Text Processing using Machine Learning

Machine Learning Foundations + Nuts and Bolts

Liling Tan

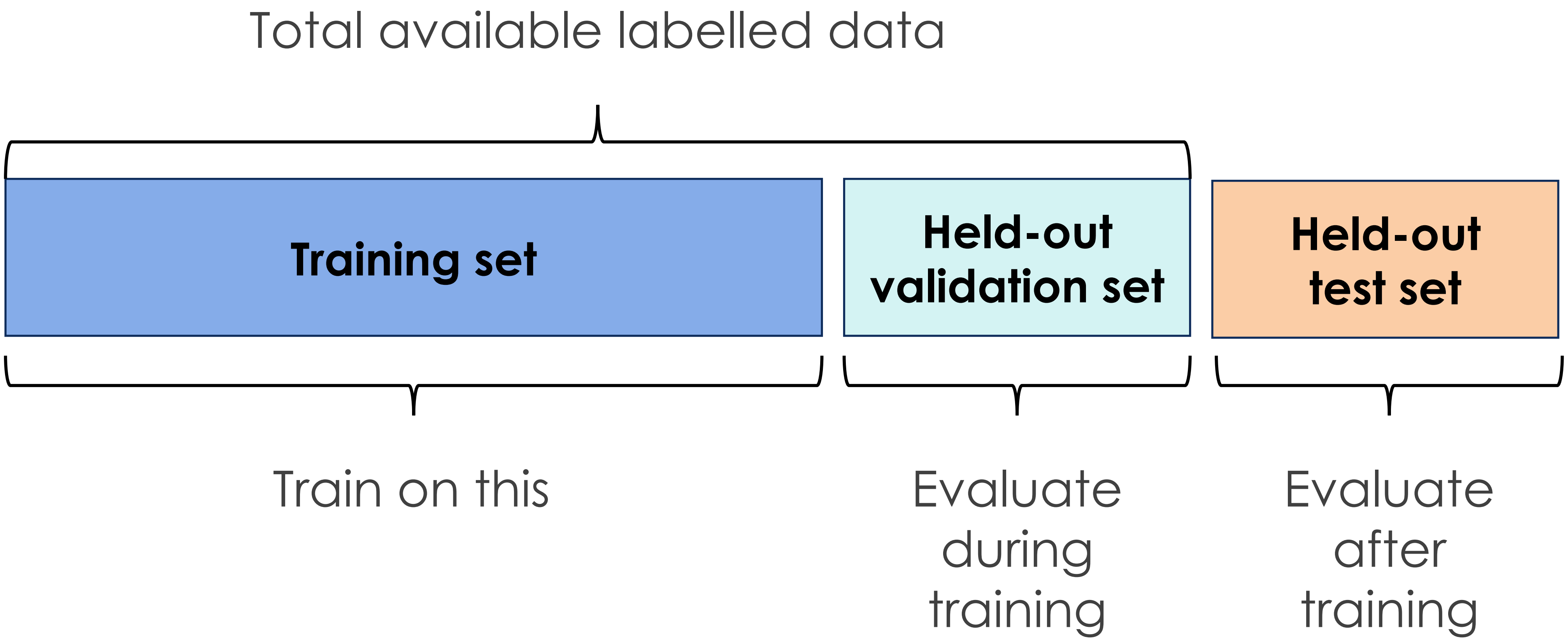
03 Dec 2019

Overview

- **Nuts and Bolts**
  - Data Splits
  - Bias and Variance
  - Overfitting
  - Last Activation and Loss Function
  - Optimizers

Data Splits

Hold-Out Evaluation



Hold-Out Evaluation

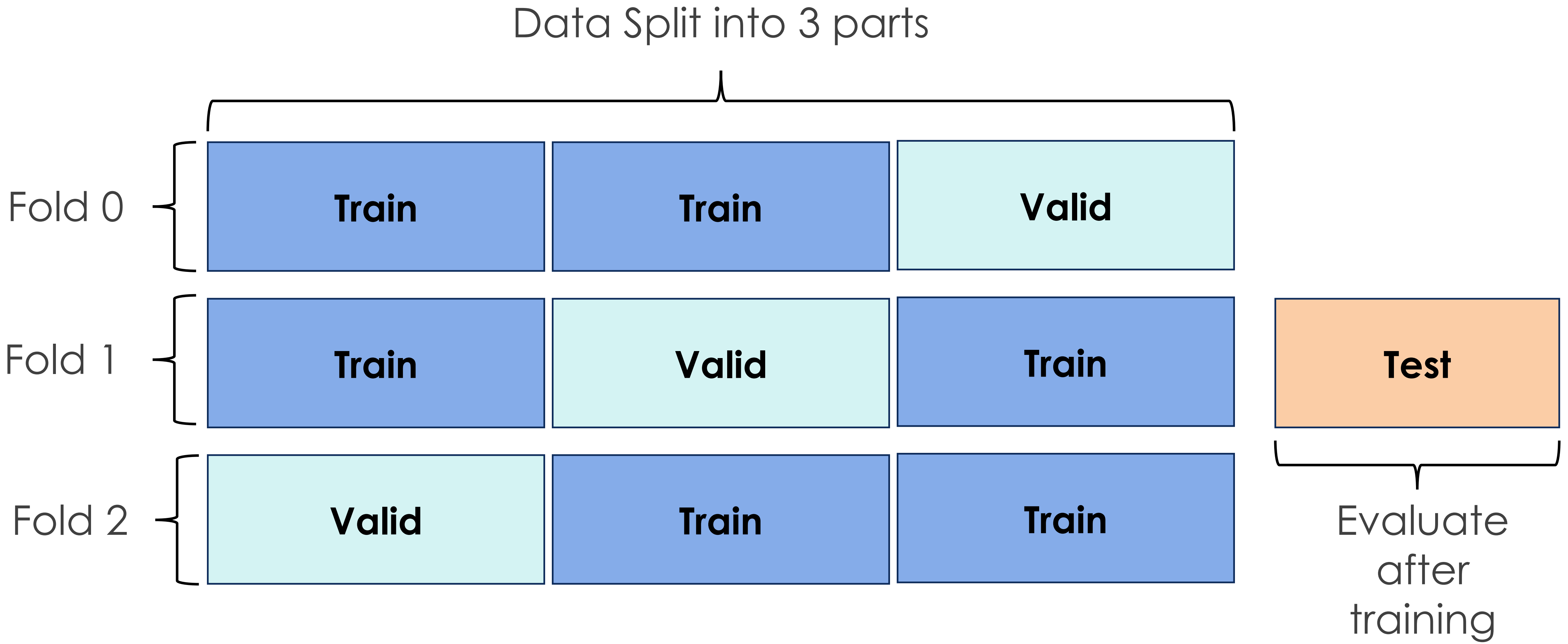
```
>>> from sklearn.model_selection import train_test_split
>>> import torch

>>> x, y = torch.rand(10, 2).numpy(), torch.rand(10).numpy()
>>> print(x.shape, y.shape)
(10, 2) (10,)

>>> split = 0.2
>>> x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=split)
>>> x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train, test_size=split)

>>> print(x_train.shape, x_valid.shape, x_test.shape)
(6, 2) (2, 2) (2, 2)
```

K-Fold Cross Validation Evaluation



K-Fold Cross Validation Evaluation

```
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.model_selection import KFold
>>> import torch

>>> x, y = torch.rand(10, 2).numpy(), torch.rand(10).numpy()
>>> print(x.shape, y.shape)
(10, 2) (10,)

# Split out the held-out test set first.
>>> split = 0.2
>>> x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=split)

>>> kf = KFold(n_splits=3)
>>> three_folds = {i:{'x_train': x_train[train_idx], 'x_valid': x_train[valid_idx],
...                   'y_train': x_train[train_idx], 'y_valid': x_train[valid_idx]}
...               for i, (train_idx, valid_idx) in enumerate(kf.split(x_train))}
```

K-Fold Cross Validation Evaluation

```
>>> three_folds[0]['x_train'].shape # Fold 0 train set.
(5, 2)
>>> three_folds[1]['x_train'].shape # Fold 1 train set.
(5, 2)
>>> three_folds[2]['x_train'].shape # Fold 2 train set.
(6, 2)

>>> three_folds[0]['x_valid'].shape # Fold 0 valid set.
(3, 2)
>>> three_folds[1]['x_valid'].shape # Fold 1 valid set.
(3, 2)
>>> three_folds[2]['x_valid'].shape # Fold 2 valid set.
(2, 2)
```

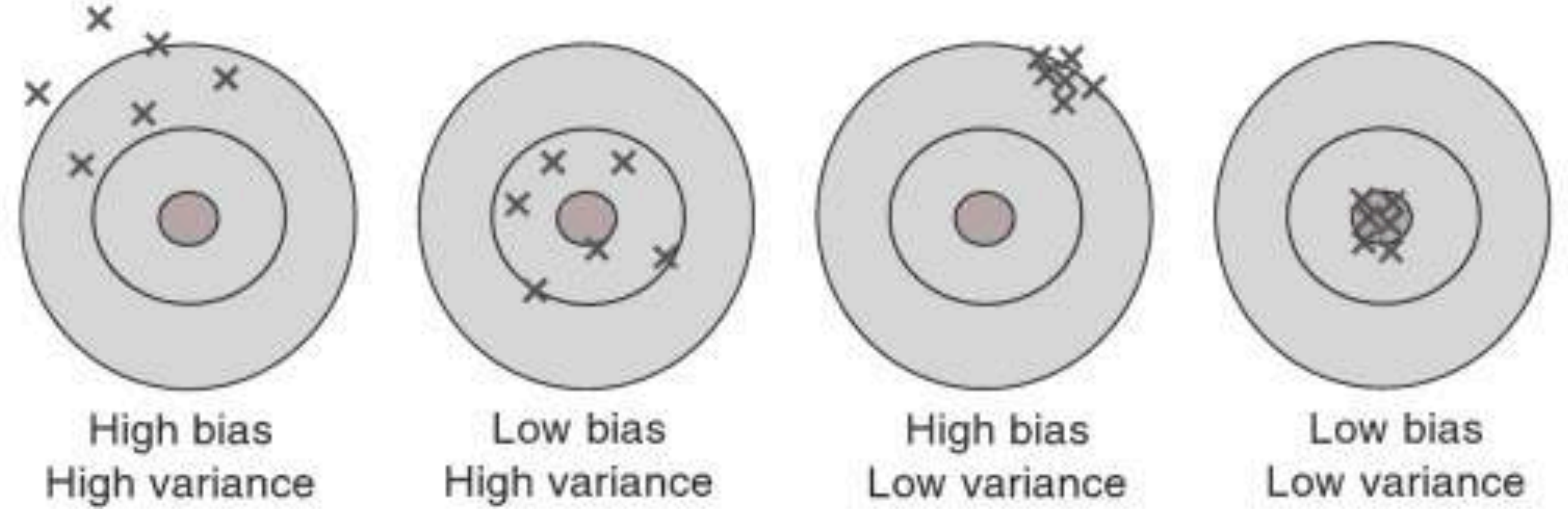
Bias-Variance Tadeoff

- *"A **small network**, with say one hidden unit **is likely to be biased**, since the repertoire of available functions spanned by $f(x, w)$ over allowable weights will in this case be quite limited."*

- *"if we **overparameterize**, via a large number of hidden units and associated weights, then bias will be reduced (… **with enough weights and hidden units, the network will interpolate the data**) but there is then the **danger of significant variance** contribution to the mean-square error"*

(German et al. 1992)

Bias-Variance Tadeoff



(Moore and McCabe, 2009)

Optimization and Generalization

- **Optimization:** *process of adjusting a model to get the best performance possible on the training data*

- **Generalization:** *how well trained model performs on data it has never seen before*

Over-/Underfitting

- **Underfit:** when model is *not optimized*

- **Overfit:** when model *fails to generalize*

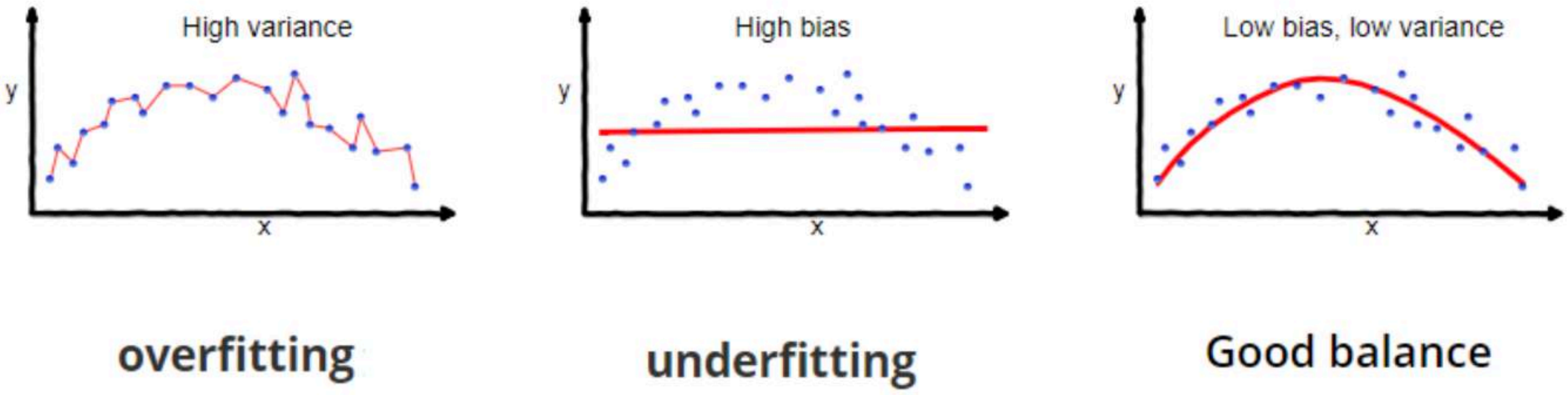


Image from https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

Optimization and Generalization

- **Optimization:** *process of adjusting a model to get the best performance possible on the training data*

- **Generalization:** *how well trained model performs on data it has never seen before*

- **What to do when model is underfitting (not optimal)?**
  - Train longer

- **How to prevent overfitting (i.e. generalize)?**
  - Get more data
  - Modulate quantity of information fed to model

(Chollet, 2017)

Overcoming Overfitting

- **Reducing size of the model** (i.e. no. of layers and no. of units per layer) prevents overfitting

- **Weights regularization** (i.e. adding a cost associated with having large weights) put constraints on complexity of the model by forcing its weights to take small values

- **Dropping out** (i.e. randomly setting activated outputs to zero) is effective in regularizing the model

Last Layer Activation & Lost Functions

MSELoss

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input *x* and target *y*.

The loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If reduce is `True`, then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if size\_average} = \text{True}, \\ \operatorname{sum}(L), & \text{if size\_average} = \text{False}. \end{cases}$$

The sum operation still operates over all the elements, and divides by *n*.

The division by *n* can be avoided if one sets `size_average` to `False`.

To get a batch of losses, a loss per batch element, set *reduce* to `False`. These losses are not averaged and are not affected by *size_average*.

(PyTorch Documentation: https://pytorch.org/docs/stable/nn.html)

NLLLoss

The negative log likelihood loss. It is useful to train a classification problem with *C* classes.

If provided, the optional argument *weight* should be a 1D Tensor assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The input given through a forward call is expected to contain log-probabilities of each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 2$ for the *K*-dimensional case (described later).

Obtaining log-probabilities in a neural network is easily achieved by adding a *LogSoftmax* layer in the last layer of your network. You may use *CrossEntropyLoss* instead, if you prefer not to add an extra layer.

The target that this loss expects is a class index (*0 to C-1, where C = number of classes*)

(PyTorch Documentation: https://pytorch.org/docs/stable/nn.html)

NLLLoss

If `reduce` is `False`, the loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n, y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

where N is the batch size. If `reduce` is `True` (default), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^N \frac{1}{\sum_{n=1}^N w_{y_n}} l_n, & \text{if size\_average} = \text{True},\\ \sum_{n=1}^N l_n, & \text{if size\_average} = \text{False}. \end{cases}$$

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 2$, where K is the number of dimensions, and a target of appropriate shape (see below). In the case of images, it computes NLL loss per-pixel.

(PyTorch Documentation: https://pytorch.org/docs/stable/nn.html)

CrossEntropyLoss

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 2$ for the *K*-dimensional case (described later).

This criterion expects a class index (0 to *C-1*) as the *target* for each value of a 1D tensor of size *minibatch*

(PyTorch Documentation: https://pytorch.org/docs/stable/nn.html)

CrossEntropyLoss

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

or in the case of the *weight* argument being specified:

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

The losses are averaged across observations for each minibatch.

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 2$, where K is the number of dimensions, and a target of appropriate shape (see below).

(PyTorch Documentation: https://pytorch.org/docs/stable/nn.html)

BCELoss

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

The loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where N is the batch size. If reduce is `True`, then

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if size\_average} = \text{True}, \\ \operatorname{sum}(L), & \text{if size\_average} = \text{False}. \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets *y* should be numbers between 0 and 1.

(PyTorch Documentation: https://pytorch.org/docs/stable/nn.html)

Last Layer Activation and Loss Function

For labels $y_i \in \{0,1\}$the likelihood of some binary data under the Bernoulli model with parameters θ is

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p(y_i = 1|\theta)^{y_i} p(y_i = 0|\theta)^{1-y_i}$$

NLLoss = $\ln \mathcal{L}(\theta) = -\sum_{i=1}^{n} y_i \ln(p(y = 1|\theta) + (1 - y_i)\ln(y = 0|\theta)$

CE = $\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln(p(y = 1|\theta) + (1 - y_i)\ln(y = 0|\theta)$

Last Layer Activation and Loss Function

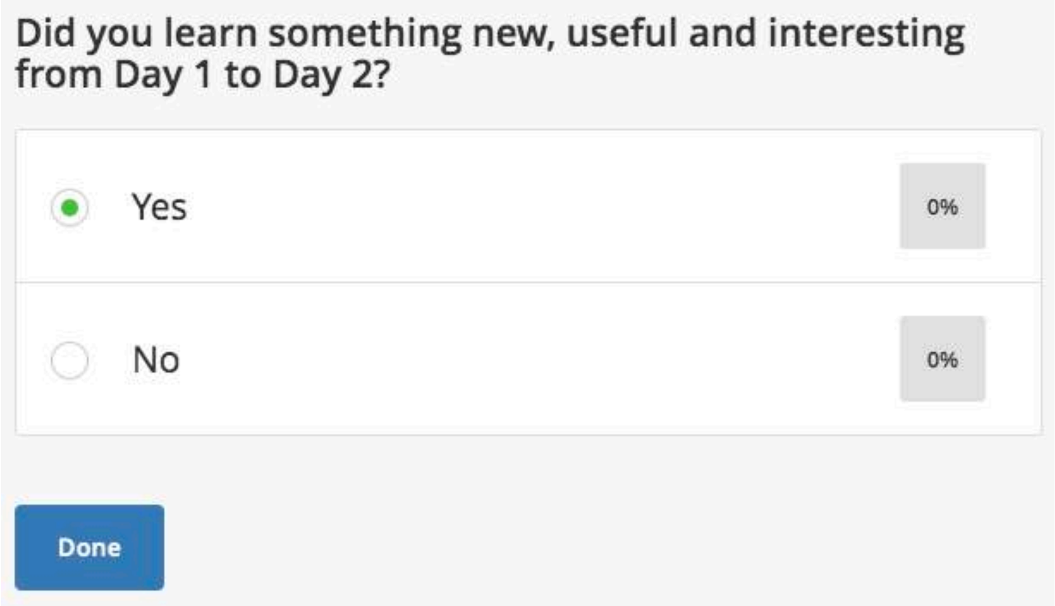
For labels $y_i \in \{0,1\}$the likelihood of some binary data under the Bernoulli model with parameters θ is

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p(y_i = 1|\theta)^{y_i} p(y_i = 0|\theta)^{1-y_i}$$

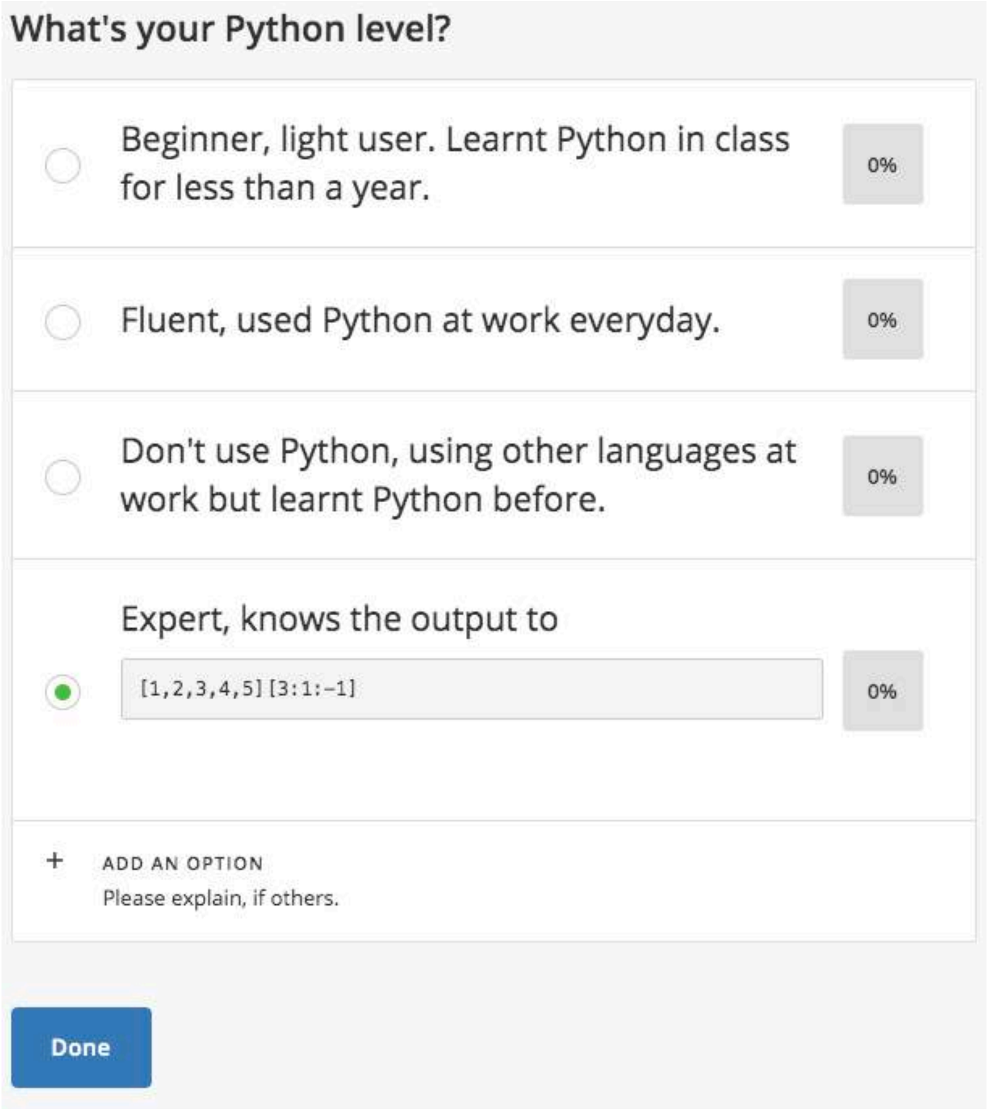
NLLoss = $\ln \mathcal{L}(\theta) = -\sum_{i=1}^{n} y_i \ln(\hat{y})$

CE = $\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln(\hat{y})$

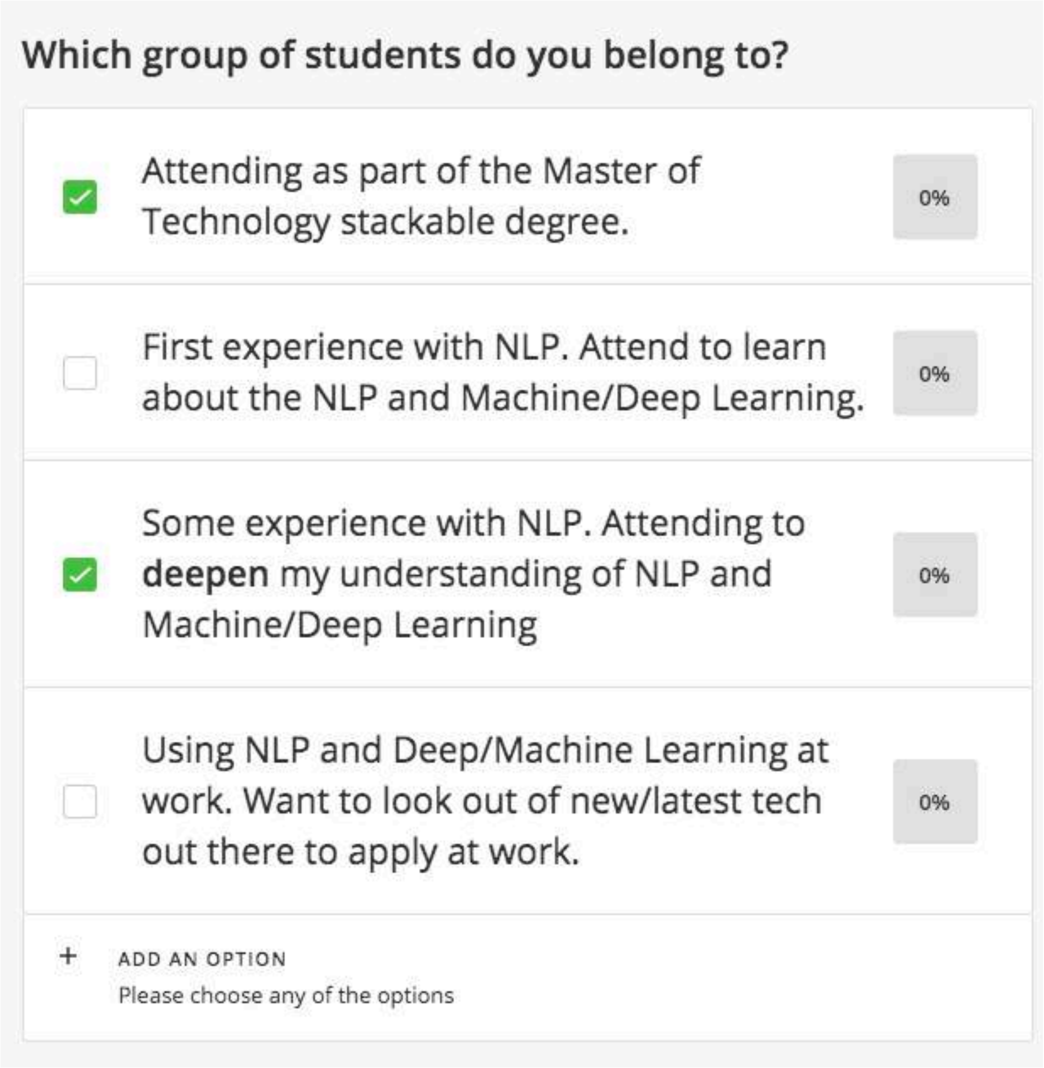
Binary Classification



Multi-Class Classification



Multi-Class Multi-Label Classification



Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification			
Multi-class, single-label classification			
Multi-class, multi-label classification			
Regression to arbitrary value			
Regression (0, 1)			

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification			
Multi-class, multi-label classification			
Regression to arbitrary value			
Regression (0, 1)			

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification	Sigmoid / Softmax		
Multi-class, multi-label classification			
Regression to arbitrary value			
Regression (0, 1)			

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification	Sigmoid / Softmax	Categorical Cross Entropy	torch.nn.CrossEntropyLoss
	LogSoftmax		
Multi-class, multi-label classification			
Regression to arbitrary value			
Regression (0, 1)			

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification	Sigmoid / Softmax	Categorical Cross Entropy	torch.nn.CrossEntropyLoss
	LogSoftmax	Negative Log Loss	torch.nn.NLLLoss
Multi-class, multi-label classification			
Regression to arbitrary value			
Regression (0, 1)			

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification	Sigmoid / Softmax	Categorical Cross Entropy	torch.nn.CrossEntropyLoss
	LogSoftmax	Negative Log Loss	torch.nn.NLLLoss
Multi-class, multi-label classification	Sigmoid / Softmax	Binary Cross Entropy	torch.nn.BCELoss
Regression to arbitrary value			
Regression (0, 1)			

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification	Sigmoid / Softmax	Categorical Cross Entropy	torch.nn.CrossEntropyLoss
	LogSoftmax	Negative Log Loss	torch.nn.NLLLoss
Multi-class, multi-label classification	Sigmoid / Softmax	Binary Cross Entropy	torch.nn.BCELoss
Regression to arbitrary value	None	L2 Loss	torch.nn.MSELoss
Regression (0, 1)	Sigmoid	L2 Loss	torch.nn.MSELoss

Last Layer Activation and Loss Function

Problem	Last Layer Activation	Loss Function	PyTorch
Binary Classification	Sigmoid	Binary Cross Entropy	torch.nn.BCELoss
Multi-class, single-label classification	Sigmoid / Softmax	Categorical Cross Entropy	torch.nn.CrossEntropyLoss
	LogSoftmax	Negative Log Loss	torch.nn.NLLLoss
Multi-class, multi-label classification	Sigmoid / Softmax	Binary Cross Entropy	torch.nn.BCELoss
Regression to arbitrary value	None	L2 Loss	torch.nn.MSELoss
Regression (0, 1)	Sigmoid	L2 Loss	torch.nn.MSELoss

Optimizers

Gradient Descents

Stochastic Gradient Descent (SGD)
torch.optim.SGD

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

Mini-batch SGD
torch.optim.SGD

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

SGD with momentum
torch.optim.SGD

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

Gradient Descents

```
def step(self, closure=None):
    """Performs a single optimization step.

    Arguments:
        closure (callable, optional): A closure that reevaluates the model
            and returns the loss.
    """
    loss = None
    if closure is not None:
        loss = closure()

    for group in self.param_groups:
        weight_decay = group['weight_decay']
        momentum = group['momentum']
        dampening = group['dampening']
        nesterov = group['nesterov']
```

Gradient Descents

```python
    for p in group['params']:
        if p.grad is None:
            continue
        d_p = p.grad.data
        if weight_decay != 0:
            d_p.add_(weight_decay, p.data)
        if momentum != 0:
            param_state = self.state[p]
            if 'momentum_buffer' not in param_state:
                buf = param_state['momentum_buffer'] = torch.zeros_like(p.data)
                buf.mul_(momentum).add_(d_p)
            else:
                buf = param_state['momentum_buffer']
                buf.mul_(momentum).add_(1 - dampening, d_p)
            if nesterov:
                d_p = d_p.add(momentum, buf)
            else:
                d_p = buf

        p.data.add_(-group['lr'], d_p)

return loss
```

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

Adaptive Moment Estimation (Kingma, 2015)

- "*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*" – (Dozat, 2016)

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

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

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

SGD vs Adam

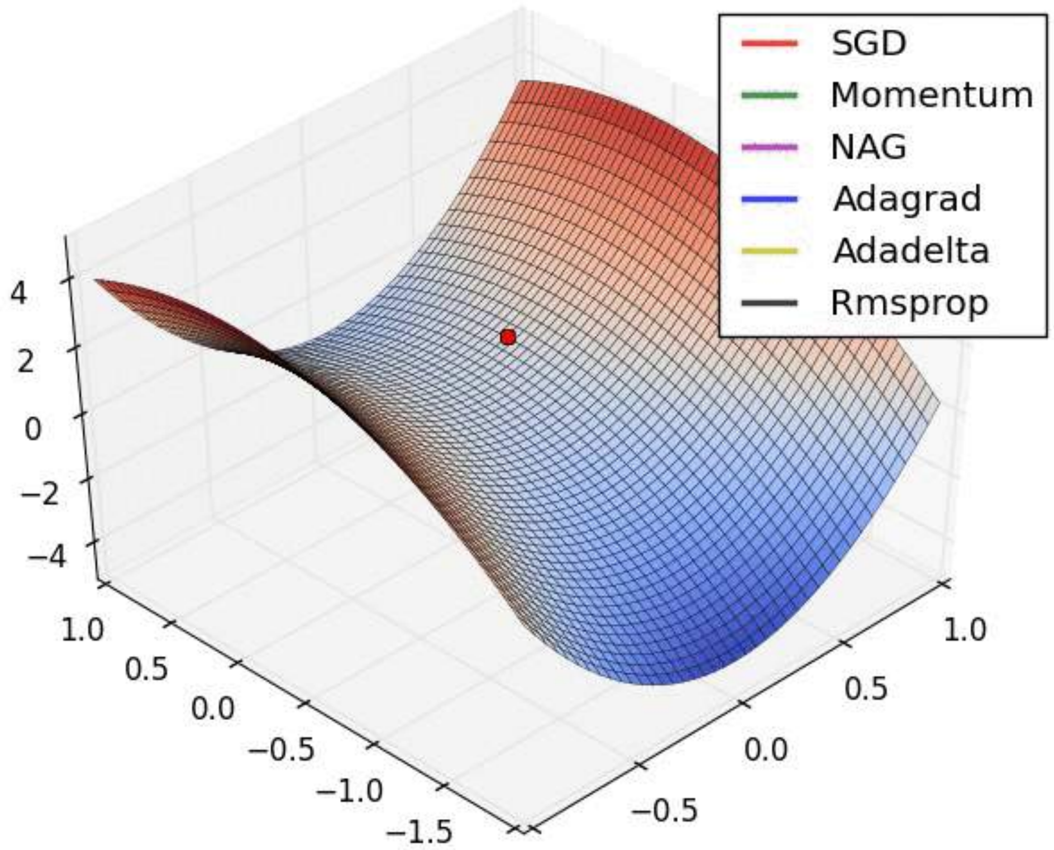
```
# Vanilla SGD
x += - learning_rate * dx
```

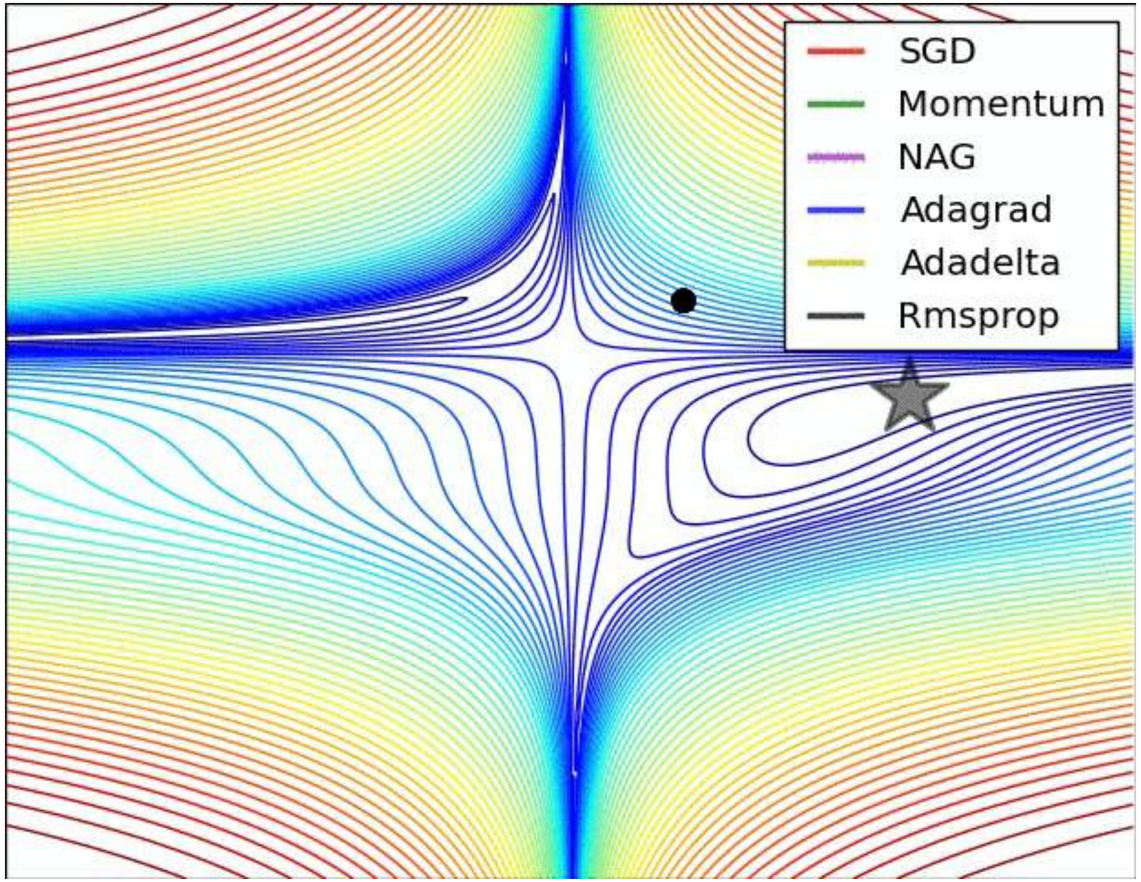
x is a vector of parameters and
dx is the gradient

```
# Adam
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m /
        (np.sqrt(v) + eps)
```

x is a vector of parameters and
dx is the gradient
m is the smoothen gradient
v is the 'cache' used to normalize **x**
eps is smoothing term (1e-4 to 1e-8)
beta1,beta2 are hypers (0.9, 0.999)

Optimizers (Effects)





Note:
Deniz Yuret has a nice blogpost on optimizers http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html
Sebestian Ruder too on optimizers http://ruder.io/optimizing-gradient-descent/
Stanford has a nice overview of optimizers on http://cs231n.github.io/neural-networks-3/

Google says…

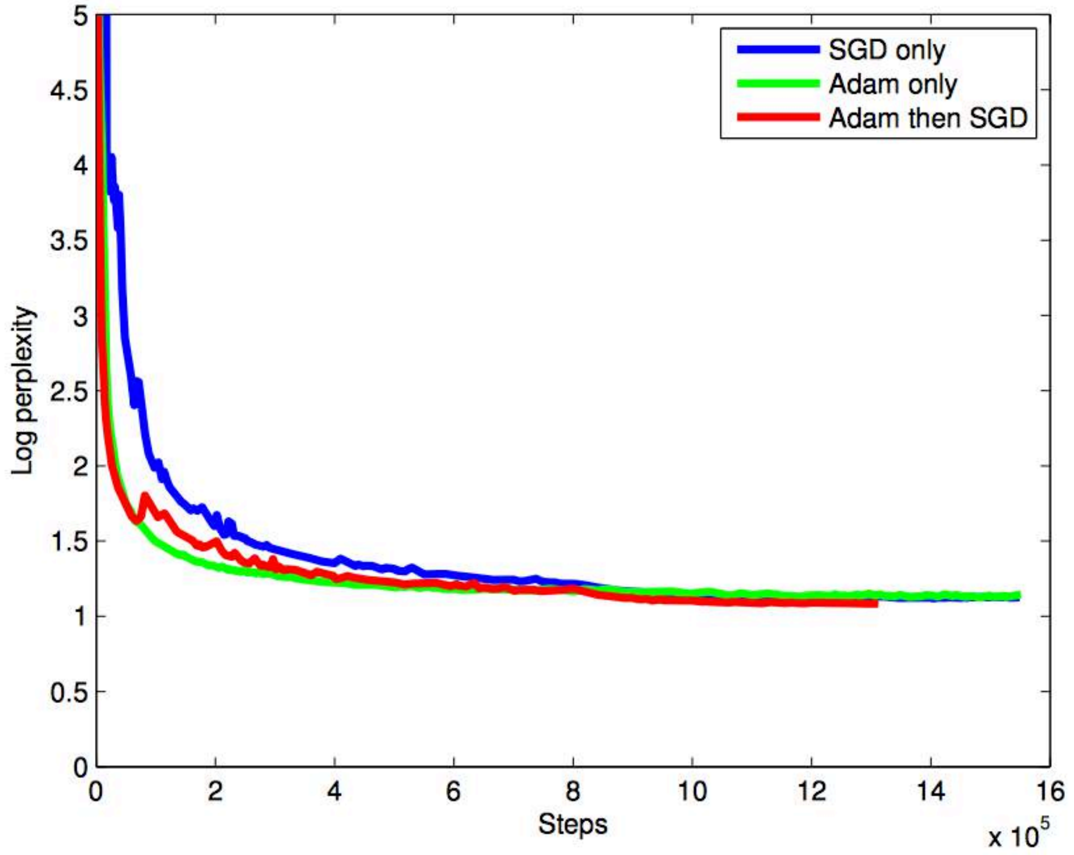
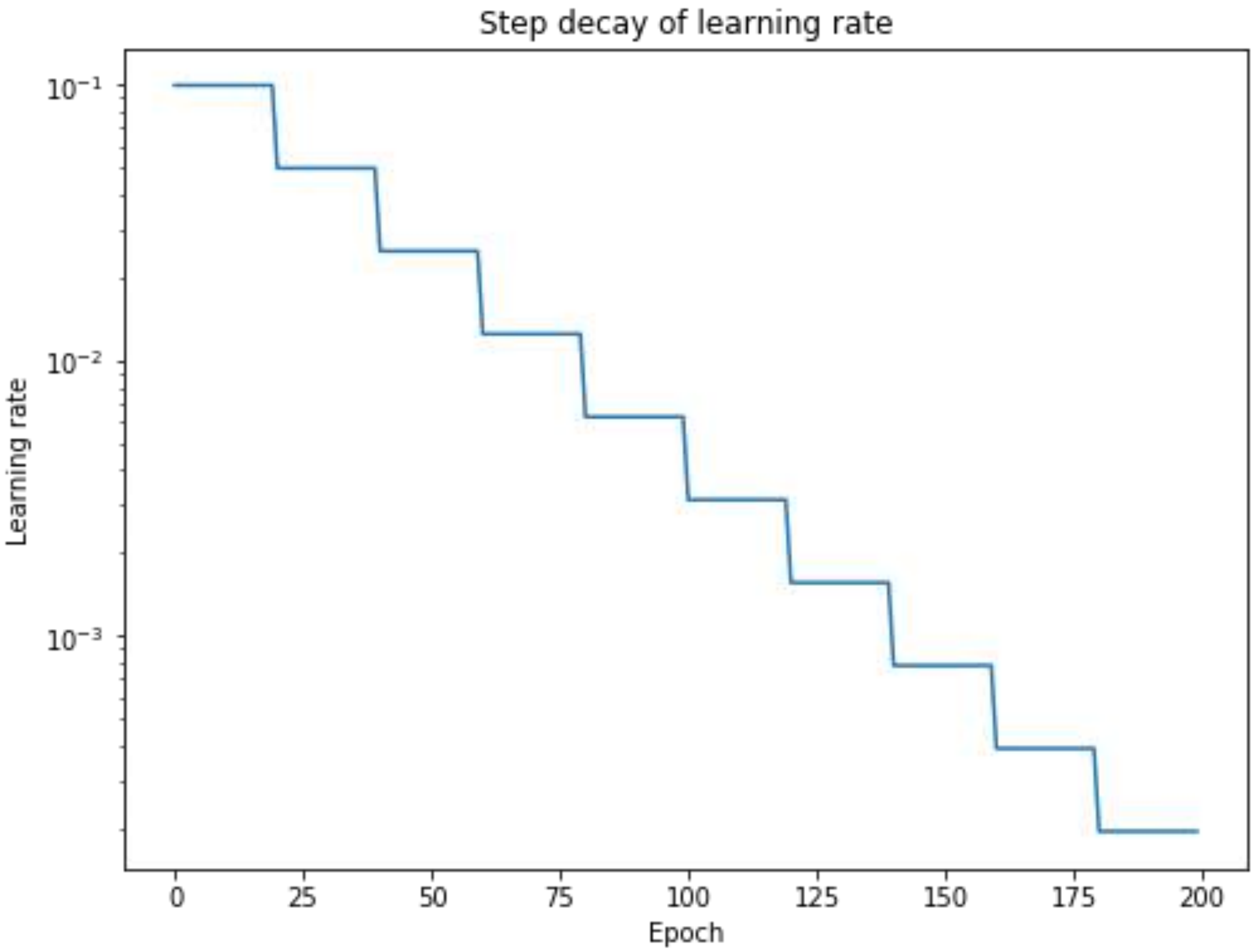


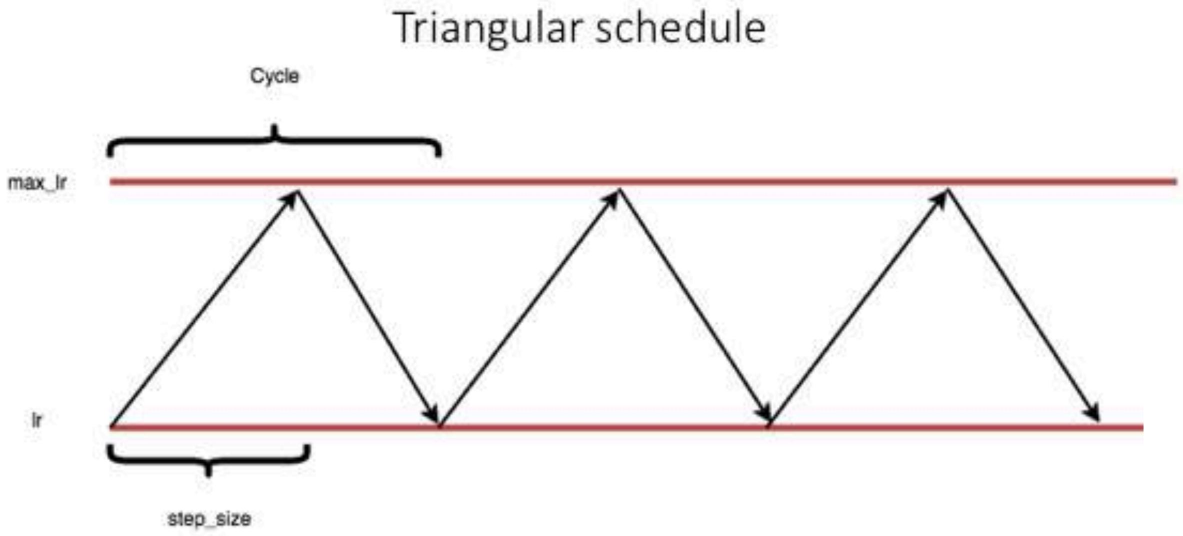
Figure 5: Log perplexity vs. steps for Adam, SGD and Adam-then-SGD on WMT En→Fr during maximum likelihood training. Adam converges much faster than SGD at the beginning. Towards the end, however, Adam-then-SGD is gradually better. Notice the bump in the red curve (Adam-then-SGD) at around 60k steps where we switch from Adam to SGD. We suspect that this bump occurs due to different optimization trajectories of Adam vs. SGD. When we switch from Adam to SGD, the model first suffers a little, but is able to quickly recover afterwards.

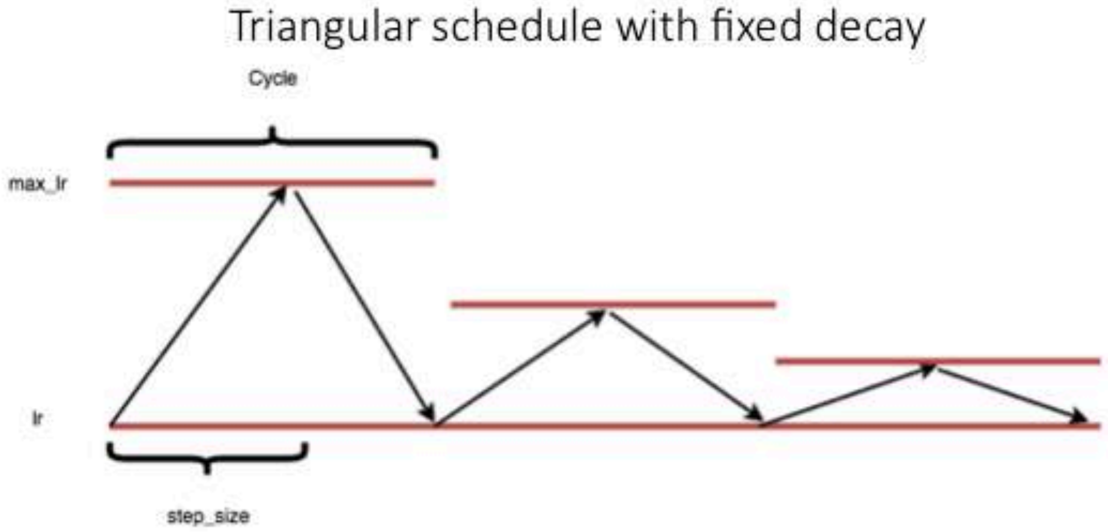
Learning Rate Annealing

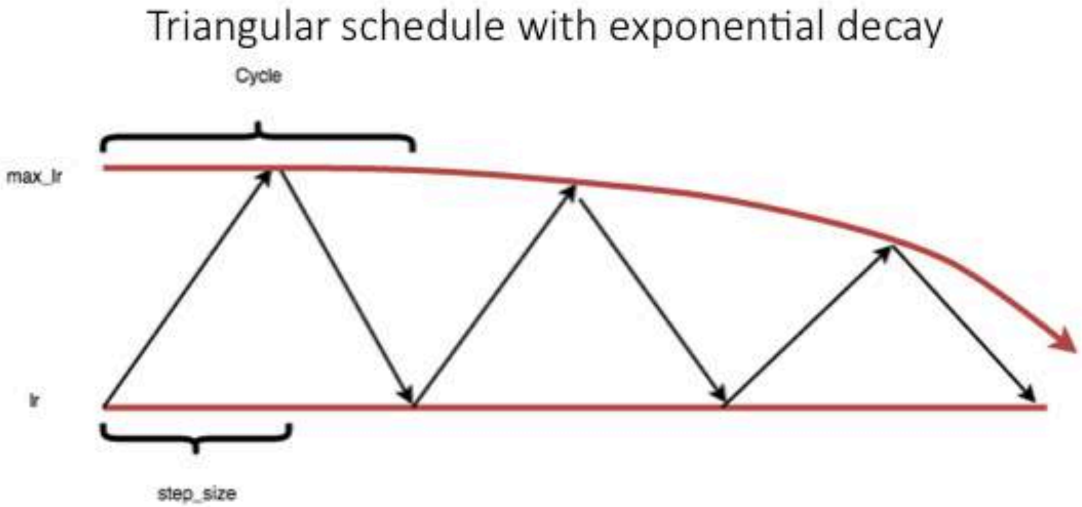


Images from https://www.jeremyjordan.me/nn-learning-rate/

Cyclic Learning Rate







(Smith, 2015)

Images from https://www.jeremyjordan.me/nn-learning-rate/

Cyclic Learning Rate

We can write the general schedule as

$$\eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min})\left(\max\left(0, 1 - x\right)\right)$$

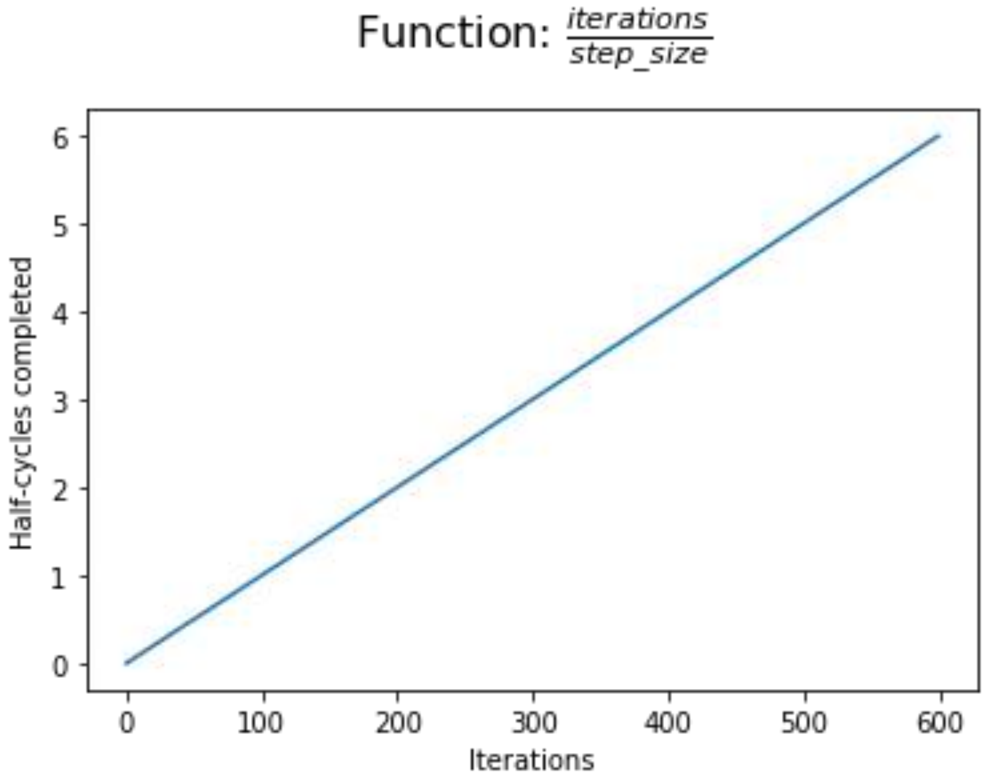
where x is defined as

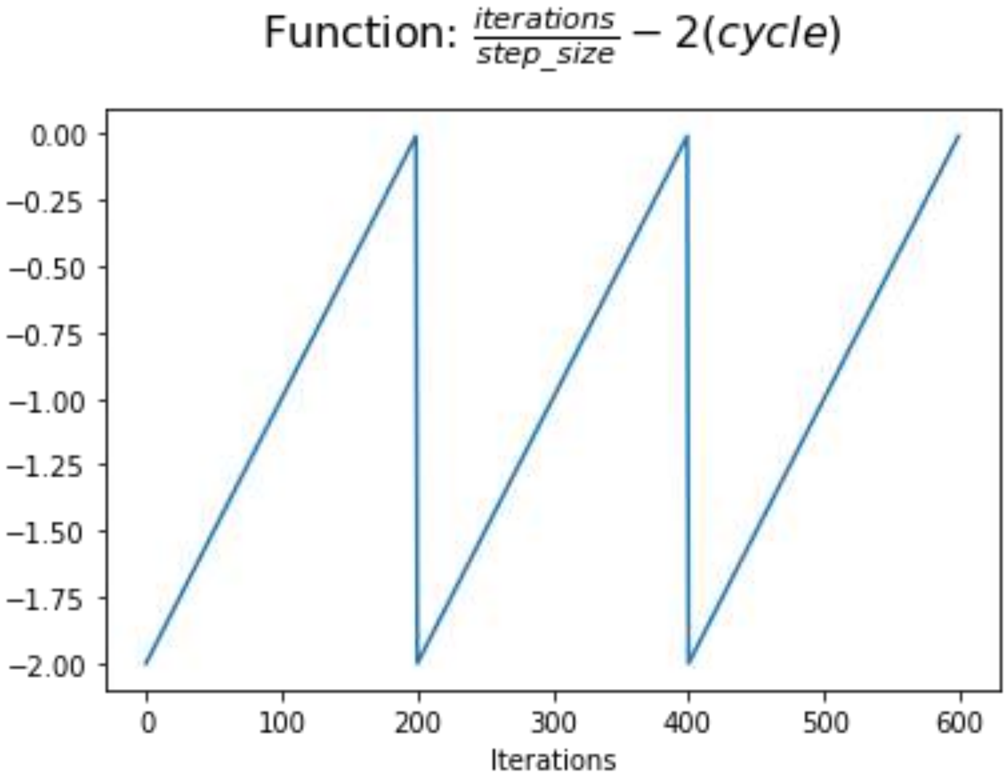
$$x = \left|\frac{iterations}{stepsize} - 2\left(cycle\right) + 1\right|$$

and $cycle$ can be calculated as

$$cycle = floor\left(\frac{1 + iterations}{2\left(stepsize\right)}\right)$$

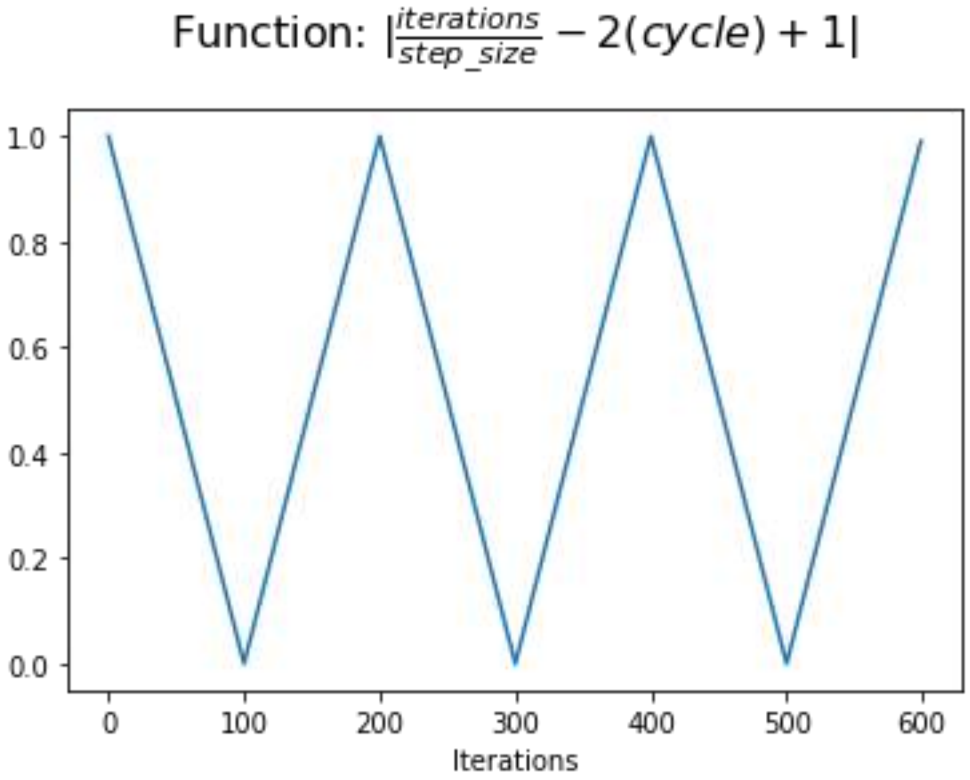
Cyclic Learning Rate

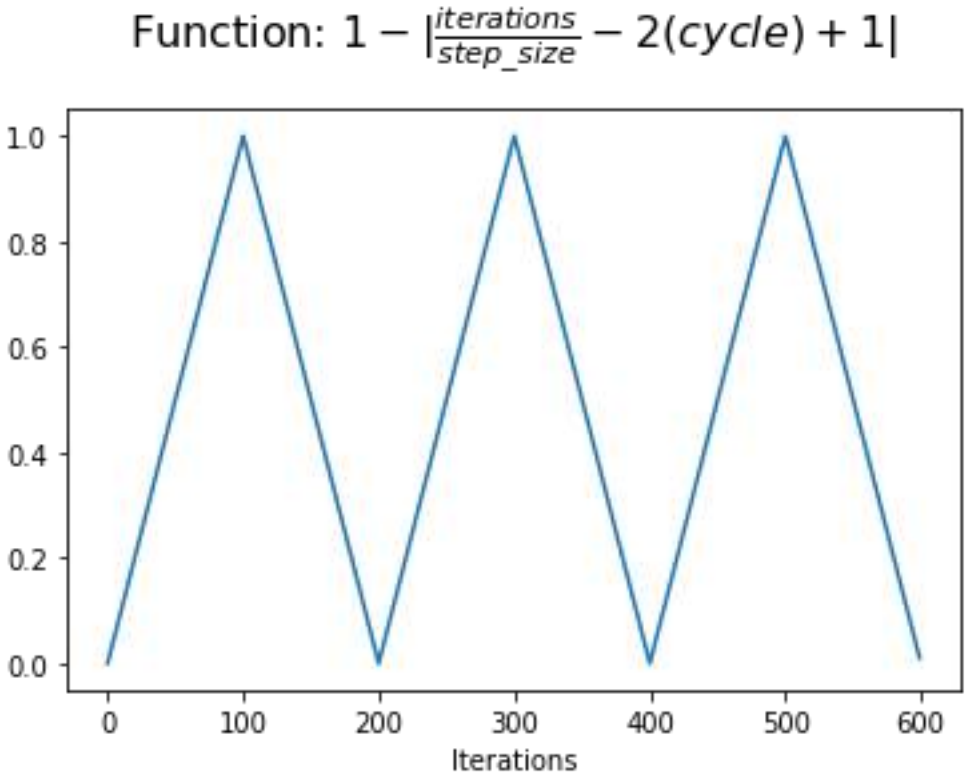




Images from https://www.jeremyjordan.me/nn-learning-rate/

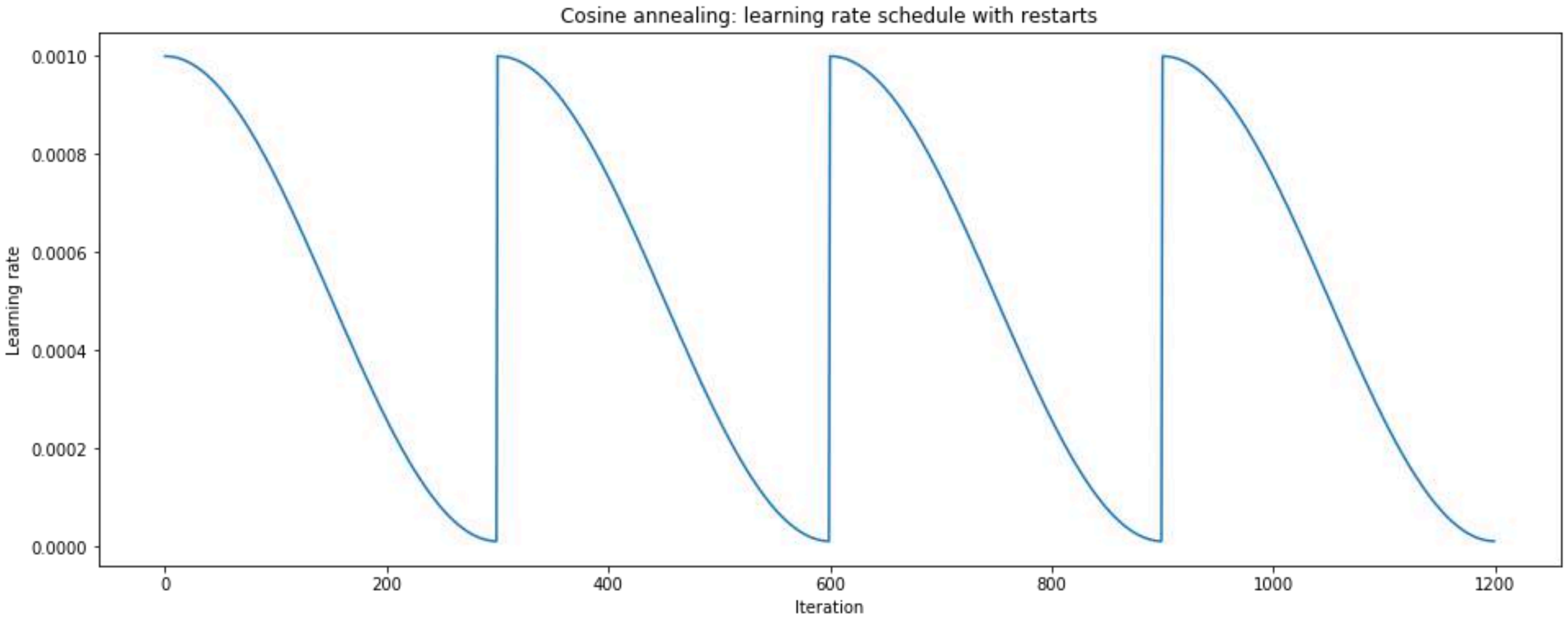
Cyclic Learning Rate





Images from https://www.jeremyjordan.me/nn-learning-rate/

Cosine SGD with Warm Restarts



$$\eta_t = \eta^i_{\min} + \frac{1}{2}\left(\eta^i_{\max} - \eta^i_{\min}\right)\left(1 + \boxed{\cos\left(\frac{T_{current}}{T_i}\pi\right)}\right)$$

Annealing, i.e. taking a partial step

Images from https://www.jeremyjordan.me/nn-learning-rate/

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from CLR_preview import CyclicLR
from adamW import AdamW

# Step 1: Initialization.
model = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.Sigmoid(),
            nn.Linear(hidden_dim, output_dim),
            nn.Sigmoid()
        )

optimizer = AdamW(model.parameters(), lr=0.001, betas=(0.9, 0.99), weight_decay = 0.1)
# For classification.
##clr_stepsize = (num_classes*50//int(batchsz))*4
clr_stepsize = 3e-4
clr_wrapper = CyclicLR(optimizer, step_size=clr_stepsize)

# Initialize the loss function.
criterion = nn.MSELoss()

losses = [] # Keeps track of the loses.
# Step 2-4 of training routine.
for _e in tqdm(range(num_epochs)):
    # Reset the gradient after every epoch.
    optimizer.zero_grad()
    # Step 2: Foward Propagation
    predictions = model(X)
    # Step 3: Back Propagation
    # Calculate the cost between the predictions and the truth.
    loss = criterion(predictions, Y)
    # Remember to back propagate the loss you've computed above.
    loss.backward()
    # Step 4: Optimizer take a step and update the weights.
    optimizer.step()
    losses.append(loss.data.item())
```

Super convergence, see https://www.fast.ai/2018/07/02/adam-weight-decay/

CLR_preview from https://github.com/ahirner/pytorch-retraining/blob/master/CLR_preview.py

AdamW from https://github.com/egg-west/AdamW-pytorch

Environment Setup

Open Anaconda Navigator.

Go to the PyTorch installation page, copy the command as per configuration: https://pytorch.org/get-started/locally/

Fire up the terminal in Anaconda Navigator.

Start a Jupyter Notebook.

https://github.com/alvations/tsundoku

Import the .ipynb to the Jupyter Notebook

Fin