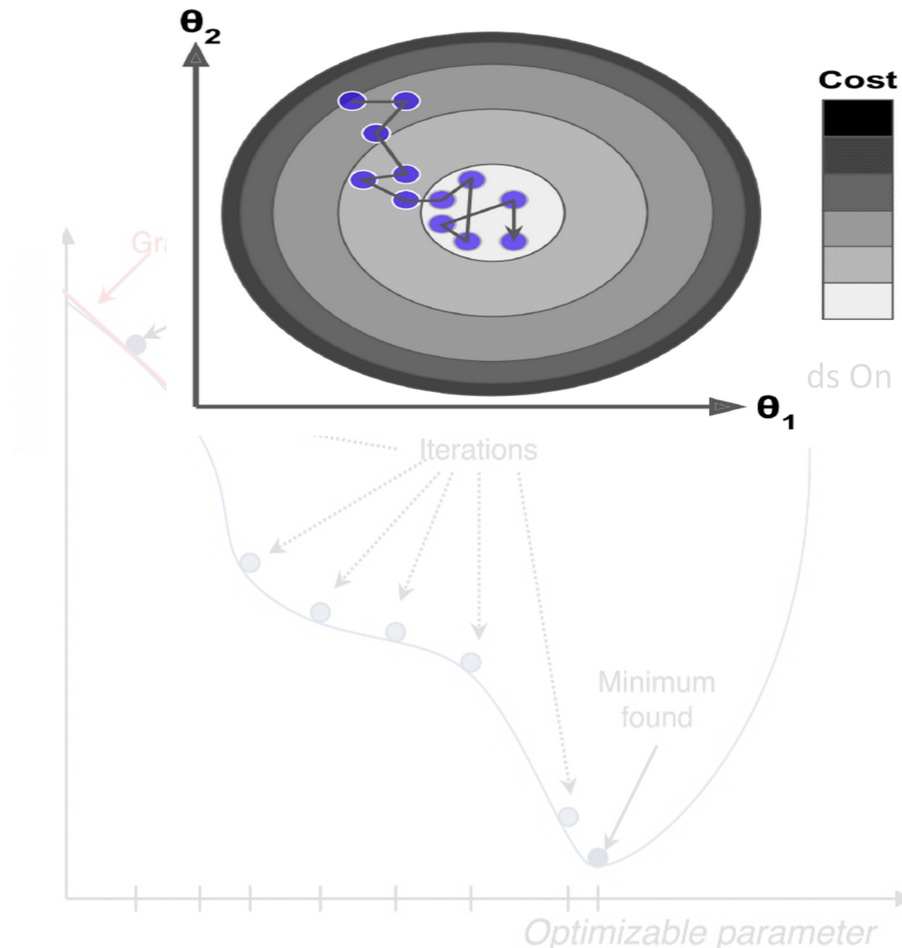


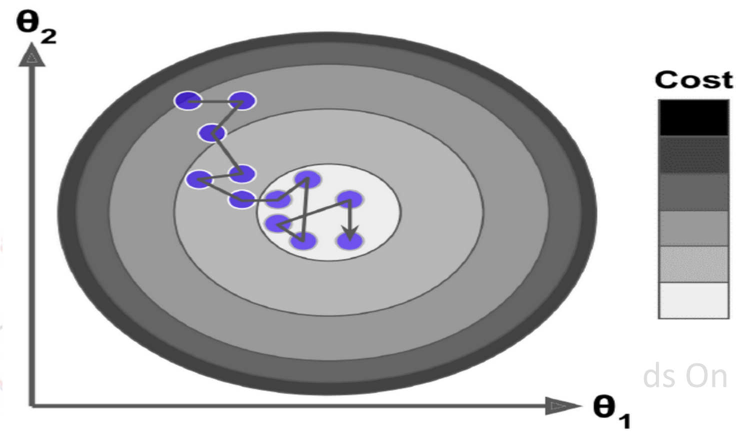
Image: Chartrand et al. RadioGraphics 2017

Optimisers: Baseline (Stochastic) GD

Gradient Descent (from Lecture 5):

Idea: follow the gradient downhill

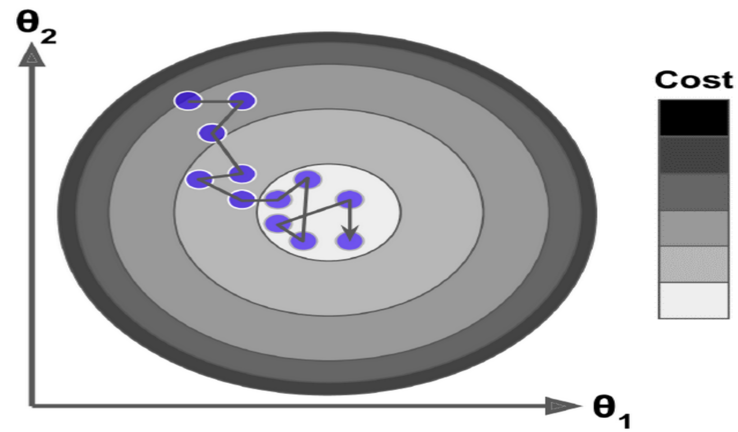
- need to pick a step size
- not always the most efficient
- variations exist to improve on this
- often uses physics for inspiration



Optimisers: Baseline (Stochastic) GD

Options are:

- Momentum Optimisations
  - Partially follow previous direction
  - Gradient = force, not displacement
- Nesterov Accelerated Gradient
  - Use gradient *at projected location*
- AdaGrad (Adaptive Subgrad. Opt.)
  - Scale gradient with decaying value
- RMSprop
  - Different scaling/decay
- Adam (Adaptive Momentum Est.)
  - Combine momentum and scaling
- Nadam
  - Adam with Nesterov calculation
- AdaMax
  - Use max rather than adding terms



Optimiser Comparison (Géron 2019)

Trade-off between:

- Speed
- Convergence quality
- Number of hyperparameters
- Assumptions about cost function landscape

✳ Based on empirical tests

✳ May not apply in all cases

✳ Can change as new approaches emerge

Optimizer (Keras)	Convergence speed	Convergence quality
SGD	*	***
Momentum	**	***
Nesterov	**	***
Adagrad	***	* (can stop too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

* bad, ** average, *** good

Early Stopping: Shorter Runtimes

- Stop if no improvement for X iterations ("patience")
- Tolerance ("min_delta")
- Implemented as a *Callback*

```
tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0,
    patience=0,
    verbose=0,
    mode="auto",
    baseline=None,
    restore_best_weights=False,
)
```

https://keras.io/api/callbacks/early_stopping/

Early Stopping: Shorter Runtimes

- Stop if no improvement for X iterations ("patience")
- Tolerance ("min_delta")
- Implemented as a *Callback*

```
tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0,
    patience=0,
    verbose=0,
    mode="auto",
    baseline=None,
    restore_best_weights=False,
)
```

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
# This callback will stop the training when there is no improvement in
# the validation loss for three consecutive epochs.
model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
model.compile(tf.keras.optimizers.SGD(), loss='mse')
history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5),
                    epochs=10, batch_size=1, callbacks=[callback],
                    verbose=0)
len(history.history['loss'])  # Only 4 epochs are run.
```

https://keras.io/api/callbacks/early_stopping/

Learning Rate

- Simplest choice: constant learning rate η
- Best learning rate* differs across training, e.g. larger at start, smaller at end
- Rate depends on initialisation, optimizer, its parameters, model, etc., etc.
- Also depends on cost function landscape in high-dimensional weight space, i.e. the data

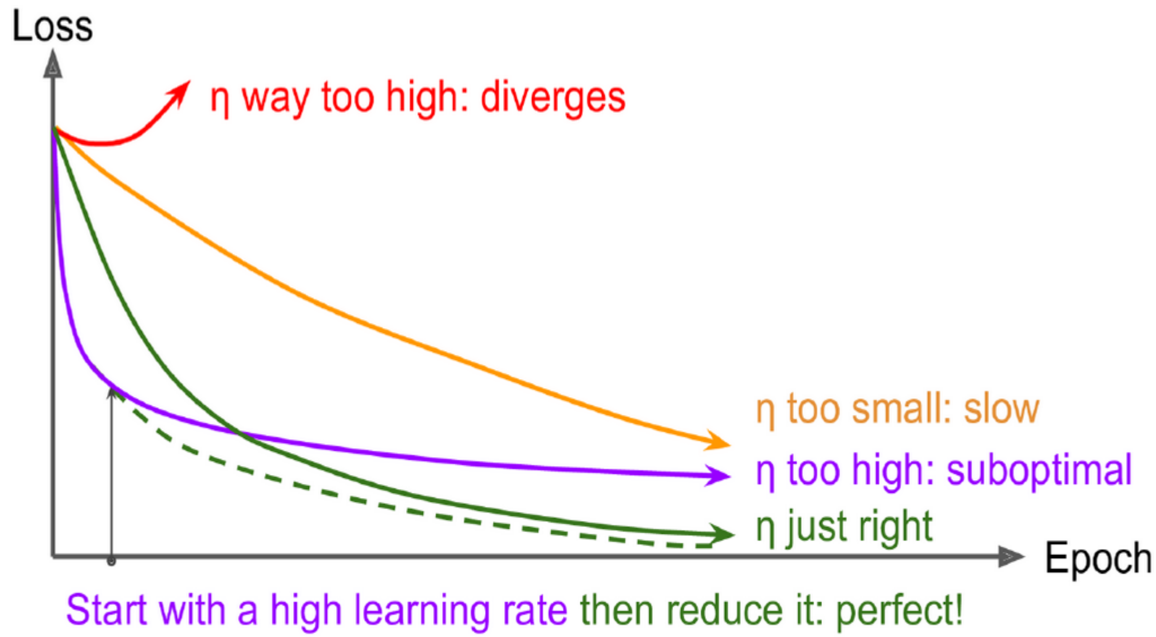


Image: Géron, Hands On ML

* As with most things, this is not always true

Learning Rate Scheduling

- Change learning rate during iterations based on iteration number *t*, error, or a test on data
- Very empirically based, often trial and error, as many things affect it

Learning Rate Scheduling

- Change learning rate during iterations based on iteration number *t*, error, or a test on data

- Very empirically based, often trial and error, as many things affect it

- Piecewise constant:

$$\eta(t) = \begin{cases} 1-5 & 0.1 \\ 6-10 & 0.01 \\ >10 & 0.001 \end{cases}$$

- Power scheduling:

$$\eta(t) = \frac{\eta_0}{\left(1+\frac{t}{s}\right)^c}$$

Initial learning rate η_0

Power c (e.g. 1)

Step

- Exponential scheduling:

$$\eta(t) = \eta_0 0.1^{\frac{t}{s}}$$

- Performance scheduling:
  - E.g. if validation error not decreasing, reduce learning rate by factor

- Implementation in Keras/ tk.keras:
  - Built-in parameter of optimizer
  - Callback in model (LearningRateScheduler)
  - Schedule object of tk.keras in optimizer

1cycle Approach (Smith 2018)

Cyclical Learning:

- Start with minimum learning rate (LR)
- Increase to a maximum LR
- Decrease back to minimum LR
- Repeat

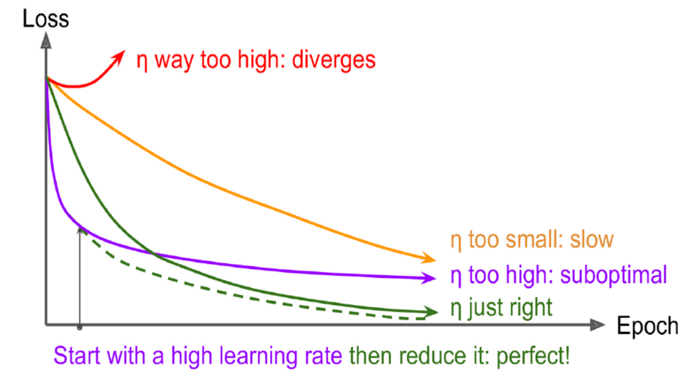


Image: Géron, Hands On ML

Smith LN 2018 A disciplined approach to neural network hyper-parameters: Part 1 – Learning rate, batch size, momentum, and weight decay. US Naval Research Laboratory Technical Report 5510-026.

1cycle Approach (Smith 2018)

Cyclical Learning:

- Start with minimum learning rate (LR)
- Increase to a maximum LR
- Decrease back to minimum LR
- Repeat

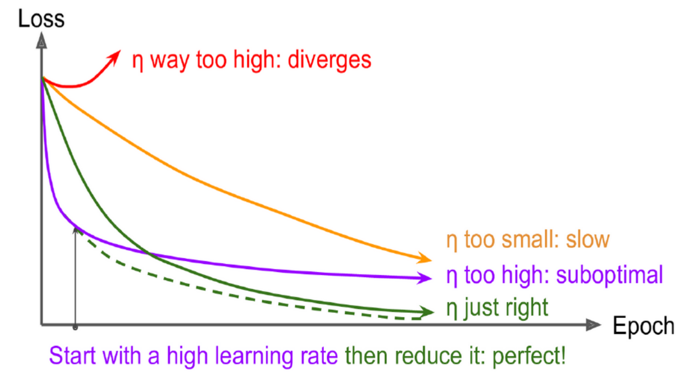


Image: Géron, Hands On ML

“1cycle” scheduling (Smith 2018)

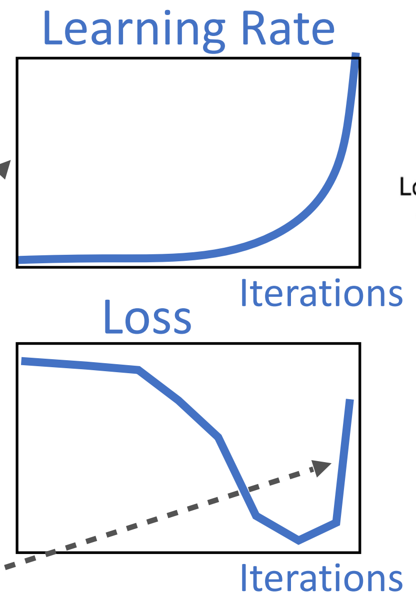
- Initial LR range test:
  - Run with increasing LR (from very small value) until training starts to diverge (when error goes up) => maximum LR
  - For cycle set: minimum LR = maximum LR / 10

Smith LN 2018 A disciplined approach to neural network hyper-parameters: Part 1 – Learning rate, batch size, momentum, and weight decay. US Naval Research Laboratory Technical Report 5510-026.

1cycle Approach (Smith 2018)

Cyclical Learning:

- Start with minimum learning rate (LR)
- Increase to a maximum LR
- Decrease back to minimum LR
- Repeat



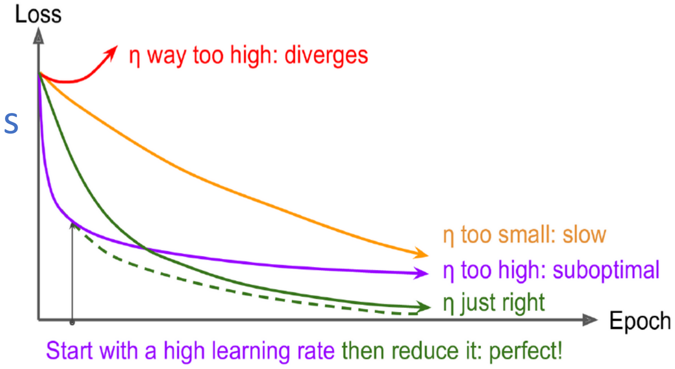


Image: Géron, Hands On ML

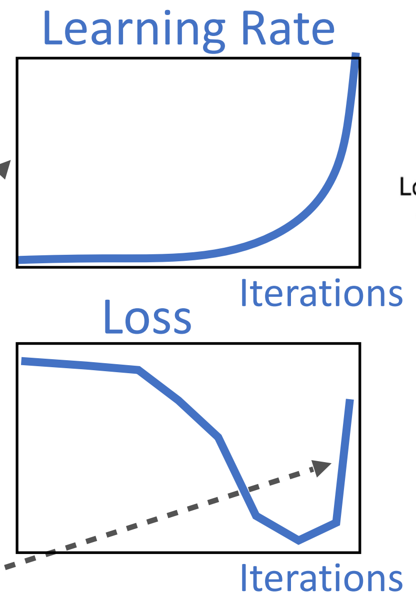
"1cycle" scheduling (Smith 2018)

- Initial LR range test:
  - Run with increasing LR (from very small value) until training starts to diverge (when error goes up) => maximum LR
  - For cycle set: minimum LR = maximum LR / 10
- Use 1 cycle with linear increase/decrease of LR across most epochs
- After that drop LR linearly to very small value

Smith LN 2018 A disciplined approach to neural network hyper-parameters: Part 1 – Learning rate, batch size, momentum, and weight decay. US Naval Research Laboratory Technical Report 5510-026.

1cycle Approach (Smith 2018)

Cyclical Learning:
- Start with minimum learning rate (LR)
- Increase to a maximum LR
- Decrease back to minimum LR
- Repeat



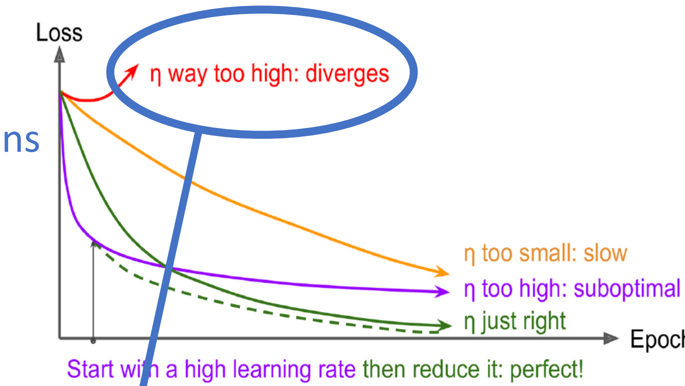
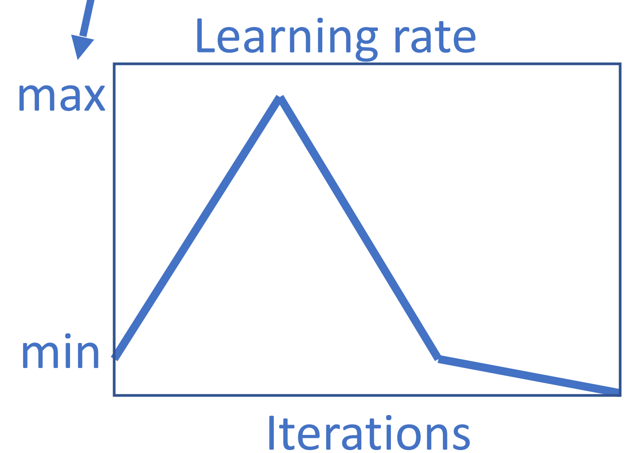


Image: Géron, Hands On ML

“1cycle” scheduling (Smith 2018)
- Initial LR range test:
  - Run with increasing LR (from very small value) until training starts to diverge (when error goes up) => maximum LR
  - For cycle set: minimum LR = maximum LR / 10
- Use 1 cycle with linear increase/decrease of LR across most epochs
- After that drop LR linearly to very small value



Smith LN 2018 A disciplined approach to neural network hyper-parameters: Part 1 – Learning rate, batch size, momentum, and weight decay. US Naval Research Laboratory Technical Report 5510-026.

Recommendations in Practice (Géron 2019)

Hyperparameter	Dense NN default	Self-normalising DNN default
Kernel initialiser	He ($\sigma^2 = 2/fan_{in}$)	LeCun ($\sigma^2 = 1/fan_{in}$)
Activation Function	ELU	SELU
Normalisation	None if shallow, batch normalisation if deep	None (self-normalising)
Regularisation	Early stopping (+ L2 norm regularisation if needed)	Alpha drop out if needed
Optimiser	Momentum (or RMSProp or Nadam)	Momentum (or RMSProp or Nadam)
Learning rate schedule	1cycle	1cycle

- Usually based on limited empirical tests and don't apply to all situations
- Field and libraries under active development, recommendations may change!

Summary

- Training deep NNs uses gradient descent
- Backpropagation used to compute the gradient sequentially
- Prevent vanishing and exploding gradients with
  - Initialisation
  - Non-saturating activation functions
  - Batch normalisation layers
  - Gradient clipping
- Optimisers have been incrementally refined
  - Use the most “patched” version of an approach (e.g. Nadam)
  - If problems, try alternative approach (e.g. Nesterov Momentum)
- Learning rate scheduling and 1cycle current default