

# Differentiate Almost Everywhere

# Differentiable Relaxations and Reparameterisations

Jonathon Hare

Vision, Learning and Control  
University of Southampton

# What are differentiable relaxations and reparameterisations?

- We've seen that we can build arbitrary computational graphs from a variety of building blocks

# What are differentiable relaxations and reparameterisations?

- We've seen that we can build arbitrary computational graphs from a variety of building blocks
- But, those blocks need to be differentiable to work in our optimisation framework
  - More specifically they need to be continuous and *differentiable almost everywhere*.

# What are differentiable relaxations and reparameterisations?

- We've seen that we can build arbitrary computational graphs from a variety of building blocks
- But, those blocks need to be differentiable to work in our optimisation framework
  - More specifically they need to be continuous and *differentiable almost everywhere*.
- That limits what we can do... Can we work around that?
  - Relaxations — make continuous (and potentially differentiable everywhere) approximations.
  - Reparameterisations — rewrite functions to factor out stochastic variables from the parameters.

## Aside: continuity and differentiable almost everywhere

- Consider the ReLU function  $f(x) = \max(0, x)$

## Aside: continuity and differentiable almost everywhere

- Consider the ReLU function  $f(x) = \max(0, x)$ 
  - ReLU is *continuous*
    - it does not have any abrupt changes in value
    - small changes in  $x$  result in small changes to  $f(x)$  everywhere in the domain of  $x$

## Aside: continuity and differentiable almost everywhere

- Consider the ReLU function  $f(x) = \max(0, x)$ 
  - ReLU is *continuous*
    - it does not have any abrupt changes in value
    - small changes in  $x$  result in small changes to  $f(x)$  everywhere in the domain of  $x$
  - ReLU is *differentiable almost everywhere*
    - No gradient at  $x = 0$ ; only *left* and *right* gradients at that point
    - There are *subgradients* at  $x = 0$ ; implementations usually just arbitrarily pick  $f'(0) = 0$

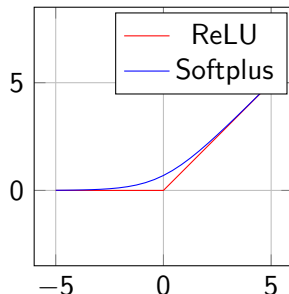


## Aside: continuity and differentiable almost everywhere

- Consider the ReLU function  $f(x) = \max(0, x)$ 
  - ReLU is *continuous*
    - it does not have any abrupt changes in value
    - small changes in  $x$  result in small changes to  $f(x)$  everywhere in the domain of  $x$
  - ReLU is *differentiable almost everywhere*
    - No gradient at  $x = 0$ ; only *left* and *right* gradients at that point
    - There are *subgradients* at  $x = 0$ ; implementations usually just arbitrarily pick  $f'(0) = 0$
- Functions that are differentiable almost everywhere or have subgradients tend to be compatible with gradient descent methods
  - We expect that the loss landscape is different for each batch & that we'll never actually reach a minima, and we only need to *mostly* take steps in the right direction.

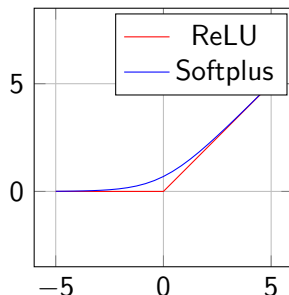
# Relaxing ReLU

- Softplus ( $\text{softplus}(x) = \ln(1 + e^x)$ ) is a relaxation of ReLU that is *differentiable everywhere*.



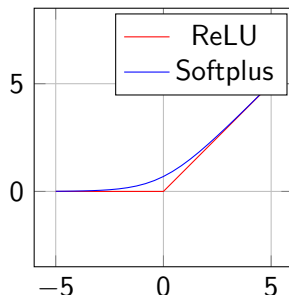
# Relaxing ReLU

- Softplus ( $\text{softplus}(x) = \ln(1 + e^x)$ ) is a relaxation of ReLU that is *differentiable everywhere*.
- Its derivative is the Sigmoid function



# Relaxing ReLU

- Softplus ( $\text{softplus}(x) = \ln(1 + e^x)$ ) is a relaxation of ReLU that is *differentiable everywhere*.
- Its derivative is the Sigmoid function
- Not widely used; counter-intuitively, even though it neither saturates completely and is differentiable everywhere, empirically it has been shown that ReLU works better.



# Interpretations of softmax

- Up until now we've really considered softmax as a generalisation of sigmoid (which represents a probability distribution over a binary variable) to many output categories.
  - softmax transforms a vector of logits into a probability distribution over categories.

# Interpretations of softmax

- Up until now we've really considered softmax as a generalisation of sigmoid (which represents a probability distribution over a binary variable) to many output categories.
  - softmax transforms a vector of logits into a probability distribution over categories.
- As you might guess from the name, softmax is a relaxation...

# Interpretations of softmax

- Up until now we've really considered softmax as a generalisation of sigmoid (which represents a probability distribution over a binary variable) to many output categories.
  - softmax transforms a vector of logits into a probability distribution over categories.
- As you might guess from the name, softmax is a relaxation...
  - but not of the max function like the name would suggest!

# Interpretations of softmax

- Up until now we've really considered softmax as a generalisation of sigmoid (which represents a probability distribution over a binary variable) to many output categories.
  - softmax transforms a vector of logits into a probability distribution over categories.
- As you might guess from the name, softmax is a relaxation...
  - but not of the max function like the name would suggest!
  - softmax can be viewed as a continuous and differentiable relaxation of the arg max function with one-hot output encoding.



# Interpretations of softmax

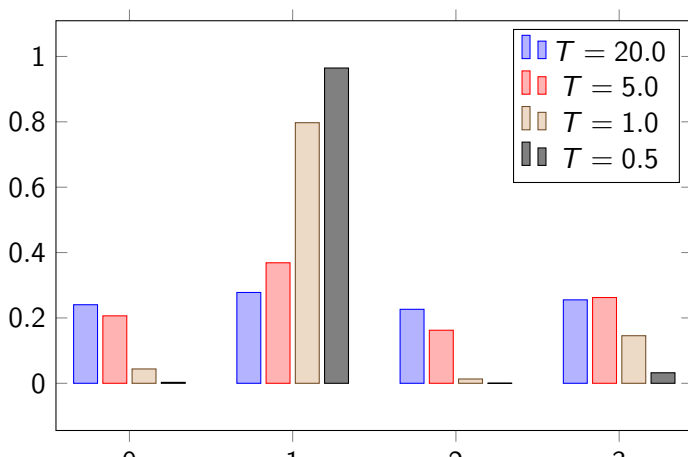
- Up until now we've really considered softmax as a generalisation of sigmoid (which represents a probability distribution over a binary variable) to many output categories.
  - softmax transforms a vector of logits into a probability distribution over categories.
- As you might guess from the name, softmax is a relaxation...
  - but not of the max function like the name would suggest!
  - softmax can be viewed as a continuous and differentiable relaxation of the arg max function with one-hot output encoding.
  - The arg max function is not continuous or differentiable; softmax provides an approximation:

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} 1.1 & 4.0 & -0.1 & 2.3 \end{bmatrix} \\ \arg \max(\mathbf{x}) &= \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \\ \text{softmax}(\mathbf{x}) &= \begin{bmatrix} 0.044 & 0.797 & 0.013 & 0.146 \end{bmatrix} \end{aligned}$$

# The Softmax function with temperature

Consider what happens if you were to divide the input logits to a softmax by a scalar temperature parameter  $T$ .

$$\text{softmax}(\mathbf{x}/T)_i = \frac{e^{x_i/T}}{\sum_{j=1}^K e^{x_j/T}} \quad \forall i = 1, 2, \dots, K$$



## arg max — softmax with temperature

$\mathbf{x} =$	[	1.1	4.0	-0.1	2.3	]
$\text{softmax}(\mathbf{x}/1.0) =$	[	0.044	0.797	0.013	0.146	]
$\text{softmax}(\mathbf{x}/0.8) =$	[	0.023	0.868	0.005	0.104	]
$\text{softmax}(\mathbf{x}/0.6) =$	[	0.008	0.937	0.001	0.055	]
$\text{softmax}(\mathbf{x}/0.4) =$	[	6.997e-04	9.852e-01	3.484e-05	1.405e-02	]
$\text{softmax}(\mathbf{x}/0.2) =$	[	5.042e-07	9.998e-01	1.250e-09	2.034e-04	]

# arg max — scalar approximation

- What if you want to get a scalar approximation to the index of the arg max rather than a probability distribution approximating the one-hot form?
  - Caveat: we are not actually going to get a guaranteed integer representation as that would be non-differentiable; we'll have to live with a float that is an approximation<sup>1</sup>.

---

<sup>1</sup>for now — we'll address this in a few slides time!

# arg max — scalar approximation

- What if you want to get a scalar approximation to the index of the arg max rather than a probability distribution approximating the one-hot form?
  - Caveat: we are not actually going to get a guaranteed integer representation as that would be non-differentiable; we'll have to live with a float that is an approximation<sup>1</sup>.
- First, consider how to convert a one-hot vector to index representation in a differentiable manner:  $[0, 0, 1, 0] \rightarrow 2$ 
  - Just dot product with a vector of indices:  $[0, 1, 2, 3]$

---

<sup>1</sup>for now — we'll address this in a few slides time!

## arg max — scalar approximation

- What if you want to get a scalar approximation to the index of the arg max rather than a probability distribution approximating the one-hot form?
  - Caveat: we are not actually going to get a guaranteed integer representation as that would be non-differentiable; we'll have to live with a float that is an approximation<sup>1</sup>.
- First, consider how to convert a one-hot vector to index representation in a differentiable manner:  $[0, 0, 1, 0] \rightarrow 2$ 
  - Just dot product with a vector of indices:  $[0, 1, 2, 3]$
- The same process can be applied to the softmax distribution
  - As temperature  $T \rightarrow 0$ ,  $\text{softmax}(\mathbf{x}/T) \cdot [0, 1, \dots, N] \rightarrow \arg \max(\mathbf{x})$  for  $\mathbf{x} \in \mathbb{R}^N$ .

---

<sup>1</sup>for now — we'll address this in a few slides time!

## arg max — scalar approximation

$$\mathbf{x} = [ 1.1 \quad 4.0 \quad -0.1 \quad 2.3 ]^\top$$

$$\mathbf{i} = [ 0.0 \quad 1.0 \quad 2.0 \quad 3.0 ]^\top$$

$$\text{softmax}(\mathbf{x}/1.0)^\top \mathbf{i} = 1.2606$$

$$\text{softmax}(\mathbf{x}/0.8)^\top \mathbf{i} = 1.1894$$

$$\text{softmax}(\mathbf{x}/0.6)^\top \mathbf{i} = 1.1037$$

$$\text{softmax}(\mathbf{x}/0.4)^\top \mathbf{i} = 1.0274$$

$$\text{softmax}(\mathbf{x}/0.2)^\top \mathbf{i} = 1.0004$$

- A similar trick applies to finding the maximum value of a vector:



- A similar trick applies to finding the maximum value of a vector:
  - Use  $\text{softmax}(\mathbf{x})$  as an approximate one-hot arg max, and dot product with the vector  $\mathbf{x}$ .

- A similar trick applies to finding the maximum value of a vector:
  - Use  $\text{softmax}(\mathbf{x})$  as an approximate one-hot arg max, and dot product with the vector  $\mathbf{x}$ .
  - As temperature  $T \rightarrow 0$ ,  $\text{softmax}(\mathbf{x}/T)^\top \mathbf{x} \rightarrow \max(\mathbf{x})$ .

- A similar trick applies to finding the maximum value of a vector:
  - Use  $\text{softmax}(\mathbf{x})$  as an approximate one-hot arg max, and dot product with the vector  $\mathbf{x}$ .
  - As temperature  $T \rightarrow 0$ ,  $\text{softmax}(\mathbf{x}/T)^\top \mathbf{x} \rightarrow \max(\mathbf{x})$ .

$$\mathbf{x} = [ 1.1 \quad 4.0 \quad -0.1 \quad 2.3 ]^\top$$

$$\text{softmax}(\mathbf{x}/1.0)^\top \mathbf{x} = 3.571$$

$$\text{softmax}(\mathbf{x}/0.8)^\top \mathbf{x} = 3.736$$

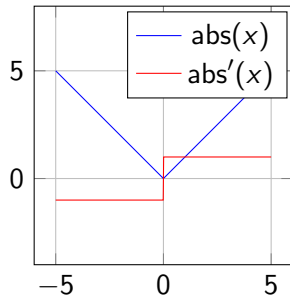
$$\text{softmax}(\mathbf{x}/0.6)^\top \mathbf{x} = 3.881$$

$$\text{softmax}(\mathbf{x}/0.4)^\top \mathbf{x} = 3.974$$

$$\text{softmax}(\mathbf{x}/0.2)^\top \mathbf{x} = 3.999$$

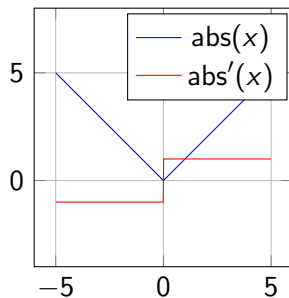
# L1 norm

- L1 norm is the sum of absolute values of a vector



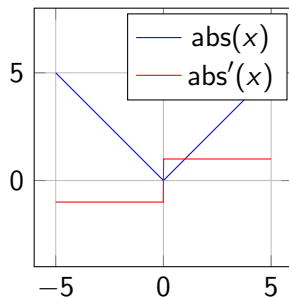
# L1 norm

- L1 norm is the sum of absolute values of a vector
- We've seen that an L1 norm regulariser can induce sparsity in a model



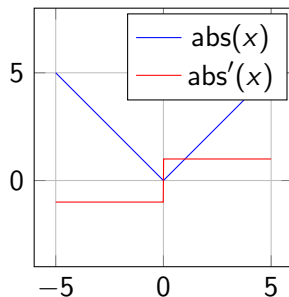
# L1 norm

- L1 norm is the sum of absolute values of a vector
- We've seen that an L1 norm regulariser can induce sparsity in a model
- $\text{abs}$  is continuous and differentiable almost everywhere, but...



# L1 norm

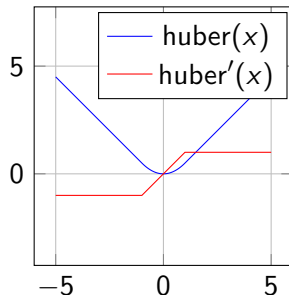
- L1 norm is the sum of absolute values of a vector
- We've seen that an L1 norm regulariser can induce sparsity in a model
- $\text{abs}$  is continuous and differentiable almost everywhere, but...
- unlike ReLU, the gradients left and right of the discontinuity point in equal and opposite directions
  - This can cause oscillations that prevent or hamper learning



# Relaxing the L1 norm

- Huber loss (aka Smooth L1 loss) relaxes L1 by mixing it with L2 near the origin:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$



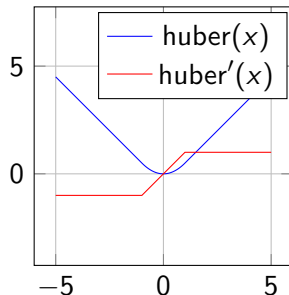


# Relaxing the L1 norm

- Huber loss (aka Smooth L1 loss) relaxes L1 by mixing it with L2 near the origin:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

- In both cases gradients reduce in magnitude and switch direction smoothly which can lead to much less oscillation.



# Backpropagation through random operations

- Up until now all the models we've considered have performed deterministic transformations of input variables  $\mathbf{x}$ .

# Backpropagation through random operations

- Up until now all the models we've considered have performed deterministic transformations of input variables  $\mathbf{x}$ .
- What if we want to build a model that performs a stochastic transformation of  $\mathbf{x}$ ?

# Backpropagation through random operations

- Up until now all the models we've considered have performed deterministic transformations of input variables  $\mathbf{x}$ .
- What if we want to build a model that performs a stochastic transformation of  $\mathbf{x}$ ?
- A simple way to do this is to augment the input  $\mathbf{x}$  with a random vector  $\mathbf{z}$  sampled from some distribution

# Backpropagation through random operations

- Up until now all the models we've considered have performed deterministic transformations of input variables  $\mathbf{x}$ .
- What if we want to build a model that performs a stochastic transformation of  $\mathbf{x}$ ?
- A simple way to do this is to augment the input  $\mathbf{x}$  with a random vector  $\mathbf{z}$  sampled from some distribution
  - The network would learn a function  $f(\mathbf{x}, \mathbf{z})$  that is internally deterministic, but appears stochastic to an observer that does not have access to  $\mathbf{z}$ .

# Backpropagation through random operations

- Up until now all the models we've considered have performed deterministic transformations of input variables  $\mathbf{x}$ .
- What if we want to build a model that performs a stochastic transformation of  $\mathbf{x}$ ?
- A simple way to do this is to augment the input  $\mathbf{x}$  with a random vector  $\mathbf{z}$  sampled from some distribution
  - The network would learn a function  $f(\mathbf{x}, \mathbf{z})$  that is internally deterministic, but appears stochastic to an observer that does not have access to  $\mathbf{z}$ .
  - provided that  $f$  is continuous and differentiable (almost everywhere) we can perform gradient based optimisation as usual.

Consider

$$y \sim \mathcal{N}(\mu, \sigma^2)$$

How can we take derivatives of  $y$  with respect to  $\mu$  and  $\sigma^2$ ?

If we rewrite

$$y = \mu + \sigma z \text{ where } z = \mathcal{N}(0, 1)$$

Then it is clear that  $y$  is a function of a deterministic operation with variables  $\mu$  and  $\sigma$  with an (extra) input  $z$ .

- Crucially the extra input is an r.v. whose distribution is not a function of any variables whose derivatives we wish to calculate.



If we rewrite

$$y = \mu + \sigma z \text{ where } z = \mathcal{N}(0, 1)$$

Then it is clear that  $y$  is a function of a deterministic operation with variables  $\mu$  and  $\sigma$  with an (extra) input  $z$ .

- Crucially the extra input is an r.v. whose distribution is not a function of any variables whose derivatives we wish to calculate.
- The derivatives  $dy/d\mu$  and  $dy/d\sigma$  tell us how an infinitesimal change in  $\mu$  or  $\sigma$  would change  $y$  if we could repeat the sampling operation with the *same* value of  $z$

# The reparameterisation trick

- The 'trick' of factoring out the source of randomness into an extra input  $z$  is often called the **reparameterisation trick**.

# The reparameterisation trick

- The 'trick' of factoring out the source of randomness into an extra input  $z$  is often called the **reparameterisation trick**.
- It doesn't just apply to the Gaussian distribution!

# The reparameterisation trick

- The 'trick' of factoring out the source of randomness into an extra input  $z$  is often called the **reparameterisation trick**.
- It doesn't just apply to the Gaussian distribution!
  - More generally we can express any probability distribution  $p(y; \theta)$  or  $p(y|\mathbf{x}; \theta)$  as  $p(y; \omega)$  where  $\omega$  contains the parameters  $\theta$  and if applicable inputs  $\mathbf{x}$ .

# The reparameterisation trick

- The 'trick' of factoring out the source of randomness into an extra input  $z$  is often called the **reparameterisation trick**.
- It doesn't just apply to the Gaussian distribution!
  - More generally we can express any probability distribution  $p(y; \theta)$  or  $p(y|x; \theta)$  as  $p(y; \omega)$  where  $\omega$  contains the parameters  $\theta$  and if applicable inputs  $x$ .
  - A sample  $y \sim p(y; \omega)$  can be rewritten as  $y = f(z, \omega)$  where  $z$  is a source of randomness.

# The reparameterisation trick

- The 'trick' of factoring out the source of randomness into an extra input  $z$  is often called the **reparameterisation trick**.
- It doesn't just apply to the Gaussian distribution!
  - More generally we can express any probability distribution  $p(y; \theta)$  or  $p(y|x; \theta)$  as  $p(y; \omega)$  where  $\omega$  contains the parameters  $\theta$  and if applicable inputs  $x$ .
  - A sample  $y \sim p(y; \omega)$  can be rewritten as  $y = f(z, \omega)$  where  $z$  is a source of randomness.
  - We can thus compute derivatives  $\partial y / \partial \omega$  and use gradient based optimisation as long as
    - $f$  is continuous and differentiable almost everywhere
    - $\omega$  is not a function of  $z$
    - and  $z$  is not a function of  $\omega$

# Backpropagation through discrete stochastic operations

- Consider a stochastic model  $\mathbf{y} = f(\mathbf{z}, \omega)$  where the outputs are **discrete**.

# Backpropagation through discrete stochastic operations

- Consider a stochastic model  $\mathbf{y} = f(\mathbf{z}, \omega)$  where the outputs are **discrete**.
  - This implies  $f$  must be a step function.



# Backpropagation through discrete stochastic operations

- Consider a stochastic model  $\mathbf{y} = f(\mathbf{z}, \omega)$  where the outputs are **discrete**.
  - This implies  $f$  must be a step function.
  - Derivatives of a step function at the step are undefined.

# Backpropagation through discrete stochastic operations

- Consider a stochastic model  $\mathbf{y} = f(\mathbf{z}, \omega)$  where the outputs are **discrete**.
  - This implies  $f$  must be a step function.
  - Derivatives of a step function at the step are undefined.
  - Derivatives are zero almost everywhere.

# Backpropagation through discrete stochastic operations

- Consider a stochastic model  $\mathbf{y} = f(\mathbf{z}, \omega)$  where the outputs are **discrete**.
  - This implies  $f$  must be a step function.
  - Derivatives of a step function at the step are undefined.
  - Derivatives are zero almost everywhere.
  - If we have a loss  $\mathcal{L}(\mathbf{y})$  the gradients don't give us any information on how to update the parameters  $\theta$  to minimise the loss

# Backpropagation through discrete stochastic operations

- Consider a stochastic model  $\mathbf{y} = f(\mathbf{z}, \omega)$  where the outputs are **discrete**.
  - This implies  $f$  must be a step function.
  - Derivatives of a step function at the step are undefined.
  - Derivatives are zero almost everywhere.
  - If we have a loss  $\mathcal{L}(\mathbf{y})$  the gradients don't give us any information on how to update the parameters  $\theta$  to minimise the loss
- Potential solutions:
  - Policy Gradient Methods (e.g. the REINFORCE algorithm)
  - A relaxation and another 'trick': Gumbel Softmax and the Straight-through operator

# REINFORCE: REward Increment = nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility

- $\mathcal{L}(f(\mathbf{z}, \boldsymbol{\omega}))$  has useless derivatives

# REINFORCE: REward Increment = nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility

- $\mathcal{L}(f(\mathbf{z}, \omega))$  has useless derivatives
- But the expected loss  $\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} \mathcal{L}(f(\mathbf{z}, \omega))$  is often smooth and continuous.
  - This is not tractable with high dimensional  $\mathbf{y} = f(\mathbf{z}, \omega)$ .
  - But, it can be estimated without bias using an Monte Carlo average.

# REINFORCE: REward Increment = nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility

- $\mathcal{L}(f(\mathbf{z}, \boldsymbol{\omega}))$  has useless derivatives
- But the expected loss  $\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} \mathcal{L}(f(\mathbf{z}, \boldsymbol{\omega}))$  is often smooth and continuous.
  - This is not tractable with high dimensional  $\mathbf{y} = f(\mathbf{z}, \boldsymbol{\omega})$ .
  - But, it can be estimated without bias using an Monte Carlo average.
- REINFORCE is a family of algorithms that utilise this idea.

# REINFORCE: REward Increment = nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility

The simplest form of REINFORCE is easy to derive by differentiating the expected loss:

$$\mathbb{E}_{\mathbf{z}}[\mathcal{L}(\mathbf{y})] = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \quad (1)$$

$$\frac{\partial \mathbb{E}[\mathcal{L}(\mathbf{y})]}{\partial \boldsymbol{\omega}} = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) \frac{\partial p(\mathbf{y})}{\partial \boldsymbol{\omega}} \quad (2)$$

$$= \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \boldsymbol{\omega}} \quad (3)$$

$$\approx \frac{1}{m} \sum_{\mathbf{y}^{(i)} \sim p(\mathbf{y}), i=1}^m \mathcal{L}(\mathbf{y}^{(i)}) \frac{\partial \log p(\mathbf{y}^{(i)})}{\partial \boldsymbol{\omega}} \quad (4)$$

- This gives us an unbiased MC estimator of the gradient.



# REINFORCE: REward Increment = nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility

The simplest form of REINFORCE is easy to derive by differentiating the expected loss:

$$\mathbb{E}_{\mathbf{z}}[\mathcal{L}(\mathbf{y})] = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \quad (1)$$

$$\frac{\partial \mathbb{E}[\mathcal{L}(\mathbf{y})]}{\partial \boldsymbol{\omega}} = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) \frac{\partial p(\mathbf{y})}{\partial \boldsymbol{\omega}} \quad (2)$$

$$= \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \boldsymbol{\omega}} \quad (3)$$

$$\approx \frac{1}{m} \sum_{\mathbf{y}^{(i)} \sim p(\mathbf{y}), i=1}^m \mathcal{L}(\mathbf{y}^{(i)}) \frac{\partial \log p(\mathbf{y}^{(i)})}{\partial \boldsymbol{\omega}} \quad (4)$$

- This gives us an unbiased MC estimator of the gradient.
- Unfortunately this is a very high variance estimator, so it would require many samples of  $\mathbf{y}$  to be drawn to obtain a good estimate

# REINFORCE: REward Increment = nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility

The simplest form of REINFORCE is easy to derive by differentiating the expected loss:

$$\mathbb{E}_{\mathbf{z}}[\mathcal{L}(\mathbf{y})] = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \quad (1)$$

$$\frac{\partial \mathbb{E}[\mathcal{L}(\mathbf{y})]}{\partial \boldsymbol{\omega}} = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) \frac{\partial p(\mathbf{y})}{\partial \boldsymbol{\omega}} \quad (2)$$

$$= \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \boldsymbol{\omega}} \quad (3)$$

$$\approx \frac{1}{m} \sum_{\mathbf{y}^{(i)} \sim p(\mathbf{y}), i=1}^m \mathcal{L}(\mathbf{y}^{(i)}) \frac{\partial \log p(\mathbf{y}^{(i)})}{\partial \boldsymbol{\omega}} \quad (4)$$

- This gives us an unbiased MC estimator of the gradient.
- Unfortunately this is a very high variance estimator, so it would require many samples of  $\mathbf{y}$  to be drawn to obtain a good estimate
  - or equivalently, if only one sample were drawn, SGD would converge very slowly and **require** a small learning rate.

# Sampling from a categorical distribution: Gumbel Softmax

The generation of a discrete token,  $t$ , from a vocabulary of  $K$  tokens is achieved by sampling a categorical distribution

$$t \sim \text{Cat}(p_1, \dots, p_K); \sum_i p_i = 1.$$

# Sampling from a categorical distribution: Gumbel Softmax

The generation of a discrete token,  $t$ , from a vocabulary of  $K$  tokens is achieved by sampling a categorical distribution

$$t \sim \text{Cat}(p_1, \dots, p_K); \sum_i p_i = 1.$$

Generating the probabilities  $p_1, \dots, p_K$  directly from a neural network has potential numerical problems; it's much easier to generate logits,  $x_1, \dots, x_K$ .

# Sampling from a categorical distribution: Gumbel Softmax

The generation of a discrete token,  $t$ , from a vocabulary of  $K$  tokens is achieved by sampling a categorical distribution

$$t \sim \text{Cat}(p_1, \dots, p_K); \sum_i p_i = 1.$$

Generating the probabilities  $p_1, \dots, p_K$  directly from a neural network has potential numerical problems; it's much easier to generate logits,  $x_1, \dots, x_K$ .

The gumbel-softmax reparameterisation allows us to sample directly using the logits:

$$t = \underset{i \in \{1, \dots, K\}}{\operatorname{argmax}} x_i + z_i$$

where  $z_1, \dots, z_K$  are i.i.d Gumbel(0,1) variates which can be computed from Uniform variates through  $-\log(-\log(\mathcal{U}(0,1)))$ .

# Differentiable Sampling: Straight-Through Gumbel Softmax

Ok, but how does that help?  $\text{argmax}$  isn't differentiable!

# Differentiable Sampling: Straight-Through Gumbel Softmax

Ok, but how does that help?  $\text{argmax}$  isn't differentiable!  
...but we've already seen that we can relax  $\text{arg max}$  using

$$\text{softargmax}(\mathbf{y}) = \sum_i \frac{e^{y_i/T}}{\sum_j e^{y_j/T}} i$$

where  $T$  is the temperature parameter.

# Differentiable Sampling: Straight-Through Gumbel Softmax

But... this clearly gives us a result that will be non-integer; we cannot round or clip because it would be non-differentiable.



# Differentiable Sampling: Straight-Through Gumbel Softmax

But... this clearly gives us a result that will be non-integer; we cannot round or clip because it would be non-differentiable.

The Straight-Through operator allows us to take the result of a true argmax that has the gradient of the softargmax:

$$\text{STargmax}(\mathbf{y}) = \text{softargmax}(\mathbf{y}) + \text{stopgradient}(\text{argmax}(\mathbf{y}) - \text{softargmax}(\mathbf{y}))$$

where stopgradient is defined such that  $\text{stopgradient}(\mathbf{a}) = \mathbf{a}$  and  $\nabla \text{stopgradient}(\mathbf{a}) = 0$ .

# Differentiable Sampling: Straight-Through Gumbel Softmax

But... this clearly gives us a result that will be non-integer; we cannot round or clip because it would be non-differentiable.

The Straight-Through operator allows us to take the result of a true  $\text{argmax}$  that has the gradient of the  $\text{softargmax}$ :

$$\text{STargmax}(\mathbf{y}) = \text{softargmax}(\mathbf{y}) + \text{stopgradient}(\text{argmax}(\mathbf{y}) - \text{softargmax}(\mathbf{y}))$$

where  $\text{stopgradient}$  is defined such that  $\text{stopgradient}(\mathbf{a}) = \mathbf{a}$  and  $\nabla \text{stopgradient}(\mathbf{a}) = 0$ .

## Straight-Through Gumbel Softmax

Combine the gumbel softmax trick with the  $\text{STargmax}$  to give you discrete samples, with a usable gradient<sup>a</sup>.

---

<sup>a</sup>The ST operator is biased but low variance; in practice it works very well and is better than the high-variance unbiased estimates you could get through REINFORCE.

- Differentiable programming works with functions that are continuous and differentiable almost everywhere.
- Some non-continuous functions can be relaxed to make them more amenable to gradient based optimisation by making continuous approximations.
- Some continuous functions with discontinuous gradients can be relaxed to make optimisation more stable.
- Reparameterisations can allow us to differentiate through random operations such as sampling
- We can even make networks output/utilise discrete variables by combining relaxations and reparameterisations.