Topics:

- Optimization Continued

CS 4644-DL / 7643-A
ZSOLT KIRA

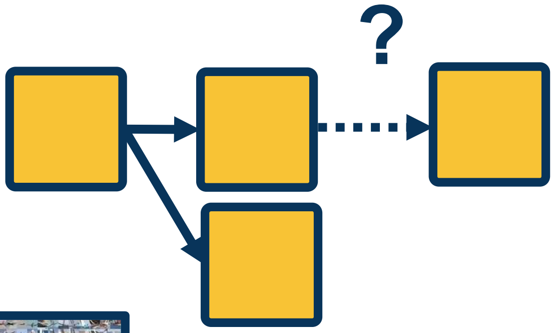
- **Assignment 1 – due tonight, grace period 02/05**

- **Assignment 2**
  - Implement convolutional neural networks

- **Facebook Lectures**: Data wrangling OH recordings available on piazza

There are still many design decisions that must be made:

- **Architecture**
- **Data Considerations**
- **Training and Optimization**
- **Machine Learning Considerations**

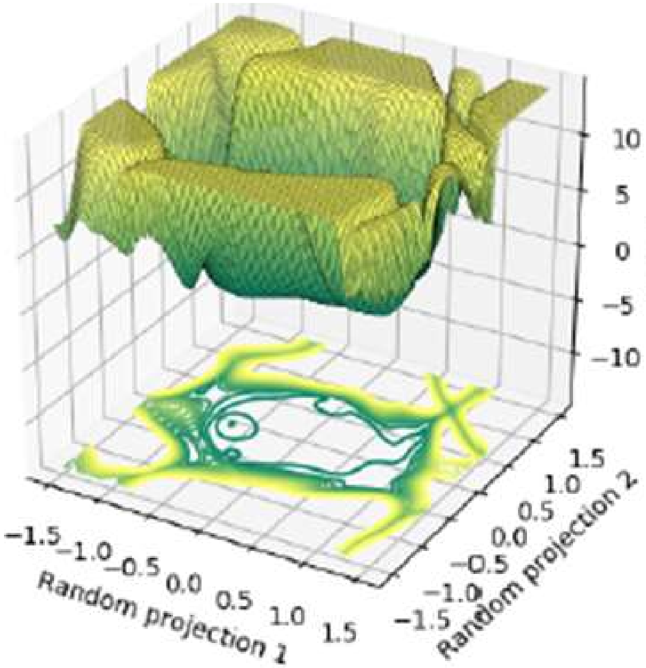


Deep learning involves **complex, compositional, non-linear functions**

The **loss landscape** is extremely **non-convex** as a result

There is **little direct theory** and a **lot of intuition/rules of thumbs** instead

- Some insight can be gained via theory for simpler cases (e.g. convex settings)

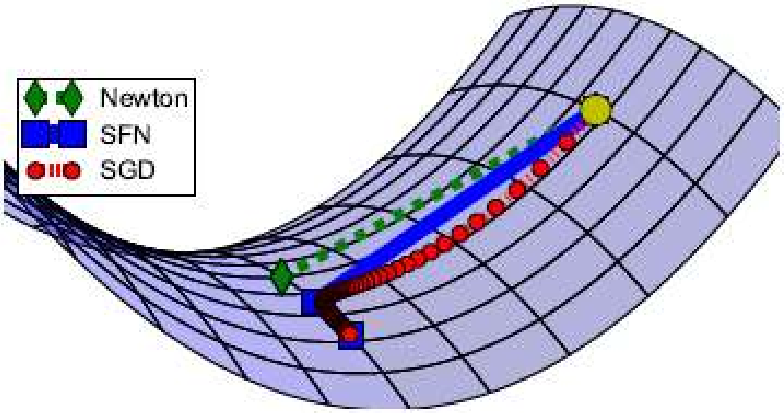


Loss Landscape

It used to be thought that **existence of local minima is the main issue** in optimization

There are other **more impactful issues**:

- Noisy gradient estimates
- Saddle points
- Ill-conditioned loss surface

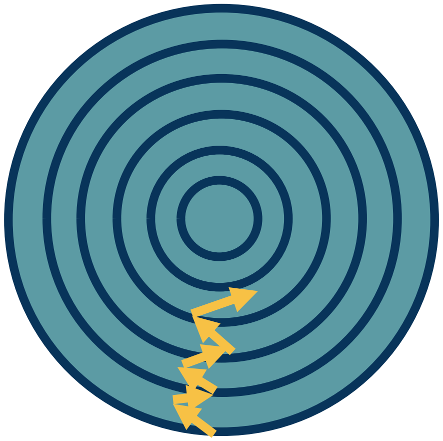


Saddle Point

From: Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, Dauphi et al., 2014.

- We use a **subset of the data at each iteration** to calculate the loss (& gradients)
- This is an **unbiased** estimator but can have high variance
- This results in **noisy steps** in gradient descent

$$L = \frac{1}{M} \sum L\left(f(x_i, W), y_i\right)$$



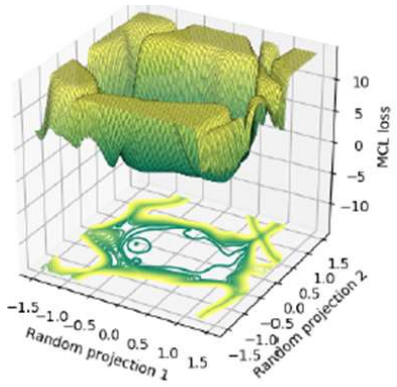
Noisy Gradients

Several **loss surface geometries** are difficult for optimization

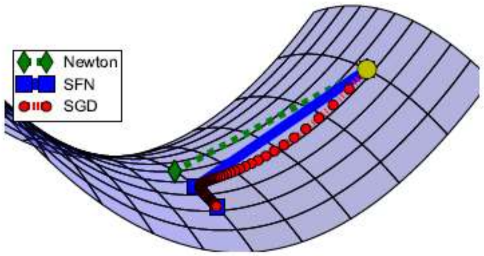
Several types of minima: Local minima, plateaus, saddle points

Saddle points are those where the gradient of orthogonal directions are zero

- But they **disagree** (it's min for one, max for another)



Plateau



Saddle Point

Loss Surface Geometry

- Gradient descent takes a step in the **steepest direction** (negative gradient)

$$w_i = w_{i-1} - \alpha \frac{\partial L}{\partial w_i}$$

- **Intuitive idea:** Imagine a ball rolling down loss surface, and use **momentum** to pass flat surfaces

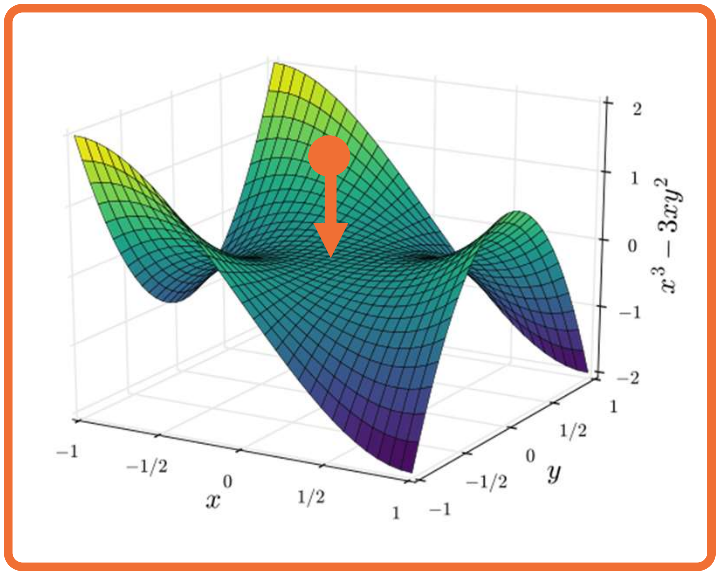
$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial w_{i-1}}$$

Update Velocity (starts as 0, $\beta = 0.99$)

$$w_i = w_{i-1} - \alpha v_i$$

Update Weights

- Generalizes SGD ($\beta = 0$)



Adding Momentum

- Velocity term is an **exponential moving average** of the gradient

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial w_{i-1}}$$

$$v_i = \beta(\beta\, v_{i-2} + \frac{\partial L}{\partial w_{i-2}}) + \frac{\partial L}{\partial w_{i-1}}$$

$$= \beta^2 v_{i-2} + \beta \frac{\partial L}{\partial w_{i-2}} + \frac{\partial L}{\partial w_{i-1}}$$

- There is a **general class of accelerated gradient methods**, with some theoretical analysis (under assumptions)

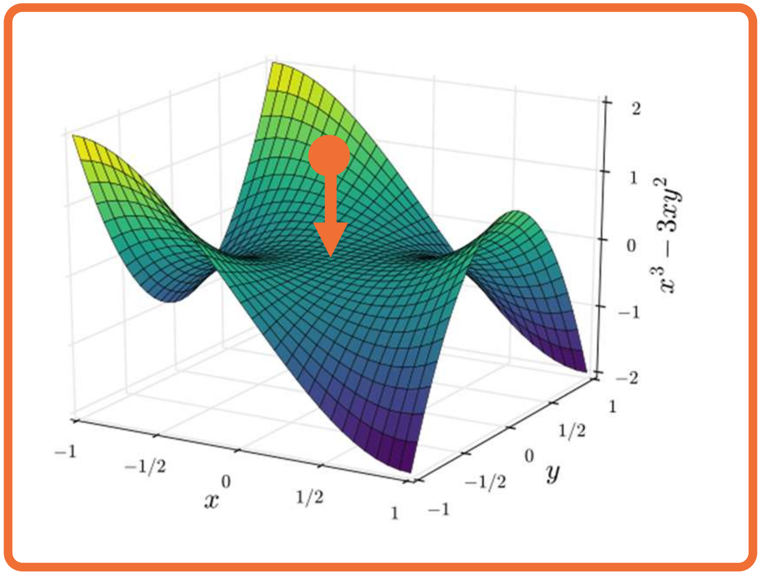
Equivalent formulation:

$$v_i = \beta v_{i-1} - \alpha \frac{\partial L}{\partial w_{i-1}}$$

Update Velocity (starts as 0)

$$w_i = w_{i-1} + v_i$$

Update Weights



Equivalent Momentum Update

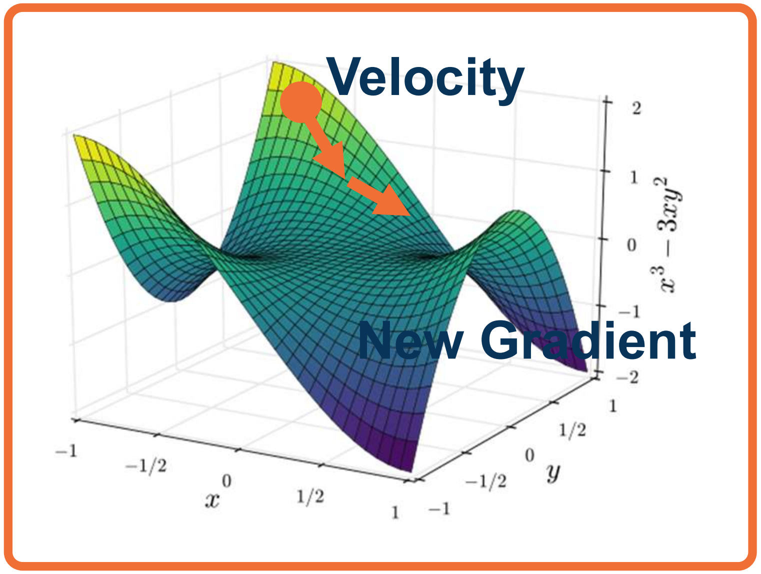
Key idea: Rather than combining velocity with current gradient, go along velocity **first** and then calculate gradient at new point

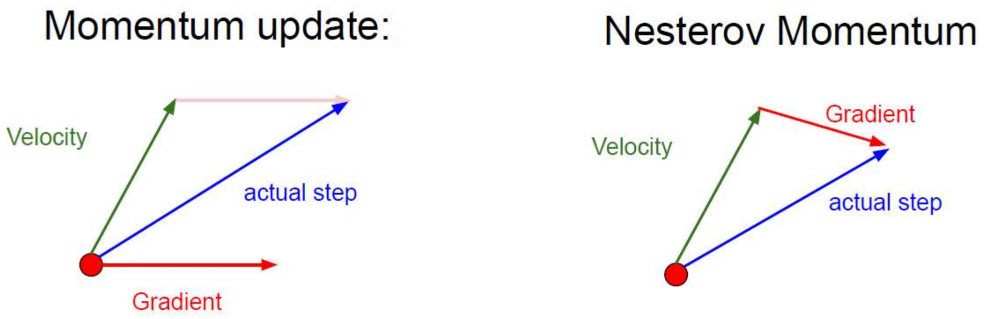
- We know velocity is probably a **reasonable direction**

$$\hat{w}_{i-1} = w_{i-1} + \beta v_{i-1}$$

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial \hat{w}_{i-1}}$$

$$w_i = w_{i-1} - \alpha\, v_i$$




Momentum update:

Nesterov Momentum

Figure Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Nesterov Momentum

Momentum

Note there are **several equivalent formulations** across deep learning frameworks!

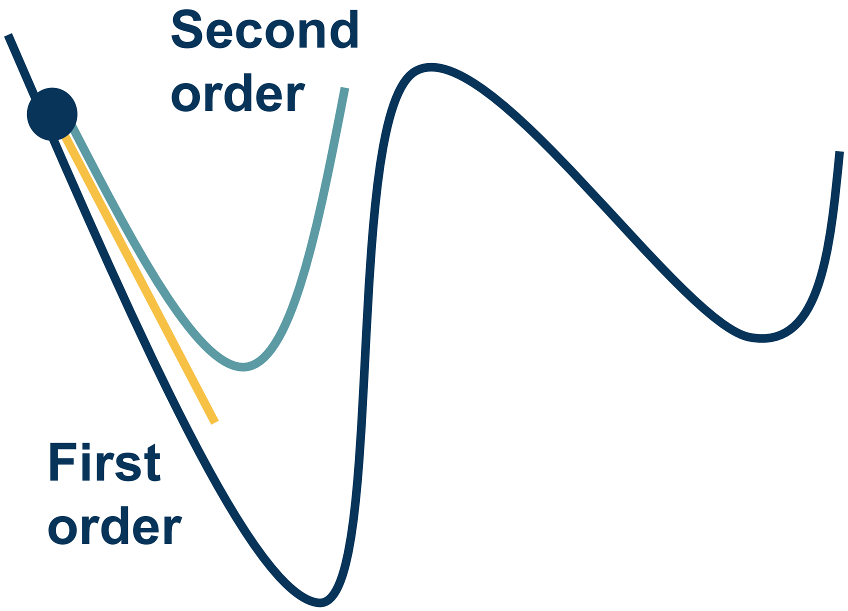
Resource: https://medium.com/the-artificial-impostor/sgd-implementation-in-pytorch-4115bcb9f02c

- Various mathematical ways to **characterize the loss landscape**

- If you liked **Jacobians**… meet the

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Gives us information about the **curvature of the loss surface**



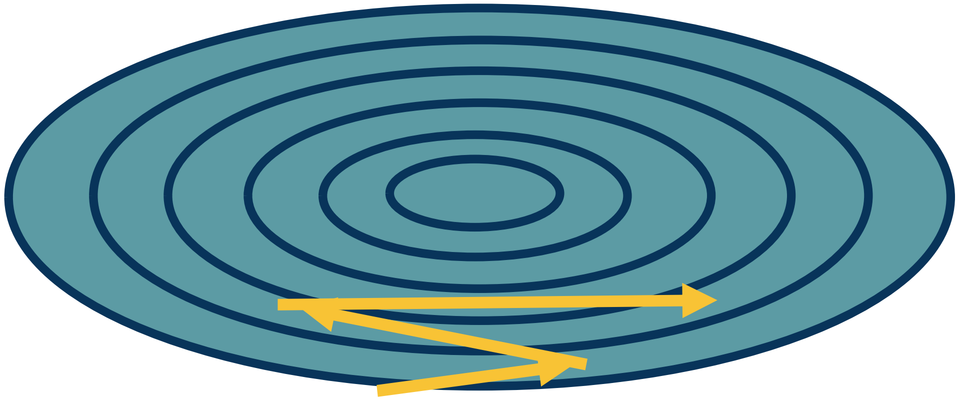
Hessian and Loss Curvature

Condition number is the ratio of the largest and smallest eigenvalue

- Tells us how different the curvature is along different dimensions

If this is high, SGD will make **big** steps in some dimensions and **small** steps in other dimension

Second-order optimization methods divide steps by curvature, but expensive to compute



Condition Number

Per-Parameter Learning Rate

Idea: Have a dynamic learning rate for each weight

Several flavors of **optimization algorithms**:

- RMSProp
- Adagrad
- Adam
- …

SGD can achieve similar results in many cases but with much more tuning

Adagrad

Idea: Use gradient statistics to reduce learning rate across iterations

Denominator: Sum up gradients over iterations

Directions with **high curvature will have higher gradients**, and learning rate will reduce

$$G_i = G_{i-1} + \left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$w_i = w_{i-1} - \frac{\alpha}{\sqrt{G_i + \epsilon}} \frac{\partial L}{\partial w_{i-1}}$$

As gradients are accumulated learning rate will go to zero

Duchi, et al., "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization"

Solution: Keep a moving average of squared gradients!

Does not saturate the learning rate

$$G_i = \beta G_{i-1} + (1-\beta)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$w_i = w_{i-1} - \frac{\alpha}{\sqrt{G_i + \epsilon}} \frac{\partial L}{\partial w_{i-1}}$$

RMSProp

Combines ideas from above algorithms

Maintains both first and second moment statistics for gradients

$$v_i = \beta_1\, v_{i-1} + (1-\beta_1)\left(\frac{\partial L}{\partial w_{i-1}}\right)$$

$$G_i = \beta_2\, G_{i-1} + (1-\beta_2)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$w_i = w_{i-1} - \frac{\alpha\, v_i}{\sqrt{G_i + \epsilon}}$$

But unstable in the beginning (one or both of moments will be tiny values)

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam

Adam

Solution: Time-varying bias correction

Typically $\boldsymbol{\beta_1 = 0.9,\ \ \beta_2 = 0.999}$

So $\hat{v}_i$ will be small number divided by (1-0.9=0.1) resulting in more reasonable values (and $\hat{G}_i$ larger)

$$v_i = \beta_1\, v_{i-1} + (1 - \beta_1)\left(\frac{\partial L}{\partial w_{i-1}}\right)$$

$$G_i = \beta_2\, G_{i-1} + (1 - \beta_2)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$\hat{v}_i = \frac{v_i}{1 - \beta_1^t} \qquad \hat{G}_i = \frac{G_i}{1 - \beta_2^t}$$

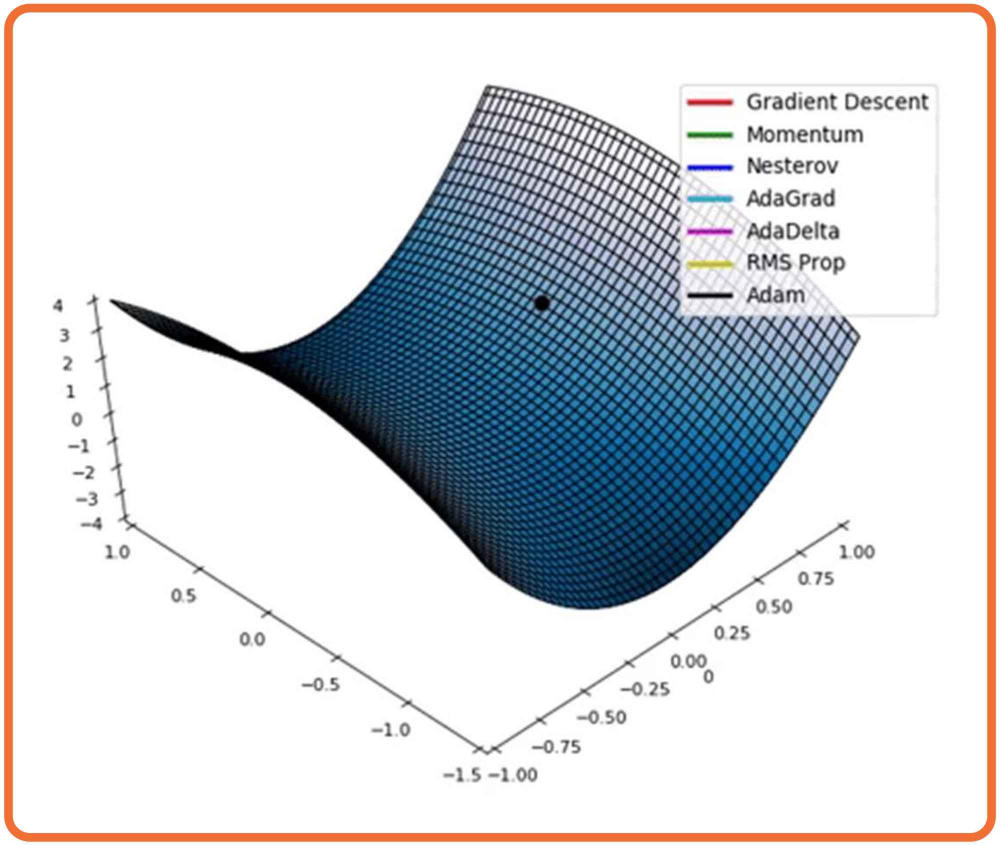
$$w_i = w_{i-1} - \frac{\alpha\, \hat{v}_i}{\sqrt{\hat{G}_i + \epsilon}}$$

Optimizers behave differently **depending on landscape**

Different behaviors such as **overshooting, stagnating, etc.**

Plain SGD+Momentum can generalize better than adaptive methods, but requires more tuning

- **See:** *Luo et al., Adaptive Gradient Methods with Dynamic Bound of Learning Rate, ICLR 2019*



From: https://mlfromscratch.com/optimizers-explained/#/

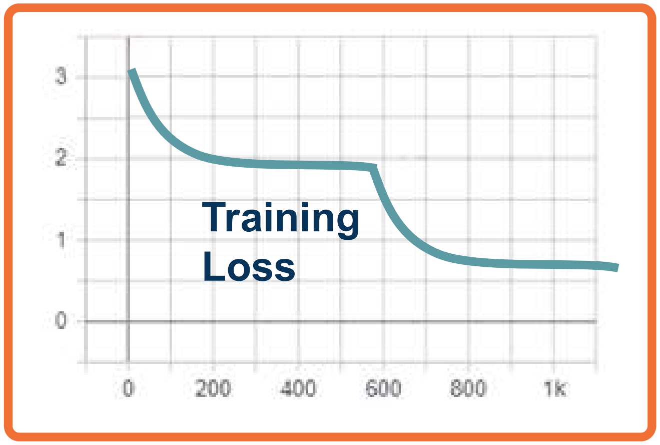
Behavior of Optimizers

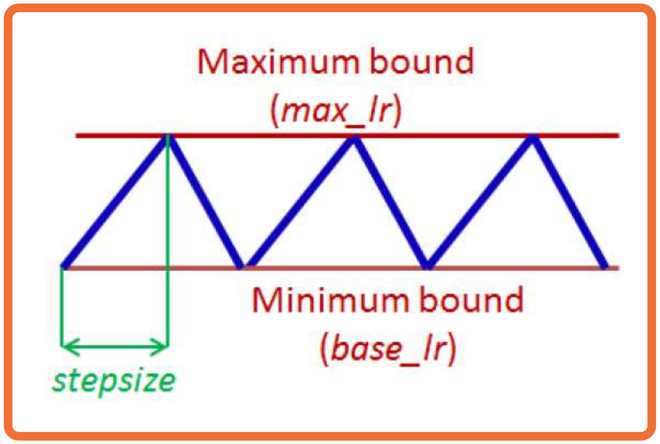
First order optimization methods have **learning rates**

Theoretical results rely on **annealed learning rate**

Several schedules that are typical:

- Graduate student!
- Step scheduler
- Exponential scheduler
- Cosine scheduler





From: Leslie Smith, "Cyclical Learning Rates for Training Neural Networks"

Learning Rate Schedules

Regularization

Many **standard regularization methods** still apply!

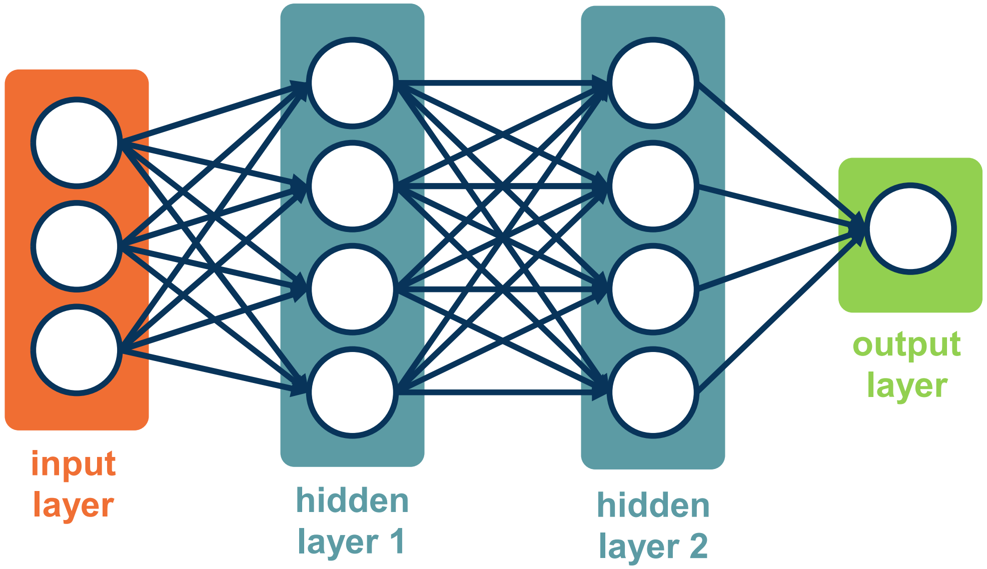
L1 Regularization

$$\boldsymbol{L = |y - Wx_i|^2 + \lambda|W|}$$

where $|\boldsymbol{W}|$ is element-wise

Example regularizations:

- L1/L2 on weights (encourage small values)
- L2: $\boldsymbol{L} = |\boldsymbol{y} - \boldsymbol{Wx_i}|^2 + \boldsymbol{\lambda}|\boldsymbol{W}|^2$ (weight decay)
- Elastic L1/L2: $|\boldsymbol{y} - \boldsymbol{Wx_i}|^2 + \boldsymbol{\alpha}|\boldsymbol{W}|^2 + \boldsymbol{\beta}|\boldsymbol{W}|$

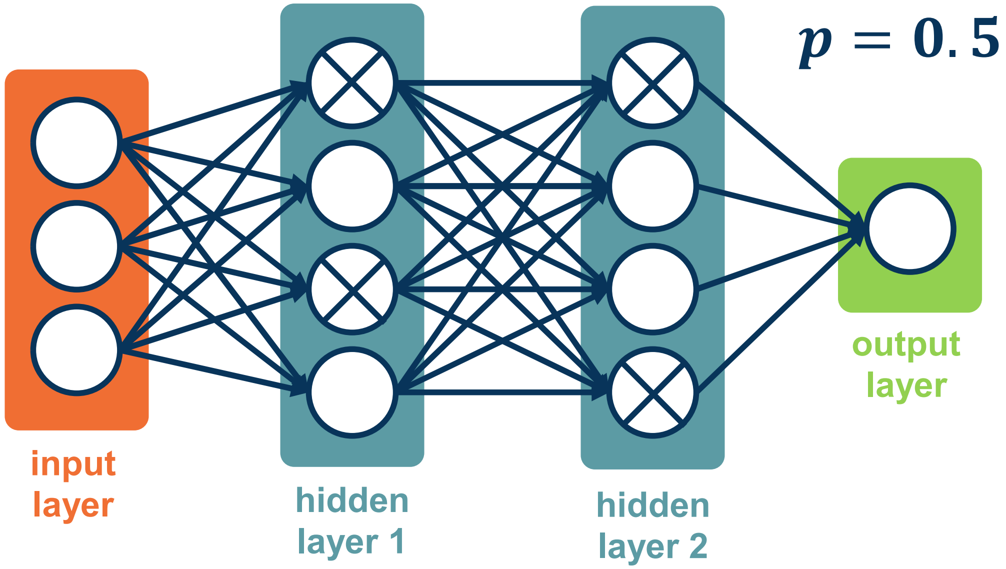


Problem: Network can learn to rely strong on a few features that work really well

- May cause **overfitting** if not representative of test data

From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Preventing Co-Adapted Features



An idea: For each node, keep its output with probability p

- Activations of deactivated nodes are essentially zero

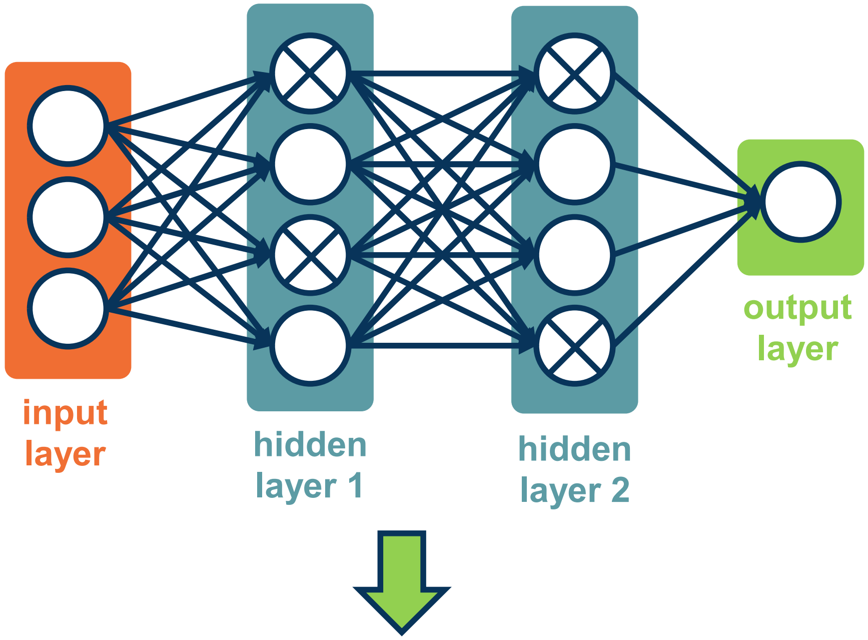
Choose whether to mask out a particular node **each iteration**

From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Dropout Regularization

- In practice, implement with a **mask** calculated each iteration

- During testing, no nodes are dropped

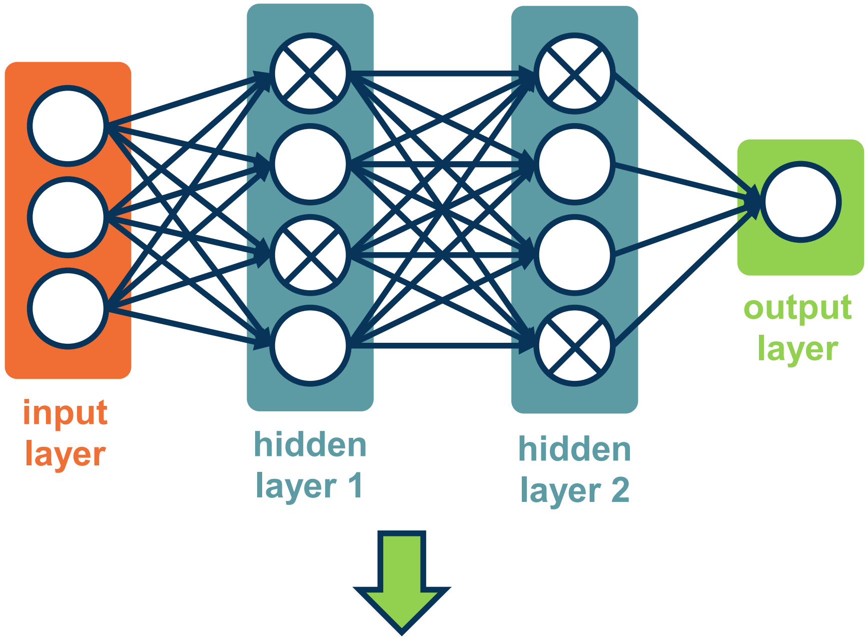


$$\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Dropout Implementation

- During training, each node has an expected $p * fan\_in$ nodes
- During test all nodes are activated
- **Principle:** Always try to have similar train and test-time input/output distributions!



$$\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Solution: During test time, scale outputs (or equivalently weights) by p

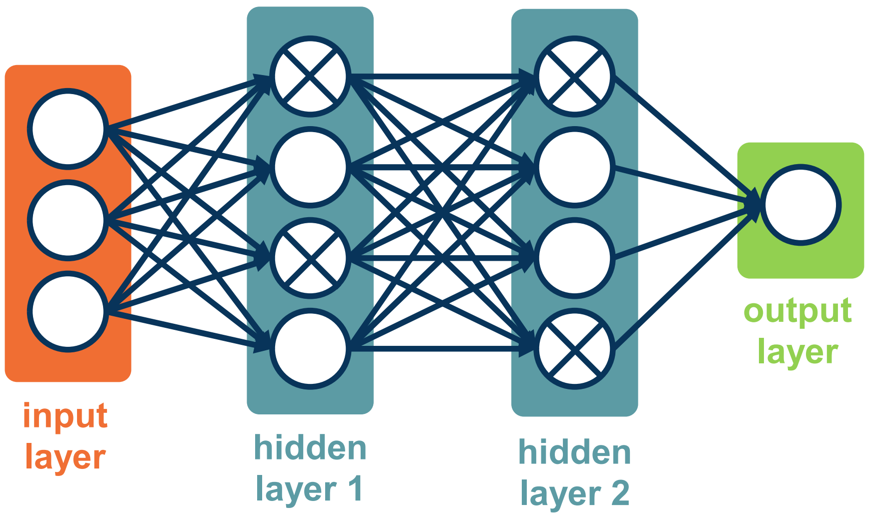
- i.e. $W_{test} = pW$
- Alternative: Scale by $\frac{1}{p}$ at train time

From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Inference with Dropout

Interpretation 1: The model should not rely too heavily on particular features

- If it does, it has probability $\mathbf{1 - p}$ of losing that feature in an iteration



From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

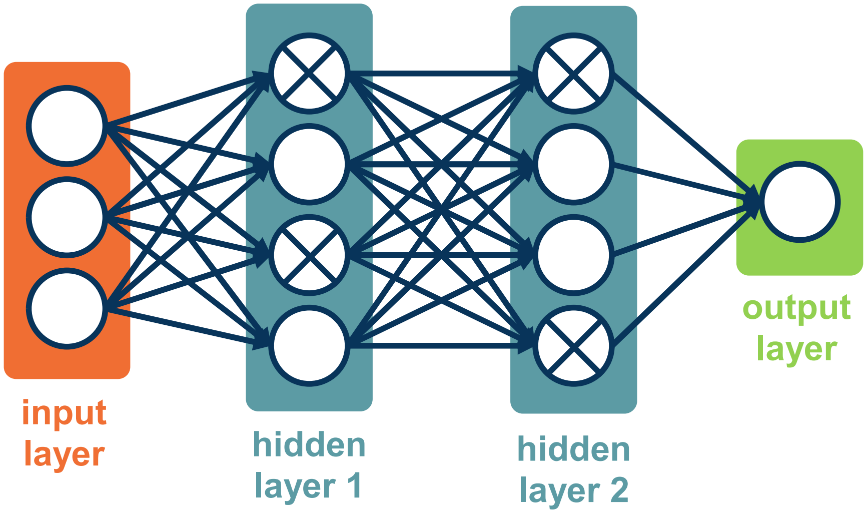
Why Dropout Works

Interpretation 1: The model should not rely too heavily on particular features

- If it does, it has probability $\mathbf{1-p}$ of losing that feature in an iteration

Interpretation 2: Training $\mathbf{2^n}$ networks:

- Each configuration is a network
- Most are trained with 1 or 2 mini-batches of data



From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

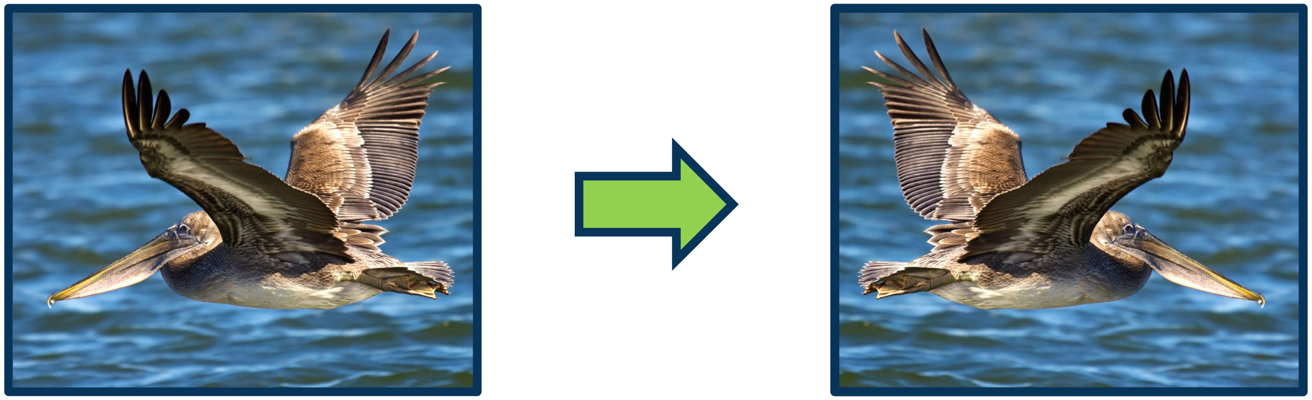
Why Dropout Works

Data Augmentation

Data augmentation – Performing a range of **transformations** to the data

- This essentially “**increases**” your dataset
- Transformations should not change meaning of the data (or label has to be changed as well)

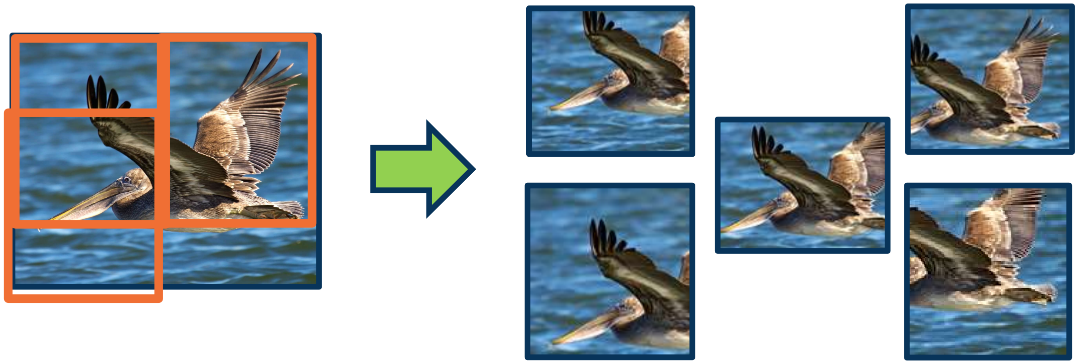
Simple example: Image Flipping



Data Augmentation: Motivation

Random crop

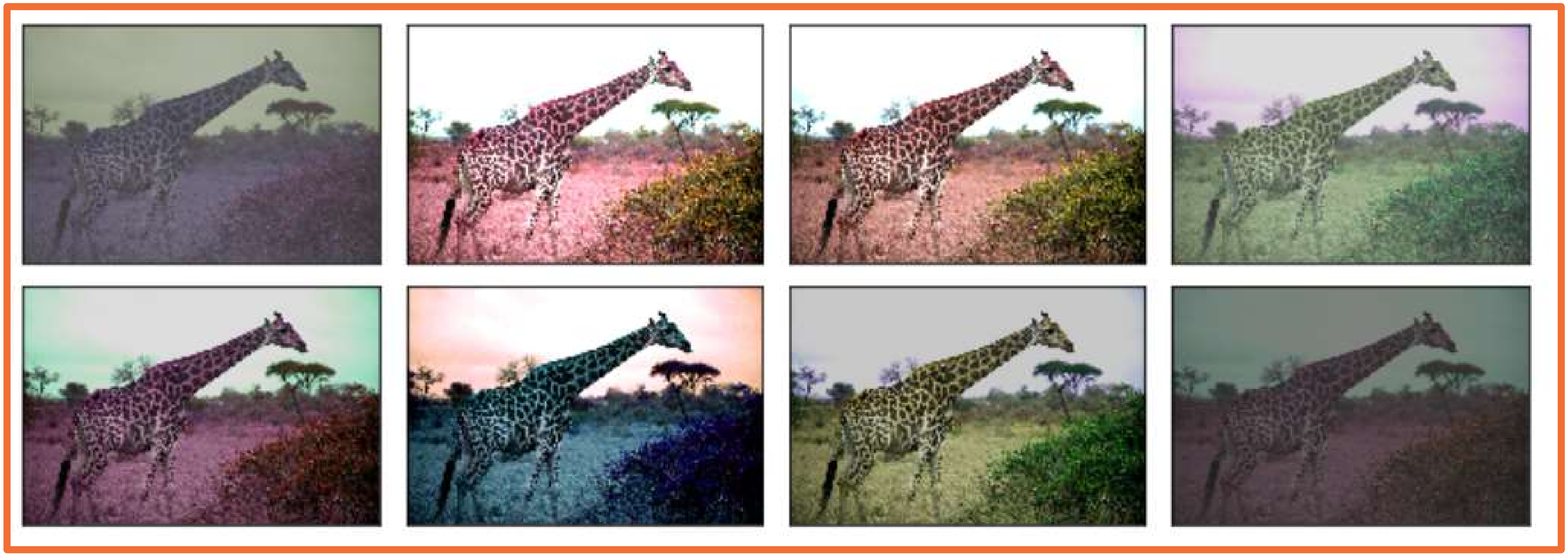
- Take different crops during training
- Can be used during inference too!



CutMix

Random Crop

Color Jitter

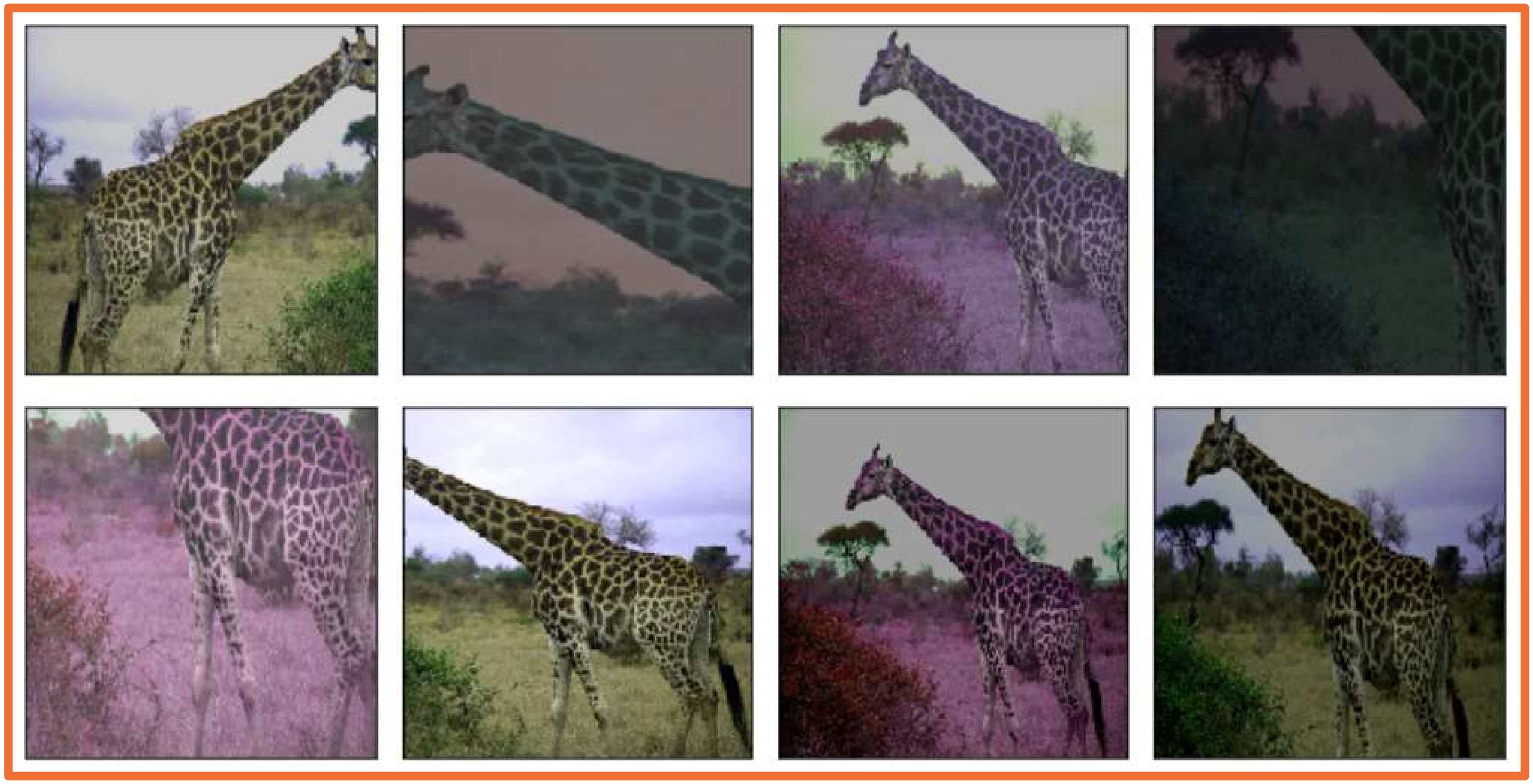


From https://mxnet.apache.org/versions/1.5.0/tutorials/gluon/data_augmentation.html

We can apply **generic affine transformations:**

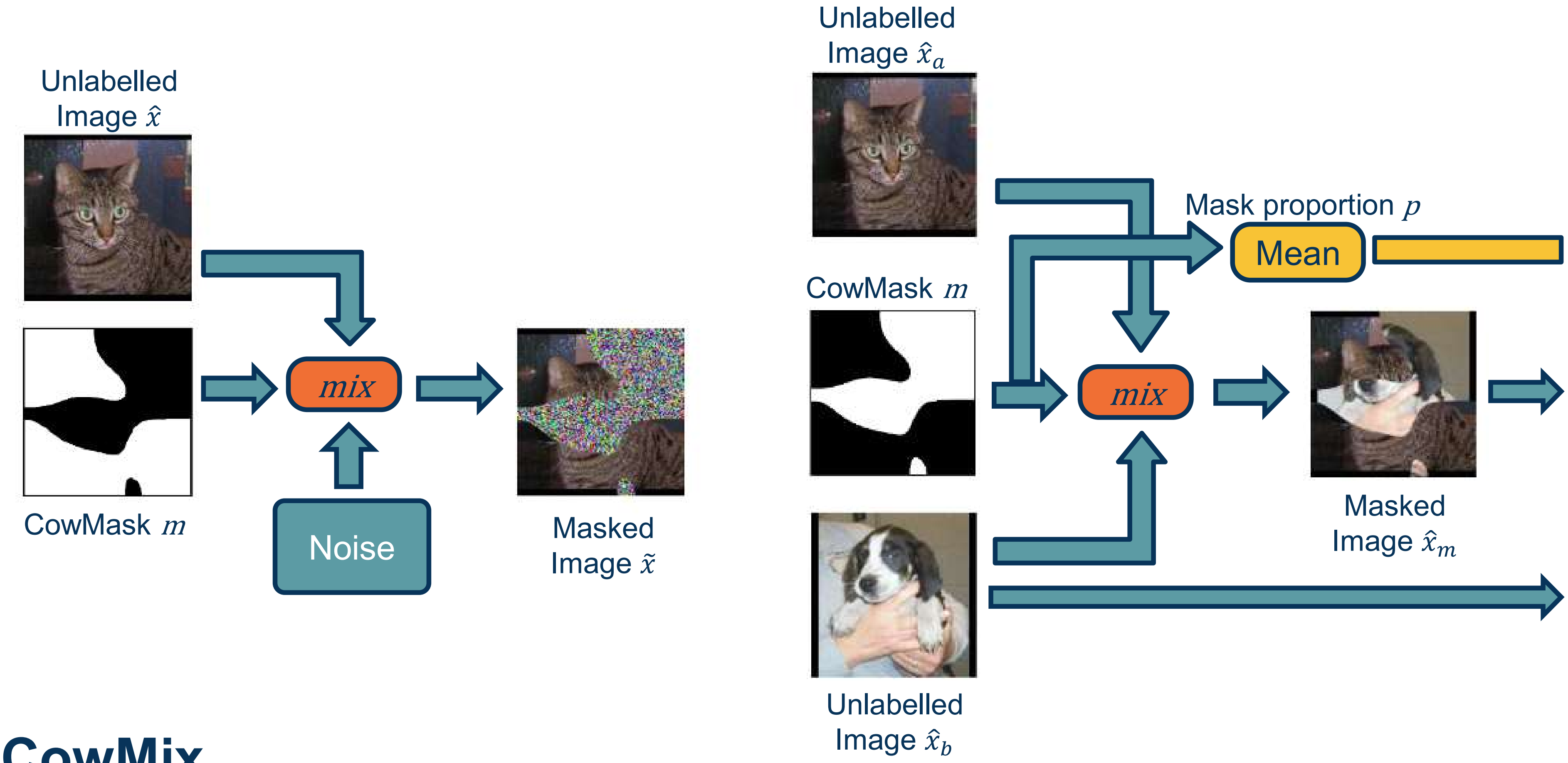
- **Translation**
- **Rotation**
- **Scale**
- **Shear**

We can **combine these transformations** to add even more variety!



From https://mxnet.apache.org/versions/1.5.0/tutorials/gluon/data_augmentation.html

Combining Transformations



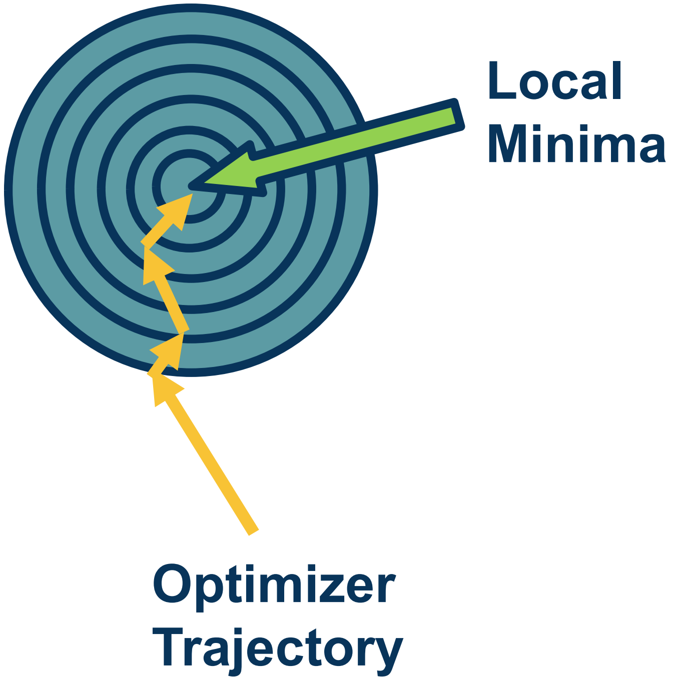
CowMix

From French et al., "Milking CowMask for Semi-Supervised Image Classification"

Other Variations

The Process of Training Neural Networks

- Training deep neural networks is an art form!
- Lots of things matter (together) – the key is to find a combination that works
- **Key principle**: Monitoring everything to understand what is going on!
  - Loss and accuracy curves
  - Gradient statistics/characteristics
  - Other aspects of computation graph



The Process of Training

Proper Methodology

Always start with **proper methodology**!

- **Not uncommon** even in published papers to get this wrong

Separate data into: **Training, validation, test set**

- **Do not look** at test set performance until you have decided on everything (including hyper-parameters)

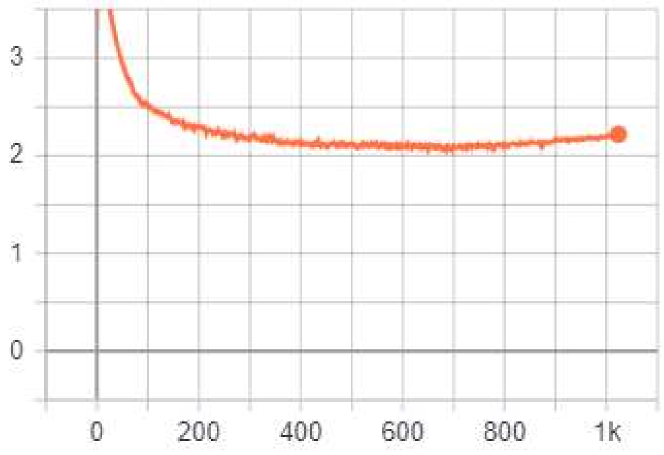
Use **cross-validation** to decide on hyper-parameters if amount of data is an issue

Check the bounds of your loss function

- E.g. cross-entropy ranges from $[\mathbf{0}, \infty]$
- Check initial loss at small random weight values
  - E.g. $-\log(p)$ for cross-entropy, where $p = 0.5$

Another example: Start without regularization and make sure loss goes up when added

Key Principle: Simplify the dataset to make sure your model can properly (over)-fit before applying regularization



Validation Loss

Sanity Checking

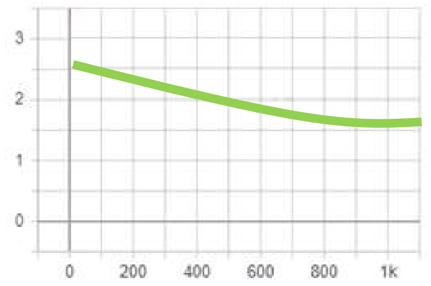
Change in loss indicates speed of learning:

- Tiny loss change -> too small of a learning rate
- Loss (and then weights) turn to NaNs -> too high of a learning rate

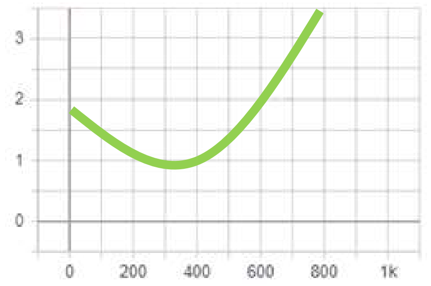
Other bugs can also cause this, e.g.:

- Divide by zero
- Forgetting the log!

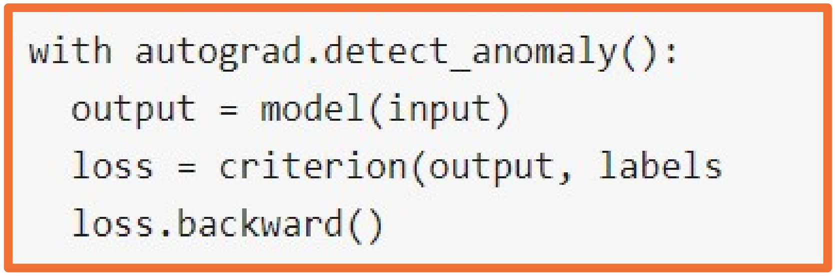
In pytorch, use autograd's detect anomaly to debug



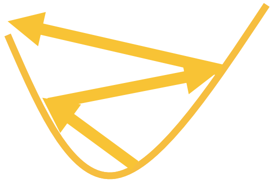
Learning Rate Too Low



Learning Rate Too High

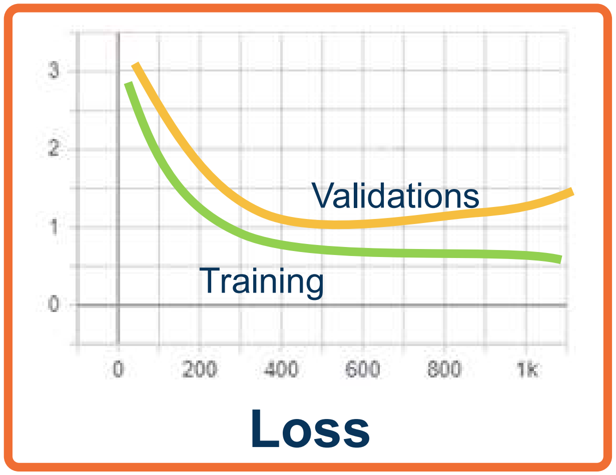


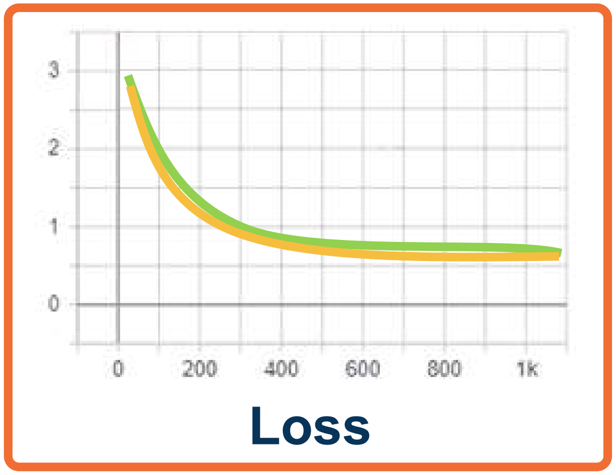
```
with autograd.detect_anomaly():
  output = model(input)
  loss = criterion(output, labels
  loss.backward()
```



Loss and Not a Number (NaN)

- Classic machine learning signs of under/overfitting still apply!
- **Over-fitting**: Validation loss/accuracy starts to get worse after a while
- **Under-fitting:** Validation loss very close to training loss, or both are high
- **Note:** You can have higher training loss!
  - Validation loss has no regularization
  - Validation loss is typically measured at the end of an epoch





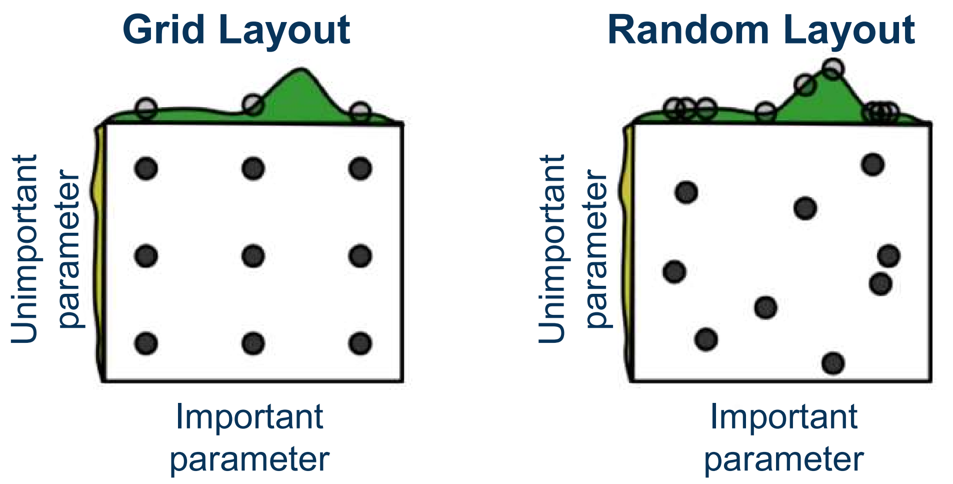
Overfitting

Many hyper-parameters to tune!

- Learning rate, weight decay crucial
- Momentum, others more stable
- **Always tune** hyper-parameters; even a good idea will fail un-tuned!

Start with coarser search:

- E.g. learning rate of {0.1, 0.05, 0.03, 0.01, 0.003, 0.001, 0.0005, 0.0001}
- Perform finer search around good values



From: Bergstra et al., "Random Search for Hyper-Parameter Optimization", JMLR, 2012

Automated methods are OK, but intuition (or random) can do well given enough of a tuning budget

Hyper-Parameter Tuning

Inter-dependence of Hyperparameters

Note that hyper-parameters and even module selection are **interdependent**!

Examples:

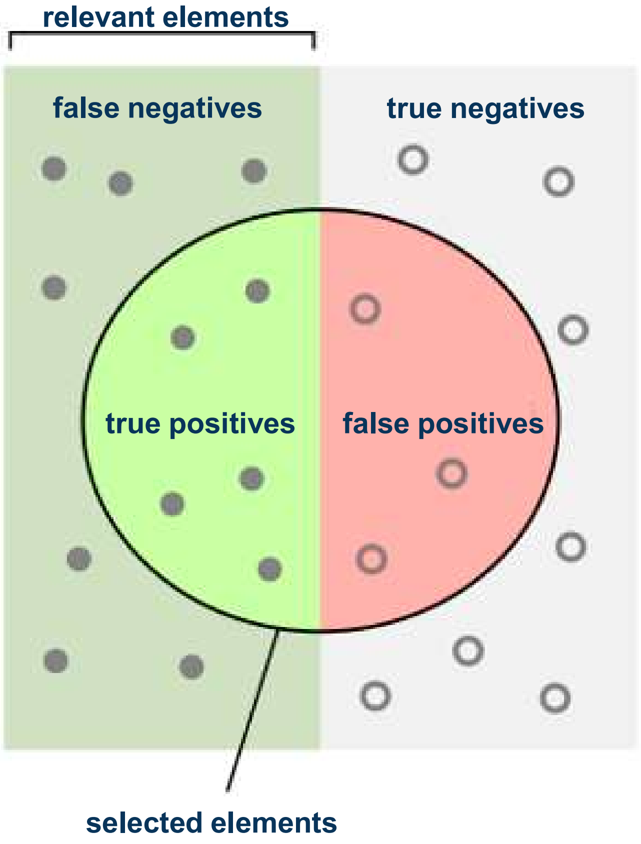
- Batch norm and dropout **maybe not be needed together** (and sometimes the combination is worse)
- The learning rate should be **changed proportionally to batch size** – increase the learning rate for larger batch sizes
  - **One interpretation:** Gradients are more reliable/smoother

Note that we are optimizing a **loss function**

What we actually care about is **typically different metrics that we can't differentiate**:

- Accuracy
- Precision/recall
- Other specialized metrics

The relationship between the two can be complex!



From https://en.wikipedia.org/wiki/Precision_and_recall

Relationship Between Loss and Other Metrics

Simple Example: Cross-Entropy and Accuracy

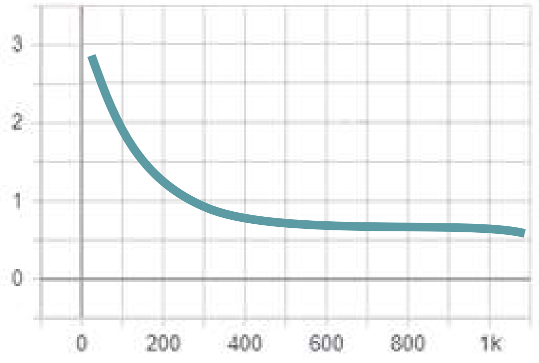
- **Example:** Cross entropy loss

$$L = -log\, P(Y = y_i | X = x_i)$$

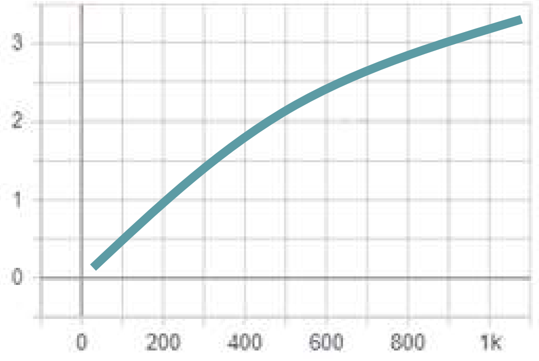
- **Accuracy** is measured based on:

$$argmax_i(P(Y = y_i\,|X = x_i))$$

- Since the correct class score only has to be slightly higher, we can have **flat loss curves but increasing accuracy**!



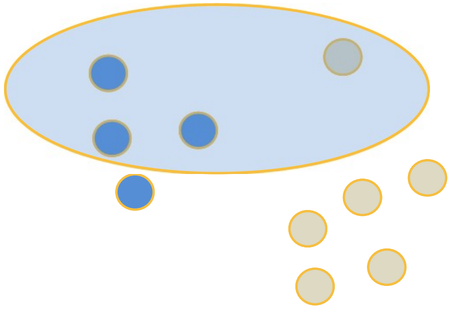
Loss

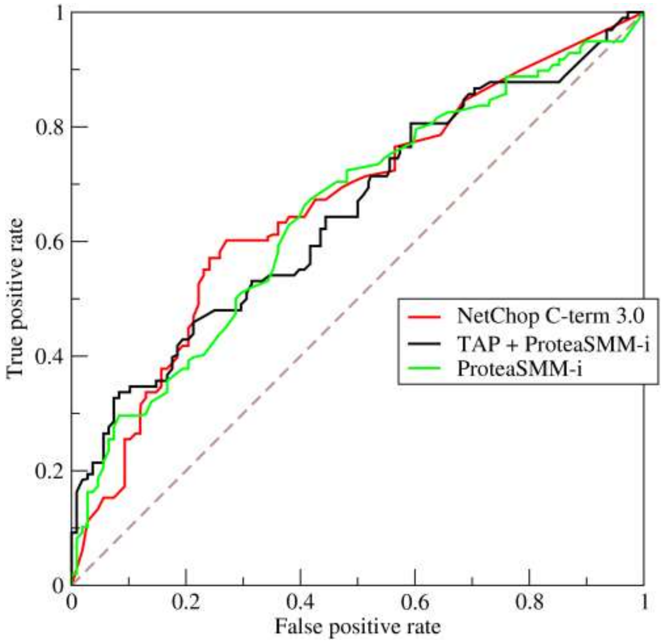


Accuracy

Example: Precision/Recall or ROC Curves

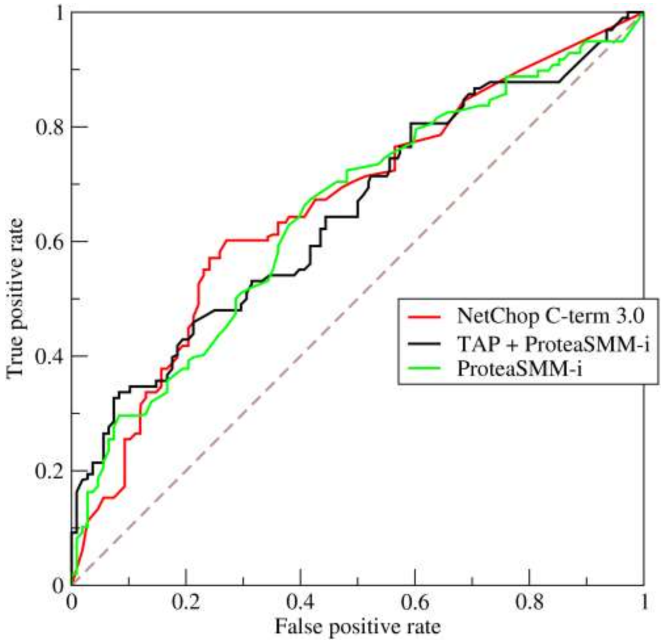
- **Precision/Recall curves** represent the inherent tradeoff between number of positive predictions and correctness of predictions
- **Definitions**
  - True Positive Rate: $\boldsymbol{TPR} = \frac{\boldsymbol{tp}}{\boldsymbol{tp+fn}}$
  - False Positive Rate: $\boldsymbol{FPR} = \frac{\boldsymbol{fp}}{\boldsymbol{fp+tn}}$
  - $\boldsymbol{Accuracy} = \frac{\boldsymbol{tp+tn}}{\boldsymbol{tp+tn+fp+fn}}$





From https://en.wikipedia.org/wiki/Receiver_operating_characteristic

- **Precision/Recall curves** represent the inherent tradeoff between number of positive predictions and correctness of predictions
- **Definitions**
  - True Positive Rate: $\boldsymbol{TPR} = \frac{\boldsymbol{tp}}{\boldsymbol{tp+fn}}$
  - False Positive Rate: $\boldsymbol{FPR} = \frac{\boldsymbol{fp}}{\boldsymbol{fp+tn}}$
  - $\boldsymbol{Accuracy} = \frac{\boldsymbol{tp+tn}}{\boldsymbol{tp+tn+fp+fn}}$
- We can obtain a **curve** by varying the (probability) threshold**:**
  - **Area under the curve (AUC)** common single-number metric to summarize
- Mapping between this and loss is **not simple**!

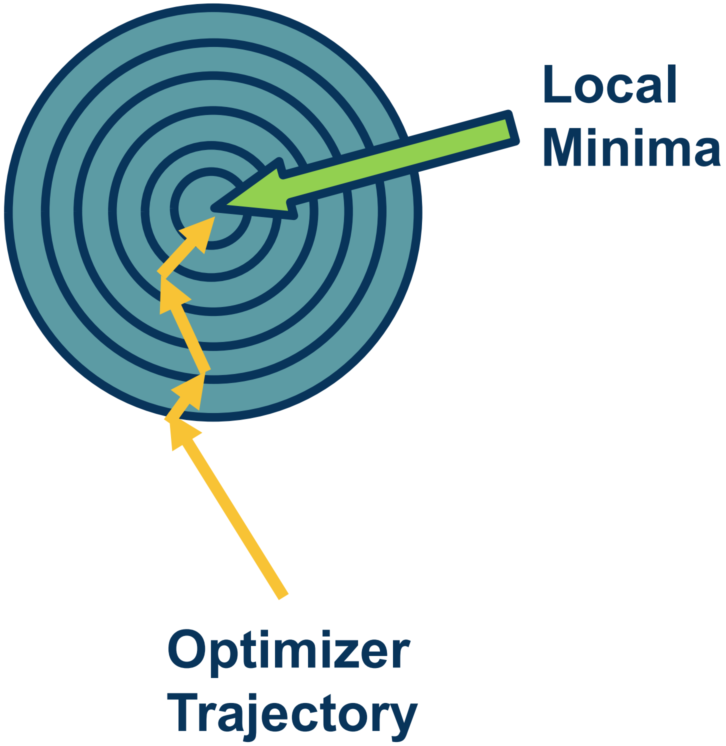


From https://en.wikipedia.org/wiki/Receiver_operating_characteristic

Example: Precision/Recall or ROC Curves

Resource:

- A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay, Leslie N. Smith


Local Minima
Optimizer Trajectory