Topics:

- Optimization

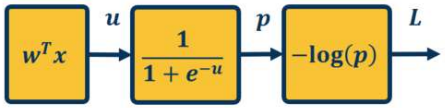
CS 4644-DL / 7643-A
ZSOLT KIRA

- **Assignment 1 – grace period ends today!!!**

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

- **Piazza**: Start with public posts so that others can benefit!
  - Doesn't mean don't post!

- **Facebook Lectures**: Data wrangling video available online, OH recordings available:
  - See dropbox link piazza @9 for lectures, @71 for first office hours
- **Project details coming soon**



$L = 1$

$\bar{p} = \frac{\partial L}{\partial p} = -\frac{1}{p}$

where $p = \sigma(w^T x)$ and $\sigma(x) = \frac{1}{1+e^{-x}}$

$\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u} = \bar{p}\,\sigma(1-\sigma)$

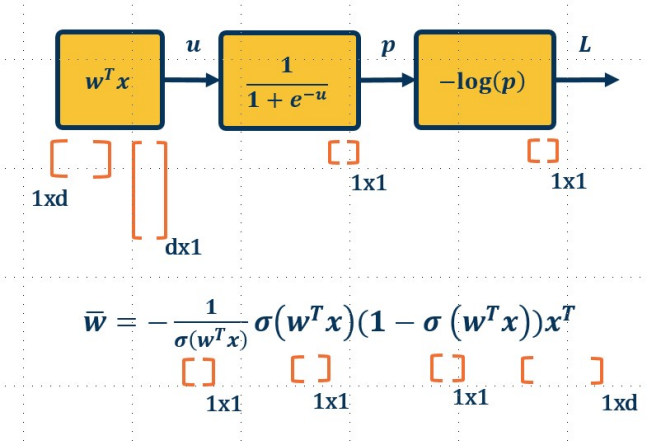
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

We can do this in a combined way to see all terms together:

$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$
$= -\left(1-\sigma(w^T x)\right)x^T$

This effectively shows gradient flow along path from L to w

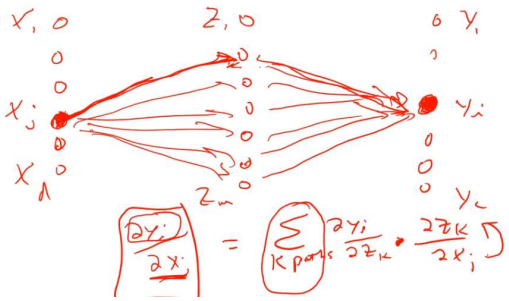
Computation Graph / Global View of Chain Rule



$\bar{w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$

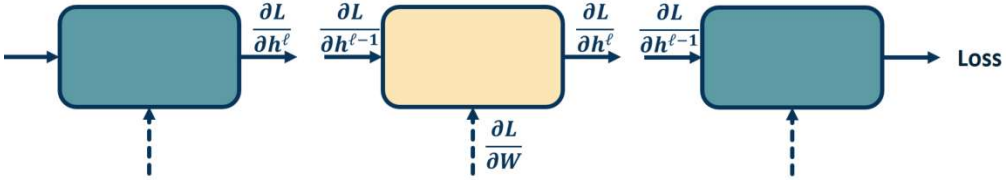
1x1 1x1 1x1 1xd

Computational / Tensor View



Graph View

- We want to to compute: $\{\frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W}\}$



Backpropagation View (Recursive Algorithm)

Different Views of Equivalent Ideas

Georgia Tech

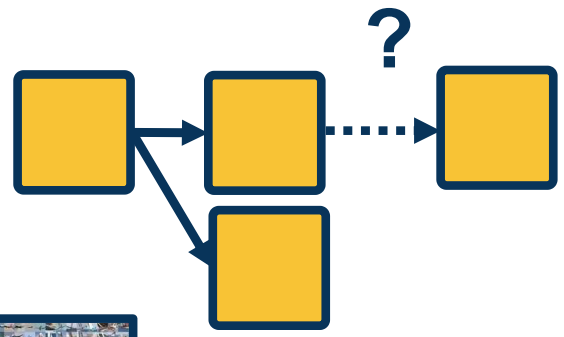
- **Backpropagation:** Recursive, modular algorithm for chain rule + gradient descent

- **When we move to vectors and matrices:**
  - Computation graph (composition of functions) => Multiplication of Jacobians along path
- **Automatic differentiation**:
  - Reduction of modules to simple operations we know (simple multiplication, etc.)
  - Automatically build computation graph in background as write code
  - Automatically compute gradients via backward pass

- **We now have a generic algorithm! Considerations:**
  - **Architecture**
  - **Data Considerations**
  - **Training and Optimization**
  - **Machine Learning Considerations**

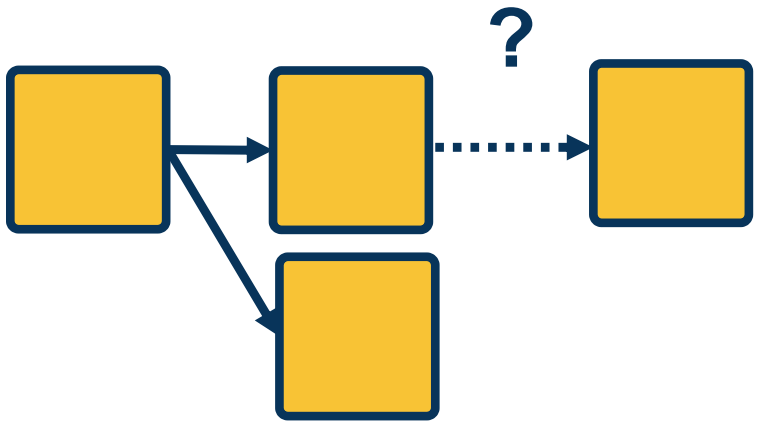
There are still many design decisions that must be made:

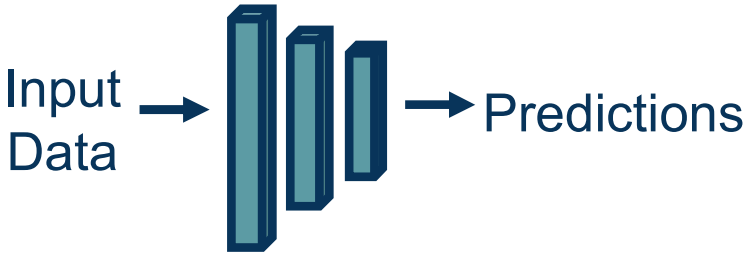
- **Architecture**
- **Data Considerations**
- **Training and Optimization**
- **Machine Learning Considerations**



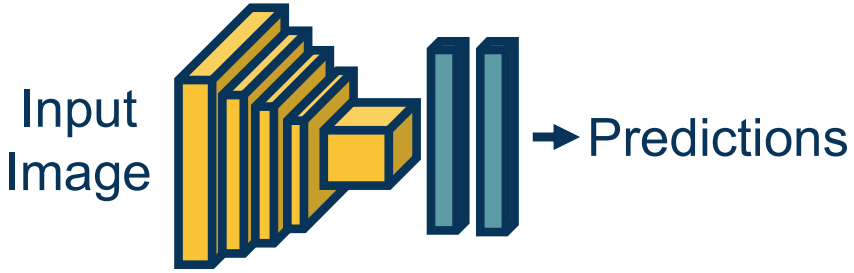
We must design the **neural network architecture:**

- What **modules (layers)** should we use?
- How should they **be connected together**?
- Can we use our **domain knowledge** to add architectural biases?

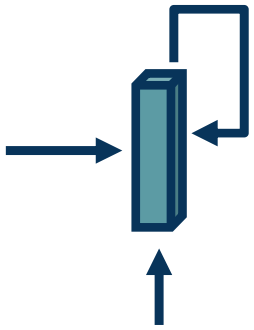




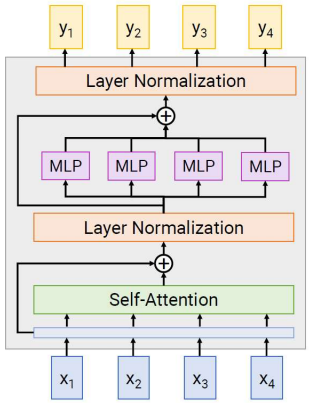
Fully Connected Neural Network



Convolutional Neural Networks



Recurrent Neural Network



Transformers

Different architectures are suitable for different applications or types of input

Example Architectures

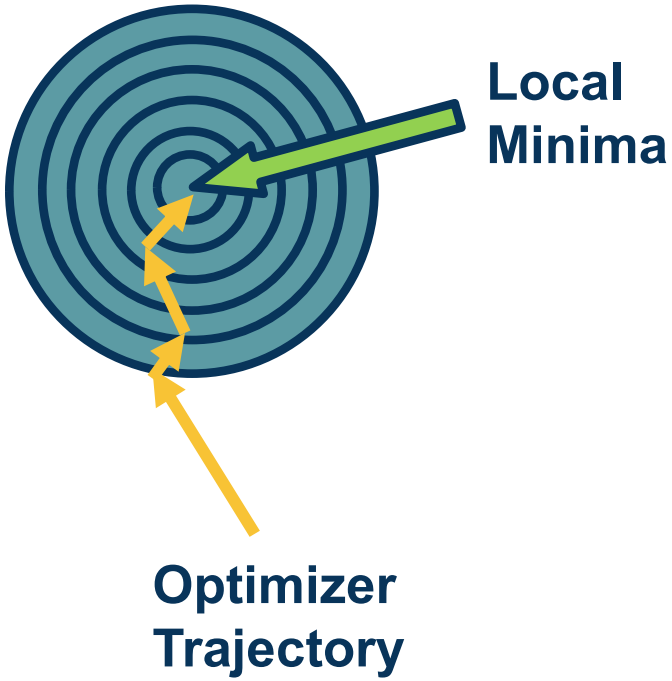
As in traditional machine learning, **data** is key:

- Should we **pre-process** the data?
- Should we **normalize** it?
- Can we **augment** our data by adding noise or other perturbations?

Data Considerations

Even given a good neural network architecture, we need a **good optimization algorithm to find good weights**

- What **optimizer** should we use?
  - Different optimizers make **different weight updates** depending on the gradients
- How should we **initialize** the weights?
- What **regularizers** should we use?
- What **loss function** is appropriate?



Machine Learning Considerations

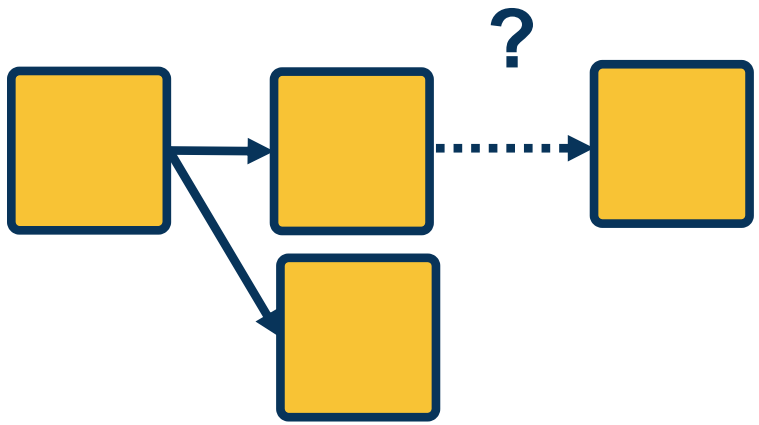
The practice of machine learning is complex: For your particular application you have to **trade off** all of the considerations together

- Trade-off between **model capacity** (e.g. measured by # of parameters) and **amount of data**
- Adding **appropriate biases** based on knowledge of the domain

Architectural Considerations

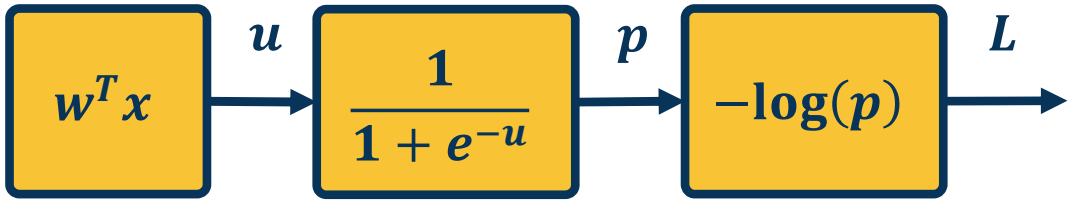
Determining what modules to use, and how to connect them is part of the **architectural design**

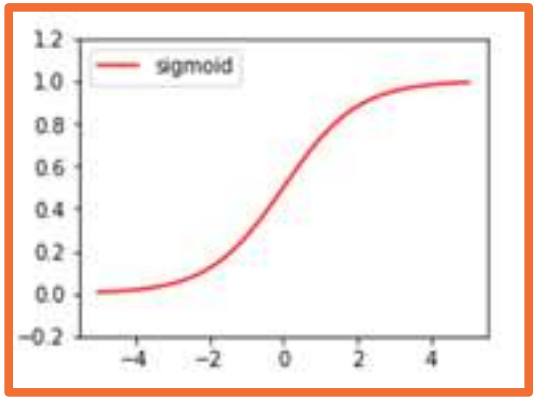
- Guided by the **type of data used** and its **characteristics**
  - Understanding your data is always the first step!
- **Lots of data types (modalities)** already have good architectures
  - Start with what others have discovered!
- **The flow of gradients** is one of the key principles to use when analyzing layers



- **Combination** of linear and non-linear layers
- Combination of **only** linear layers has same representational power as one linear layer
- **Non-linear layers** are crucial
  - Composition of non-linear layers **enables complex transformations of the data**

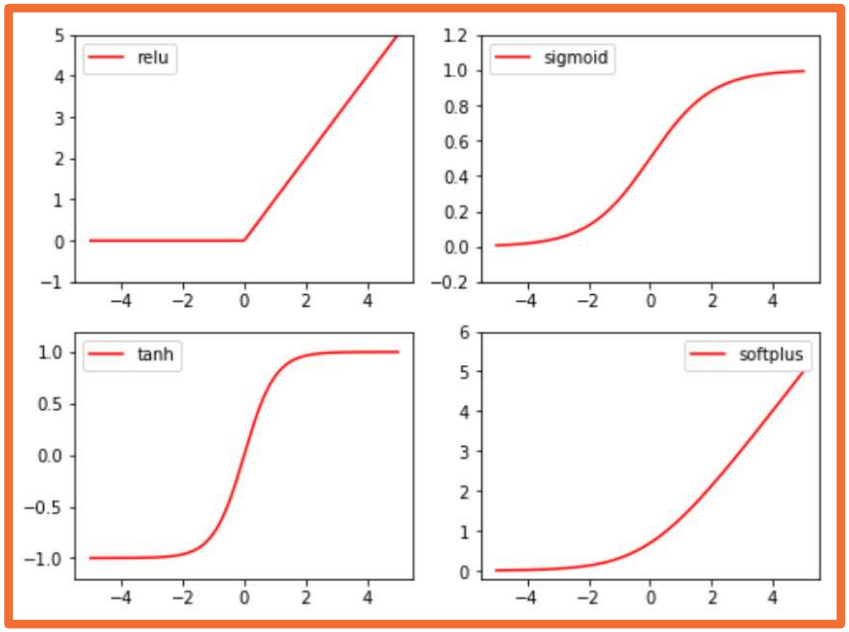
$$w_1^T(w_2^T(w_3^T x)) = w_4^T \mathbf{x}$$



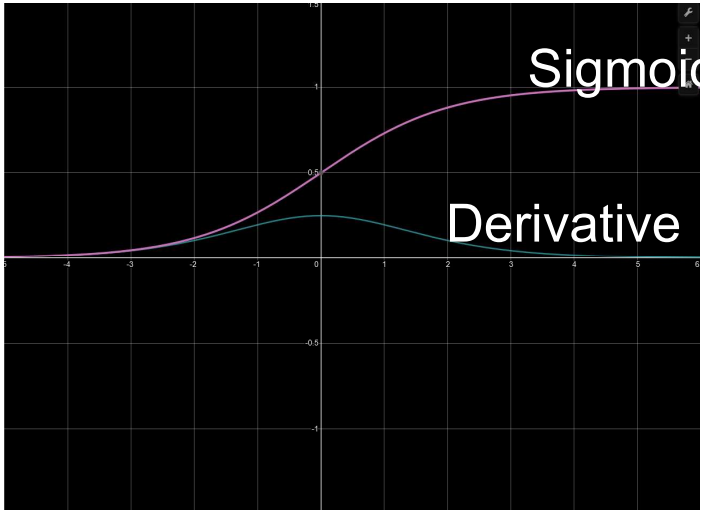


Several aspects that we can **analyze**:

- Min/Max
- Correspondence between input & output statistics
- **Gradients**
  - At initialization (e.g. small values)
  - At extremes
- Computational complexity

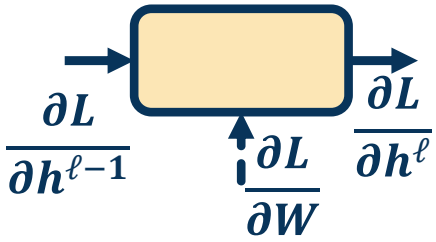


- **Min:** 0, **Max:** 1
- Output **always positive**
- Saturates at **both ends**
- **Gradients**
  - Vanishes at both end
  - Always positive
- **Computation: Exponential term**



$$h^{\ell} = \sigma\left(h^{\ell-1}\right)$$

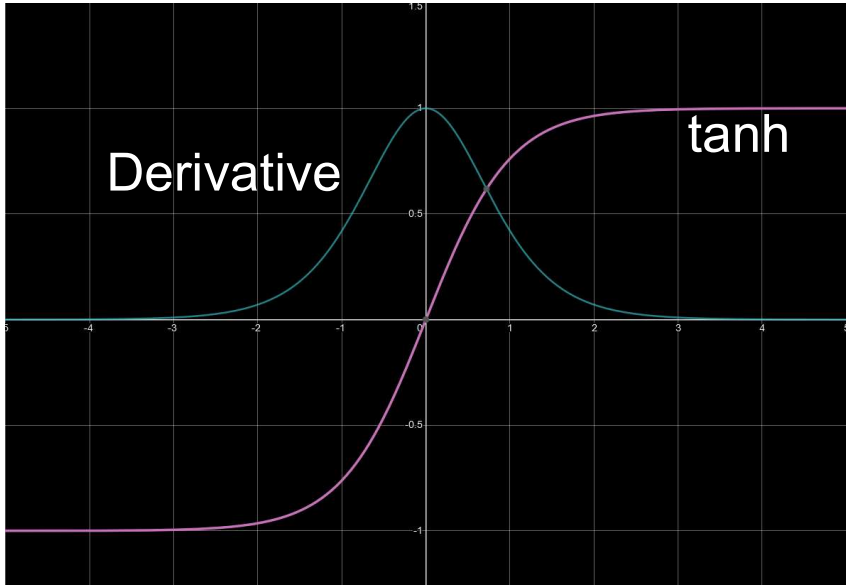
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$$

Sigmoid Function

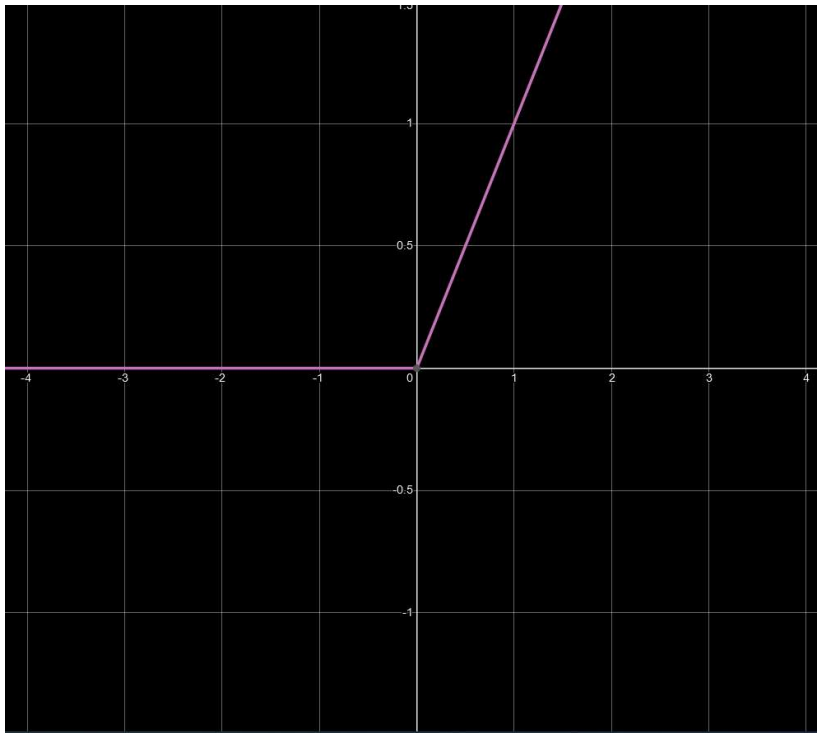
- **Min:** -1, **Max:** 1
  - **Centered**
- Saturates at **both ends**
- **Gradients**
  - Vanishes at both end
  - Always positive
- **Still somewhat computationally heavy**



$$h^{\ell} = tanh(h^{\ell-1})$$

Tanh Function

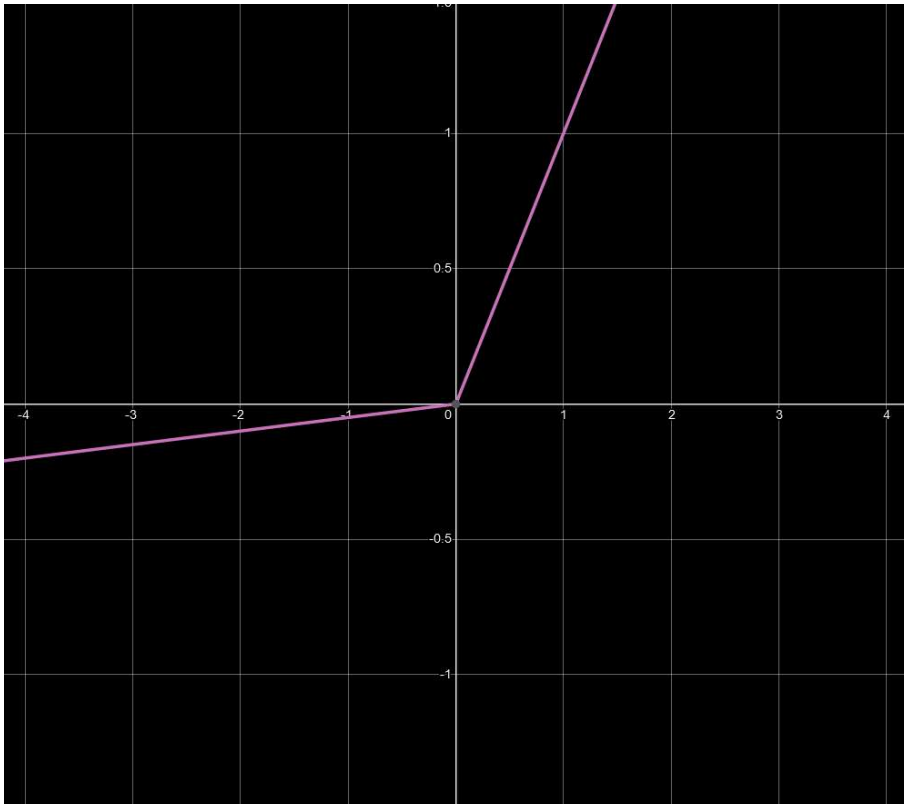
- **Min:** 0, **Max:** Infinity
- Output always **positive**
- **No saturation** on positive end!
- **Gradients**
  - $\mathbf{0}$ if $\mathbf{x} \leq \mathbf{0}$ (dead ReLU)
  - Constant otherwise (does not vanish)
- **Cheap to compute (max)**



$$h^{\ell} = max(0, h^{\ell-1})$$

Rectified Linear Unit

- **Min:** -Infinity, **Max:** Infinity
- **Learnable parameter!**
- **No saturation**
- **Gradients**
  - No dead neuron
- **Still cheap to compute**



$$h^{\ell} = max(\alpha h^{\ell-1}, h^{\ell-1})$$

Leaky ReLU

Selecting a Non-Linearity

Which **non-linearity** should you select?

- Unfortunately, **no one activation function is best** for all applications
- **ReLU** is most common starting point
  - Sometimes leaky ReLU can make a big difference
- **Sigmoid** is typically avoided unless clamping to values from [0,1] is needed

Initialization

Initializing the Parameters

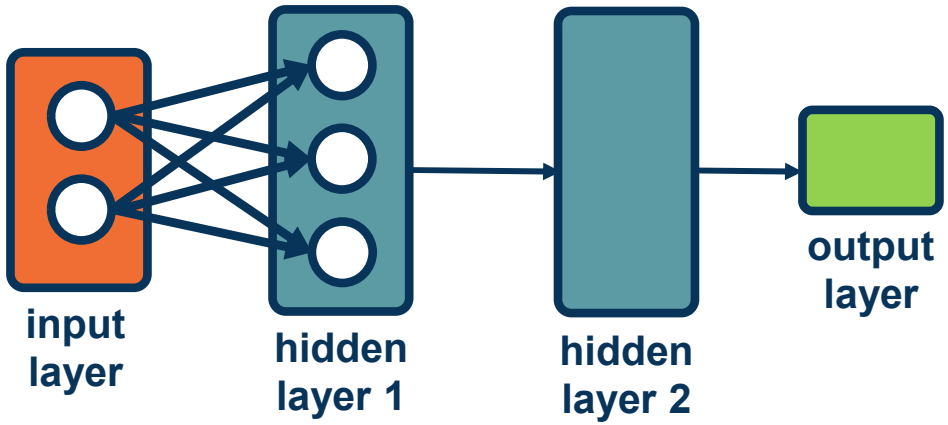
The parameters of our model must be **initialized to something**

- Initialization is **extremely important**!
  - Determines how **statistics of outputs** (given inputs) behave
  - Determines how well **gradients flow** in the beginning of training (important)
  - Could **limit use of full capacity** of the model if done improperly
- Initialization that is **close to a good (local) minima** will converge faster and to a better solution

Initializing values to a constant value leads to **a degenerate solution**!

- What happens to the **weight updates?**
- Each node has the same input from previous layers so gradients **will be the same**
- As a results**, all weights will be updated** to the same exact values

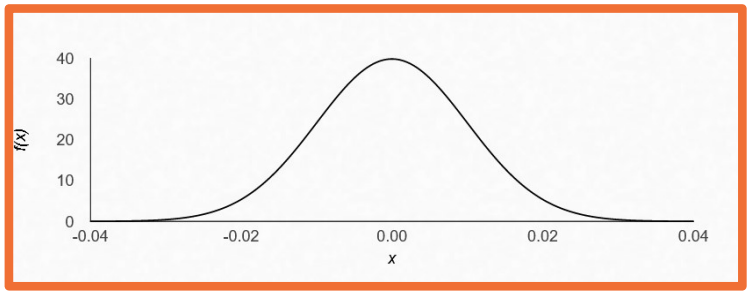
$$w_i = c \quad \forall i$$

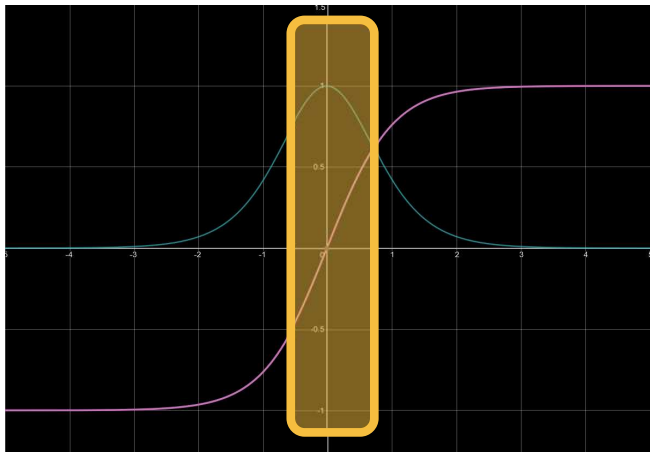


A Poor Initialization

Common approach is **small normally distributed random numbers**

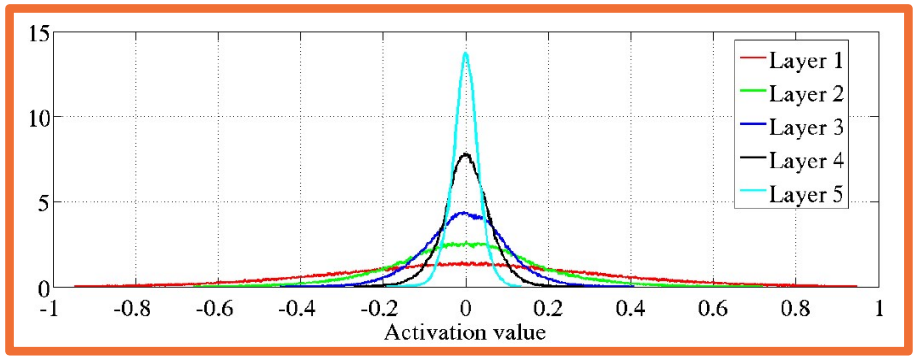
- E.g. $\boldsymbol{N(\mu, \sigma)\ where\ \mu = 0, \sigma = 0.01}$
- **Small weights** are preferred since no feature/input has prior importance
- Keeps the model within the **linear region of most activation functions**





Deeper networks (with many layers) are more sensitive to initialization

- With a deep network, **activations (outputs of nodes) get smaller**
  - Standard deviation reduces significantly
- **Leads to small updates –** smaller values multiplied by upstream gradients



Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

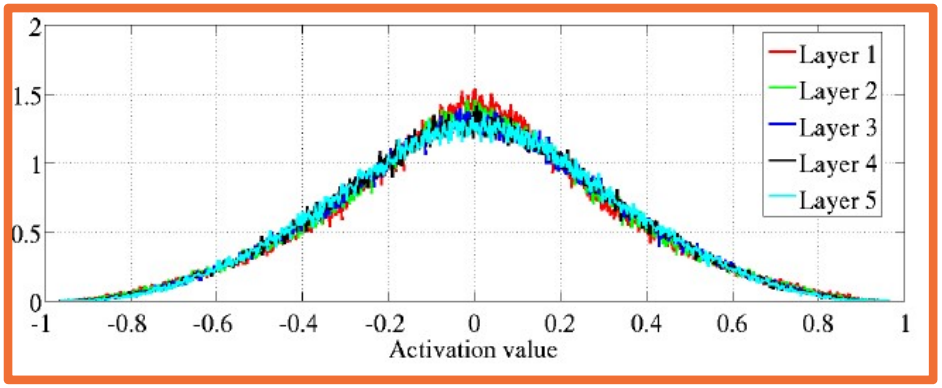
From "Understanding the difficulty of training deep feedforward neural networks." AISTATS, 2010.

Ideally, we'd like to maintain the variance at the output to be similar to that of input!

- This condition leads to a **simple initialization rule**, sampling from uniform distribution:

$$\text{Uniform}\left(-\frac{\sqrt{6}}{n_j+n_{j+1}}, +\frac{\sqrt{6}}{n_j+n_{j+1}}\right)$$

  - Where n_j is **fan-in** (number of input nodes) and n_{j+1} is **fan-out** (number of output nodes)



Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

From "Understanding the difficulty of training deep feedforward neural networks." AISTATS, **2010.**

In practice, **simpler versions** perform empirically well:

$$N(0,1) * \sqrt{\frac{1}{n_j}}$$

- This analysis holds for **tanh or similar activations**.
- Similar analysis for **ReLU activations** leads to:

$$N(0,1) * \sqrt{\frac{1}{n_j/2}}$$

"Delving Deep into Rectifiers:Surpassing Human-Level Performance on ImageNet Classification", ICCV, 2015.

Summary

Key takeaway: **Initialization matters!**

- Determines the **activation** (output) statistics, and therefore **gradient statistics**
- If gradients are **small**, no learning will occur and no improvement is possible!
- Important to reason about **output/gradient statistics** and analyze them for new layers and architectures

Normalization, Preprocessing, and Augmentation

Importance of Data

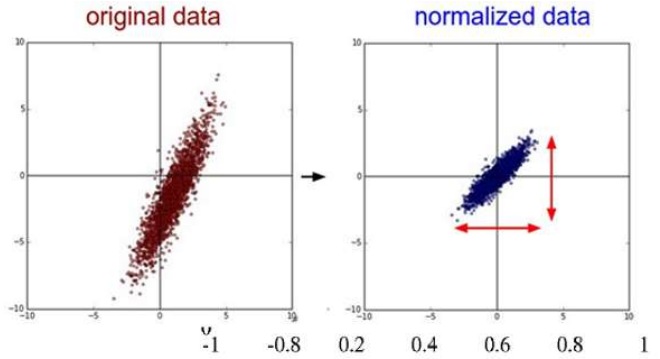
In deep learning, **data drives learning** of features and classifier

- Its **characteristics** are therefore extremely important
- Always **understand your data**!
- **Relationship** between output statistics, layers such as non-linearities, and gradients is important

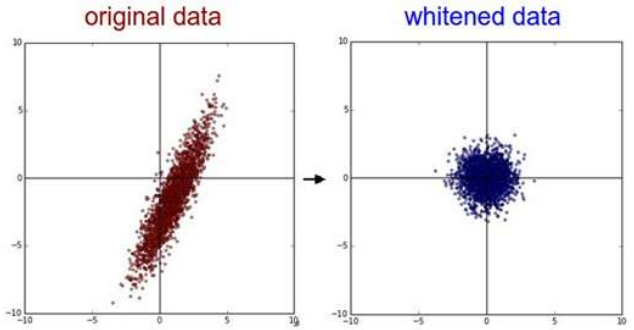
Just like initialization, **normalization** can **improve gradient flow and learning**

Typically **normalization methods** apply:

- Subtract mean, divide by standard deviation **(most common)**
  - This can be done **per** dimension
- Whitening, e.g. through Principle Component Analysis (PCA) **(not common)**



Data after subtracting mean, dividing by standard deviation



Data after whitening

Figure from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

- We can try to come up with a *layer* that can normalize the data across the neural network
- **Given:** A mini-batch of data $[\boldsymbol{B} \times \boldsymbol{D}]$ where $\boldsymbol{B}$ is batch size
- Compute mean and variance **for each dimension $\boldsymbol{d}$**

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

From: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy

Making Normalization a Layer

Normalize data

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Note: This part does not involve new parameters

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

From: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy

- We can give the model flexibility through **learnable parameters γ (scale) and β (shift)**
- Network can learn to **not normalize** if necessary!
- This layer is called a **Batch Normalization (BN) layer**

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

From: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy

Learnable Scaling and Offset

Some Complexities of BN

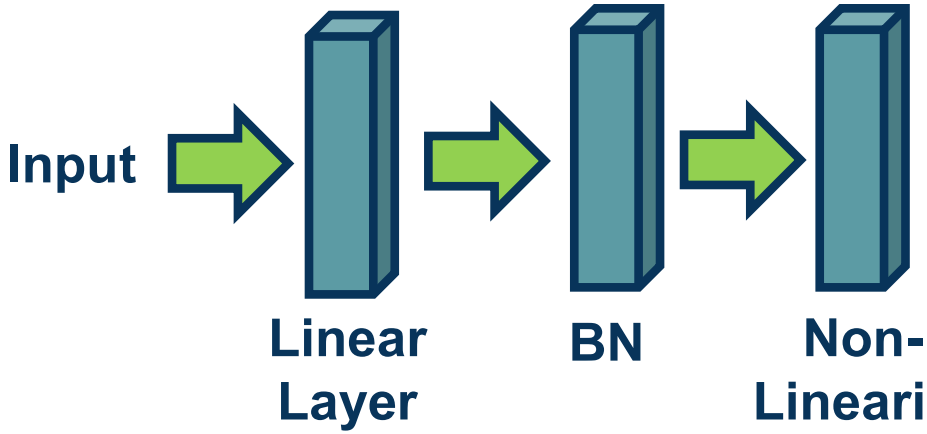
During inference, stored mean/variances calculated on training set are used

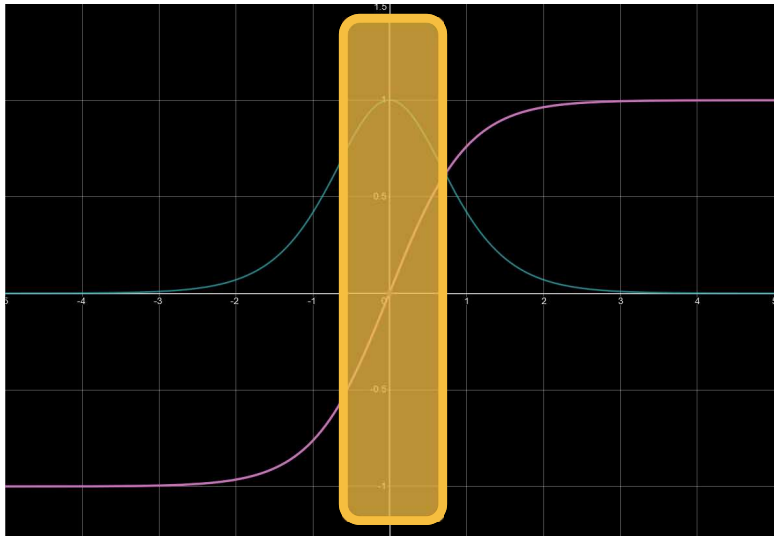
Sufficient batch sizes must be used to get stable per-batch estimates during training

- This is especially an issue when **using multi-GPU or multi-machine training**
- **Use torch.nn.SyncBatchNorm** to estimate batch statistics in these settings

Normalization especially important before **non-linearities**!

- Very low/high values (un-normalized/imbalanced data) cause **saturation**





Generalization of BN

There are **many variations of batch normalization**

- See Convolutional Neural Network lectures for an example

Resource:

- **ML Explained - Normalization**

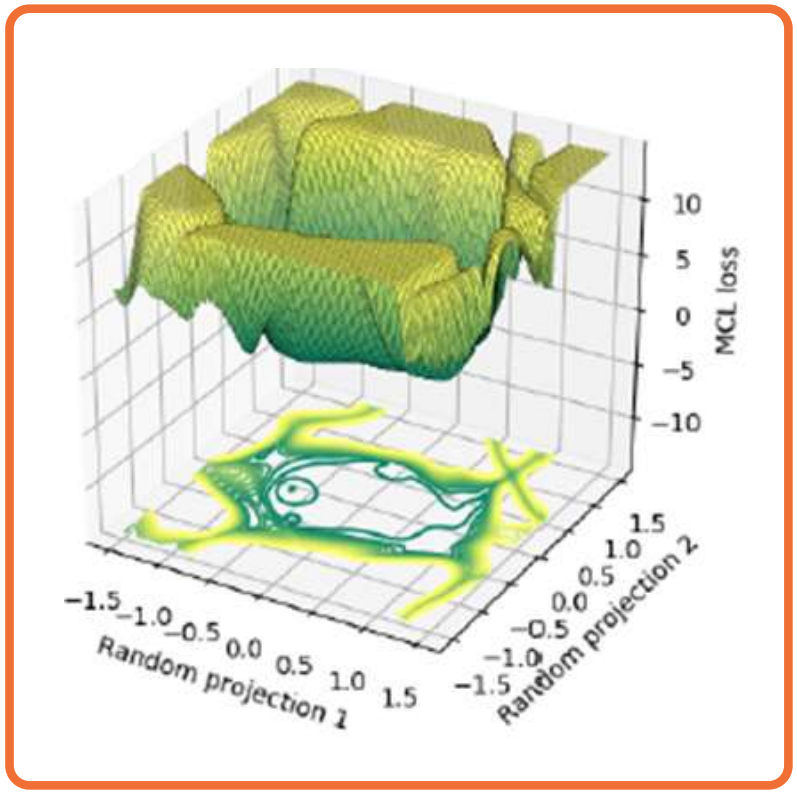
Optimizers

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)

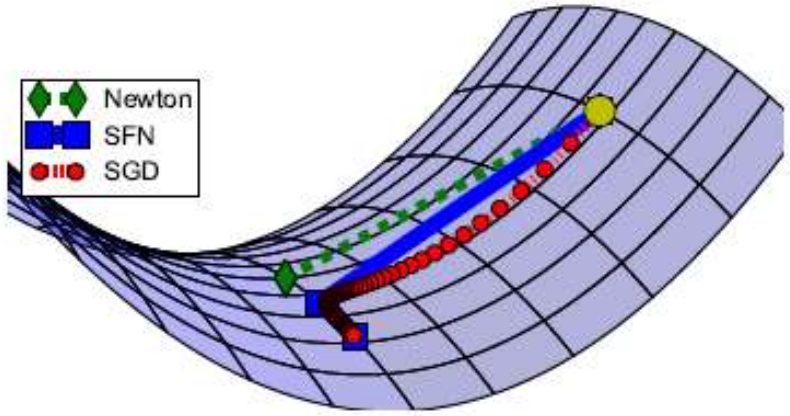


Loss Landscape

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

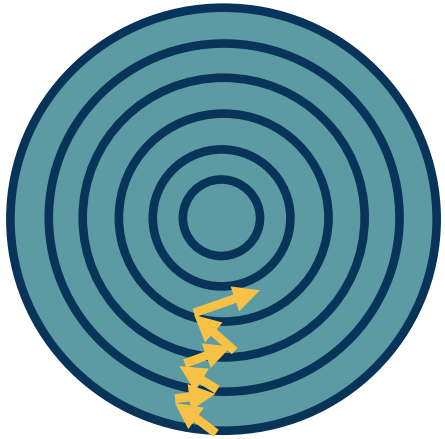


Saddle Point

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

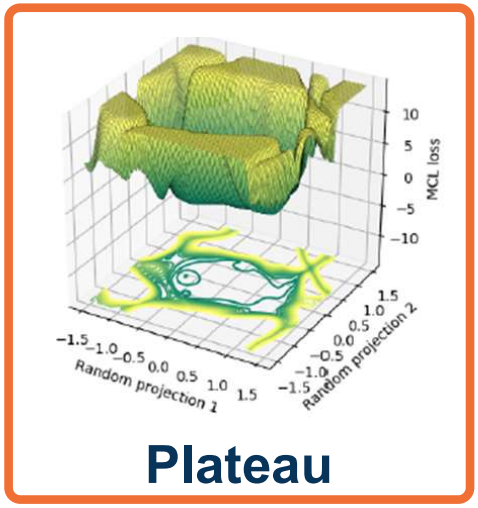
Georgia Tech

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

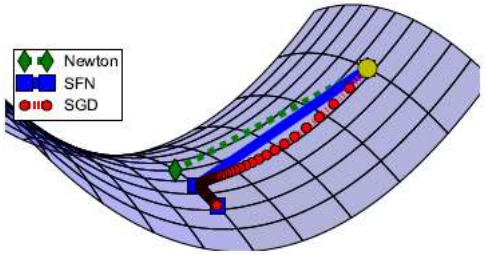
Several types of minima: Local minima, plateaus, saddle points

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

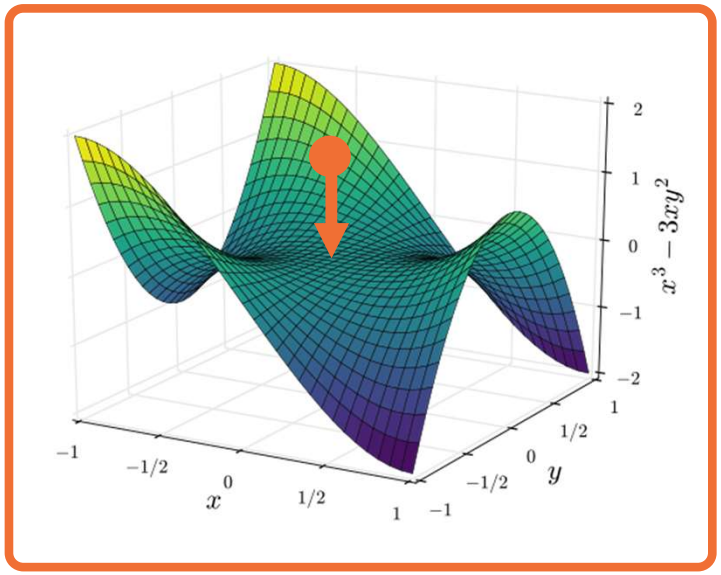
- Gradient descent takes a step in the **steepest direction** (negative gradient)
- **Intuitive idea:** Imagine a ball rolling down loss surface, and use **momentum** to pass flat surfaces

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

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial w_{i-1}}$$ **Update Velocity (starts as 0, $\beta = 0.99$)**

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

- Generalizes SGD ($\beta = 0$)



Adding Momentum

Georgia Tech

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

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

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

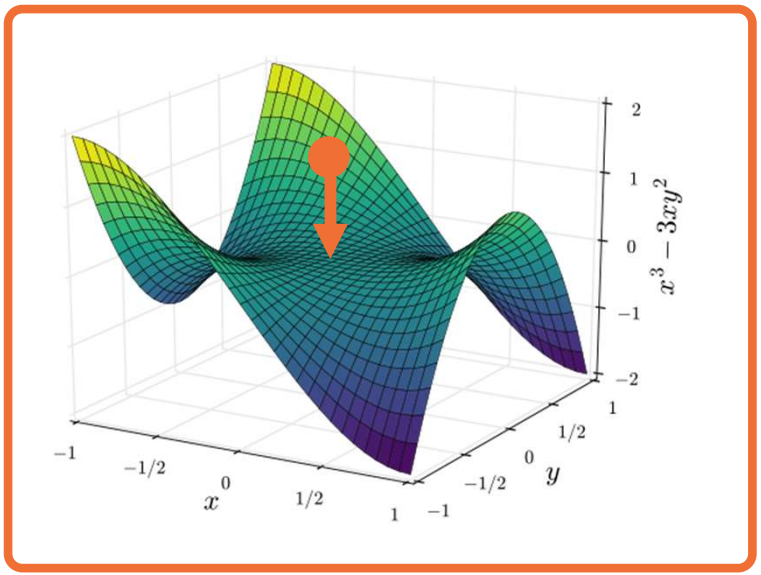
$$= \beta^2 v_{i-2} + \beta \frac{\partial L}{\partial w_{i-2}} + \frac{\partial L}{\partial w_{i-1}}$$

- 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**



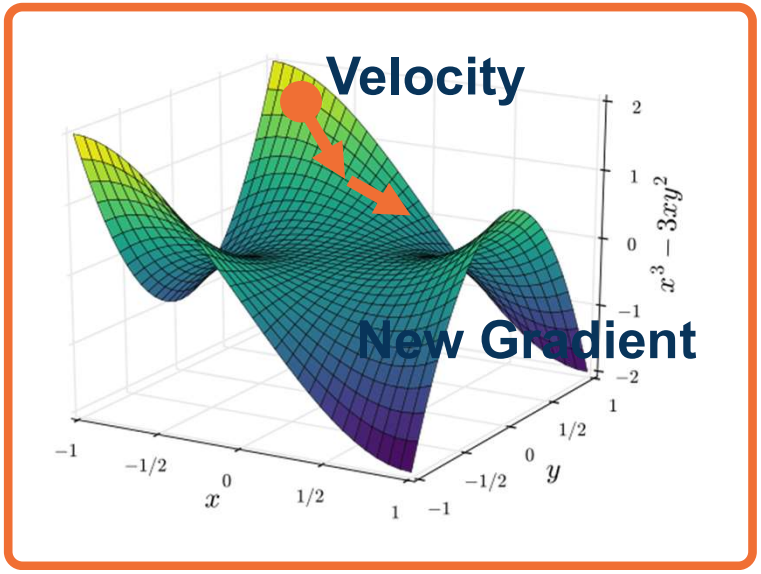
Key idea: Rather than combining velocity with current gradient, go along velocity **first** and then calculate gradient at new point

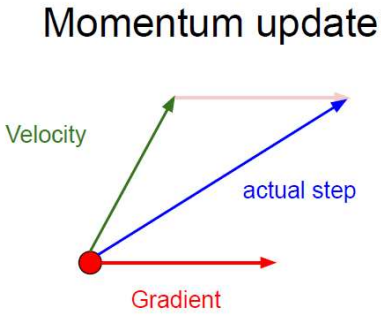
- We know velocity is probably a **reasonable direction**

$$\widehat{\boldsymbol{w}}_{\boldsymbol{i-1}} = \boldsymbol{w}_{\boldsymbol{i-1}} + \boldsymbol{\beta v}_{\boldsymbol{i-1}}$$

$$\boldsymbol{v}_{\boldsymbol{i}} = \boldsymbol{\beta v}_{\boldsymbol{i-1}} + \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial \widehat{w}}_{\boldsymbol{i-1}}}$$

$$\boldsymbol{w}_{\boldsymbol{i}} = \boldsymbol{w}_{\boldsymbol{i-1}} - \boldsymbol{\alpha}\, \boldsymbol{v}_{\boldsymbol{i}}$$





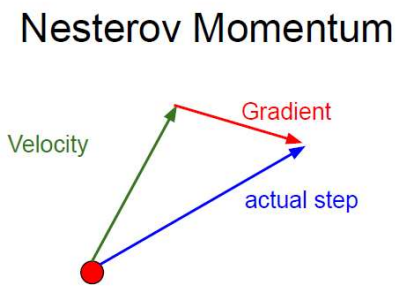


Figure Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

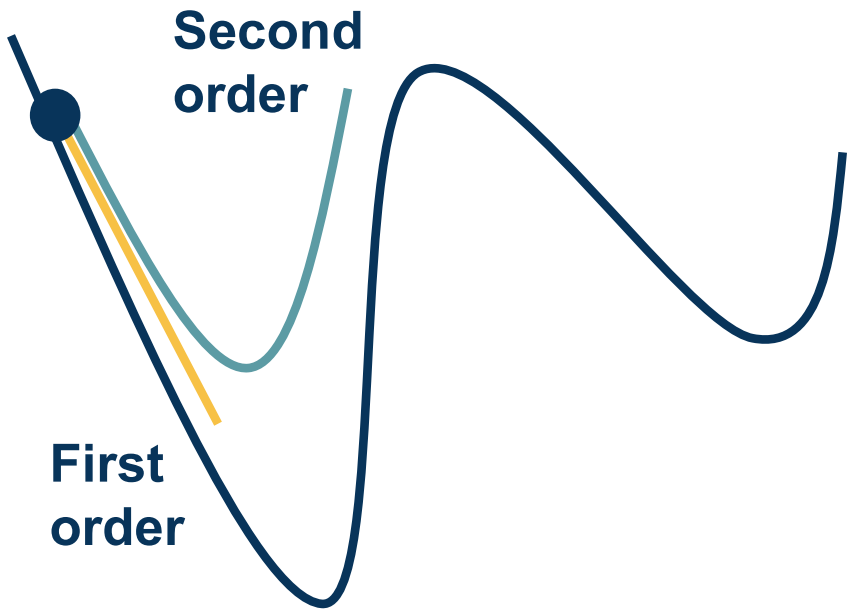
Momentum

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

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



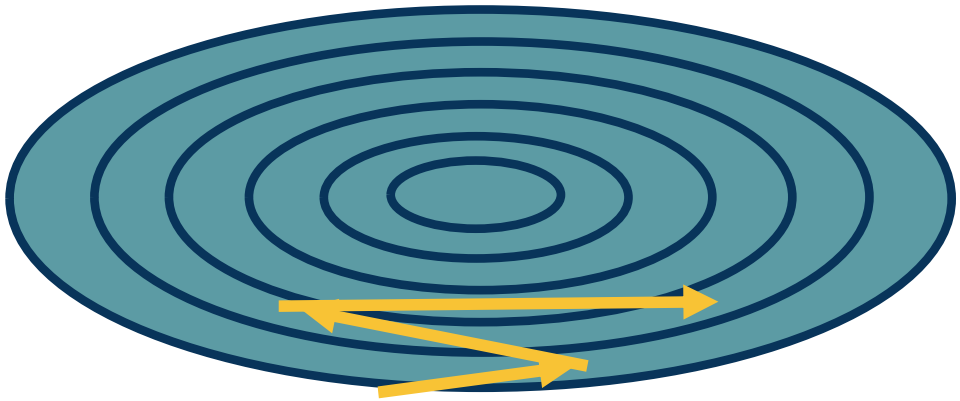
- Gives us information about the **curvature of the loss surface**

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



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

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

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

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

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

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)

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

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

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

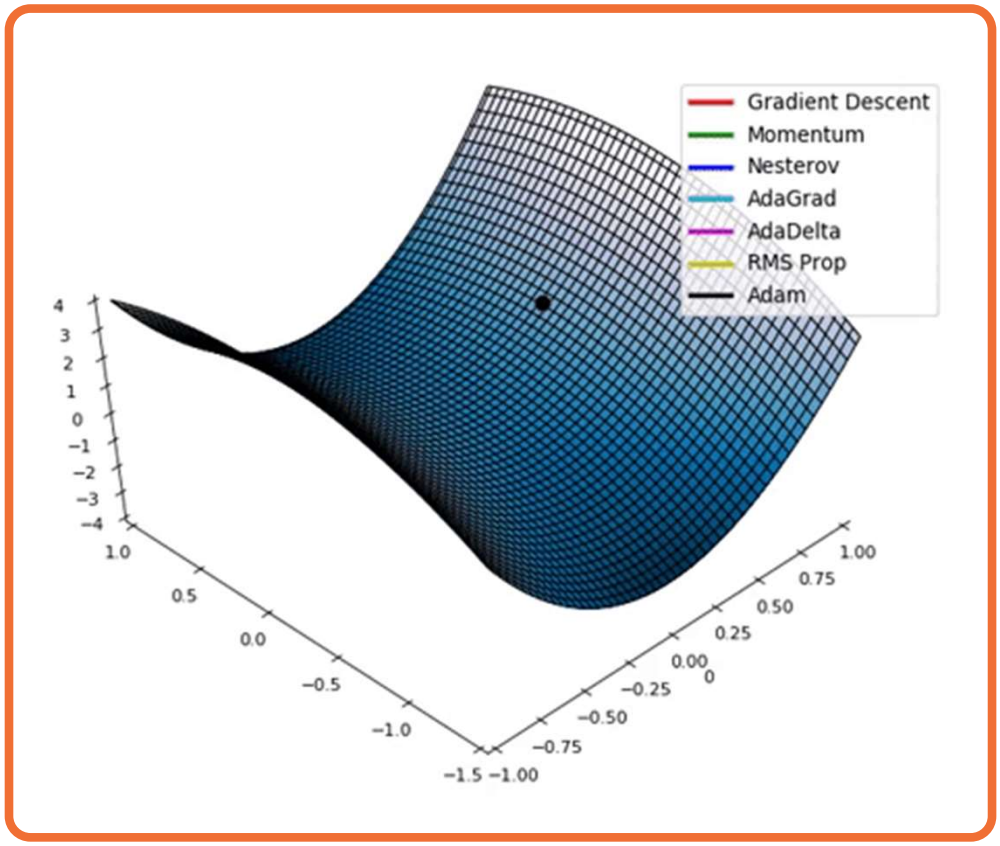
$$w_i = w_{i-1} - \frac{\alpha\, \hat{v}_i}{\sqrt{\hat{G}_i + \epsilon}}$$

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*



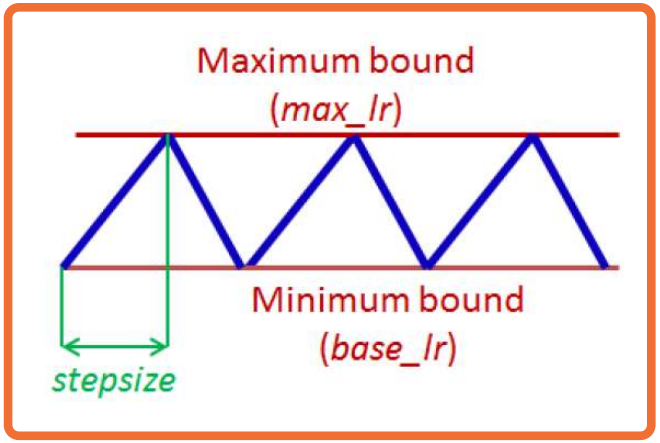
From: https://mlfromscratch.com/optimizers-explained/#/

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"

Regularization

Many **standard regularization methods** still apply!

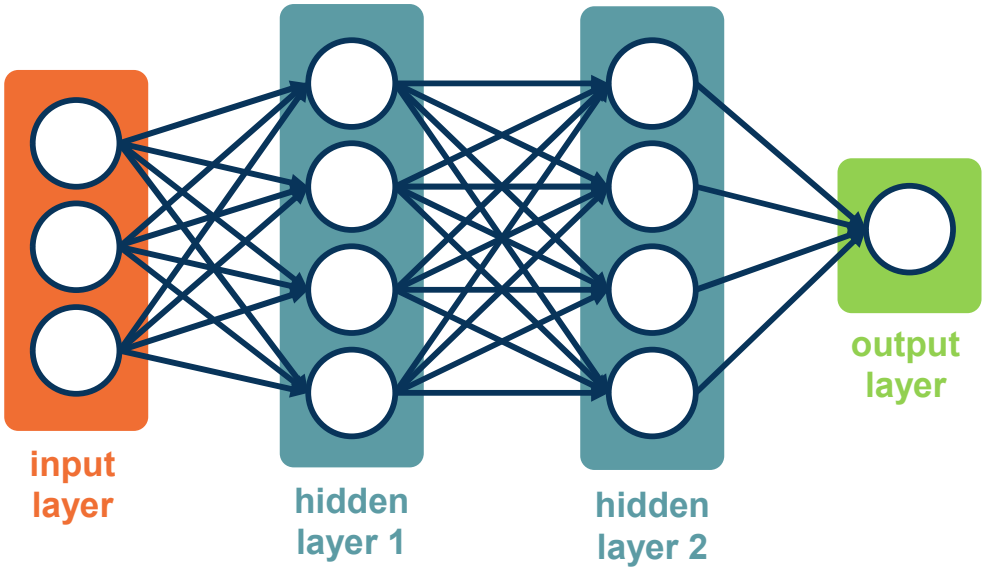
L1 Regularization

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

where $|W|$ is element-wise

Example regularizations:

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

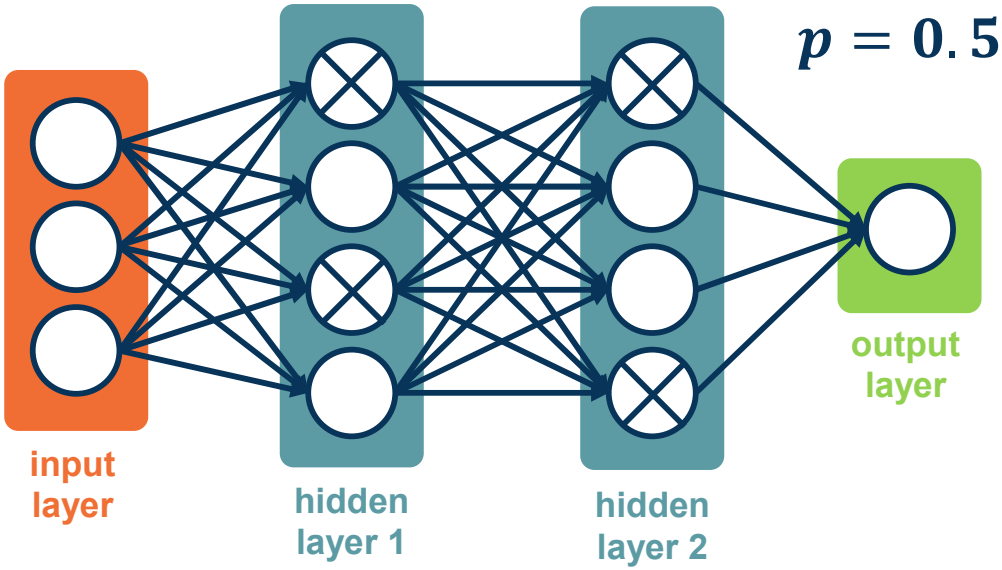


Problem: Network can learn to rely strong on a few features that work really well

- 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

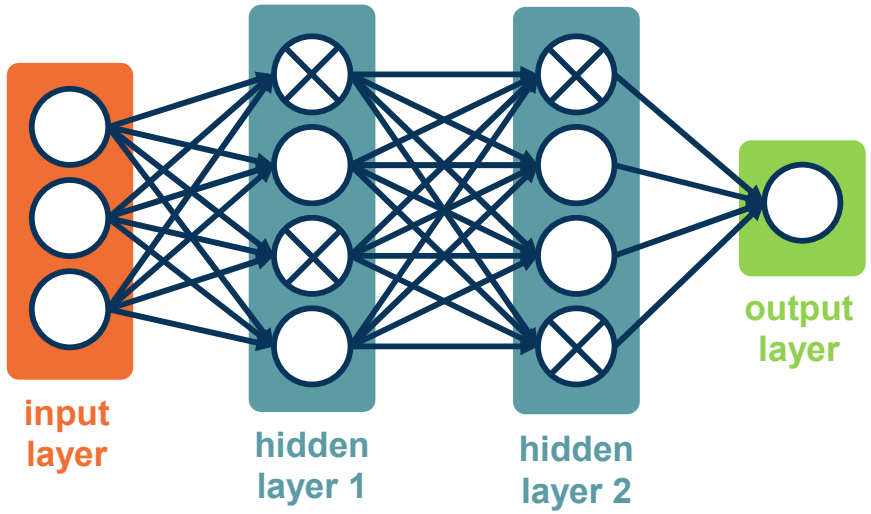
Choose whether to mask out a particular node **each iteration**

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

Dropout Regularization

Dropout Implementation

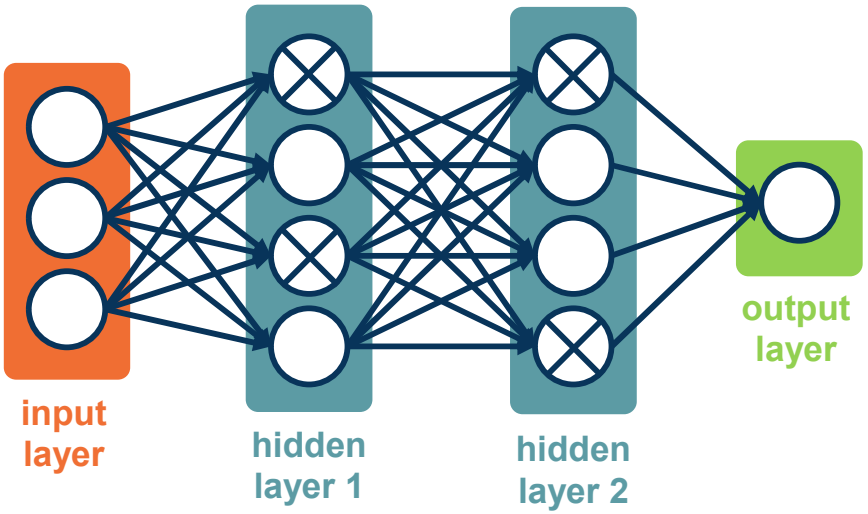
- In practice, implement with a **mask** calculated each iteration
- During testing, no nodes are dropped



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

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

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



Solution: During test time, scale outputs (or equivalently weights) by $\boldsymbol{p}$

- i.e. $\boldsymbol{W_{test} = pW}$

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

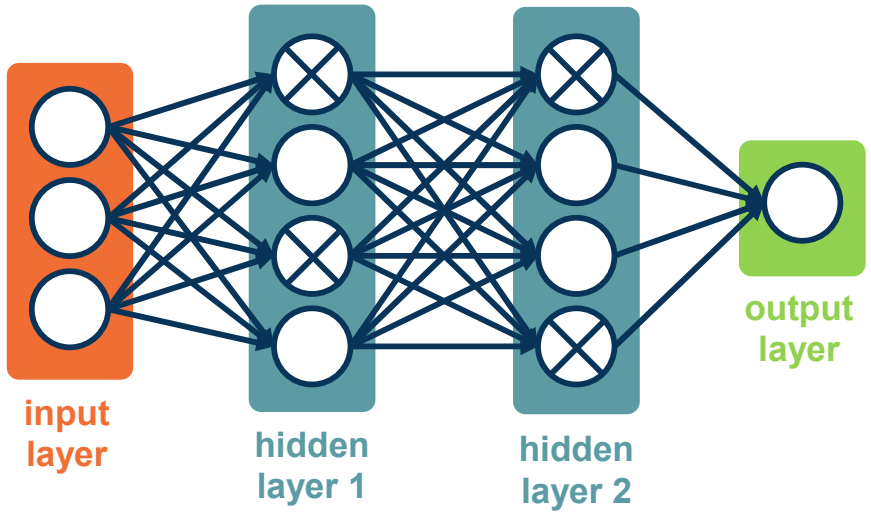
- Alternative: Scale by $\frac{1}{p}$ at train time

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 $\mathbf{1} - \boldsymbol{p}$ of losing that feature in an iteration



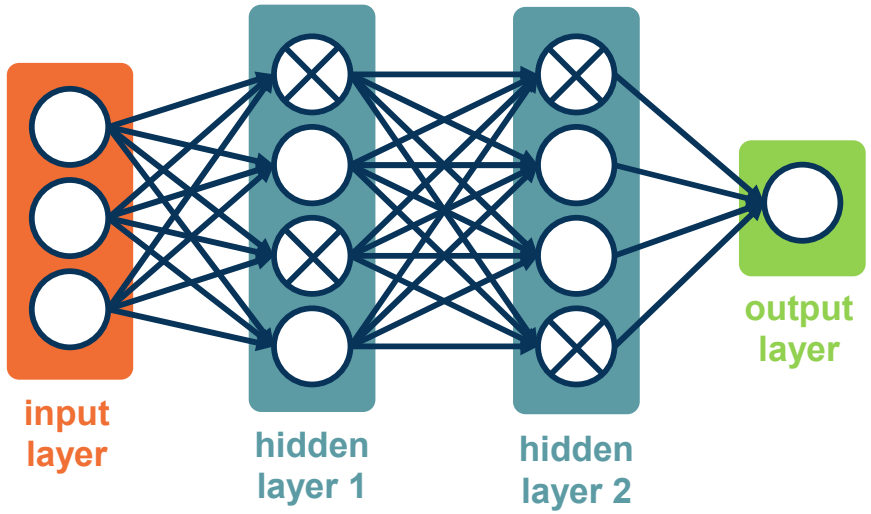
From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

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

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

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

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



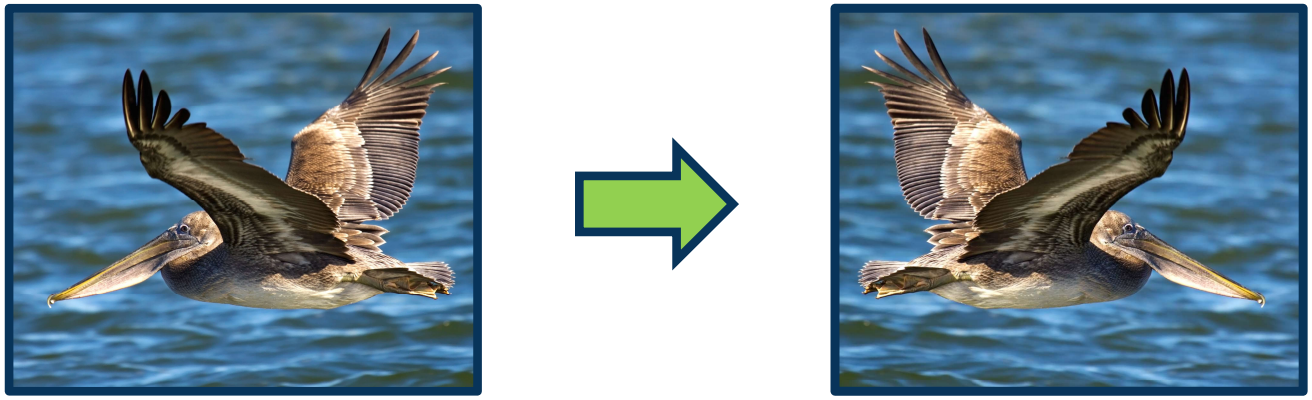
From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Data Augmentation

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

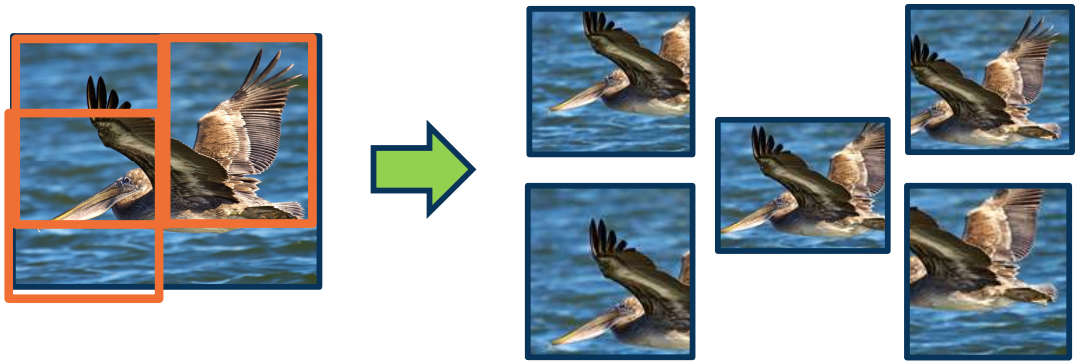
- This essentially **"increases"** your dataset
- Transformations should not change meaning of the data (or label has to be changed as well)

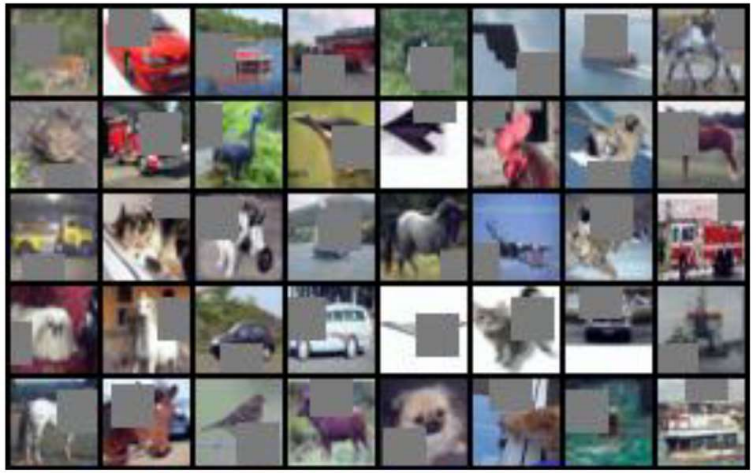
Simple example: Image Flipping



Random crop

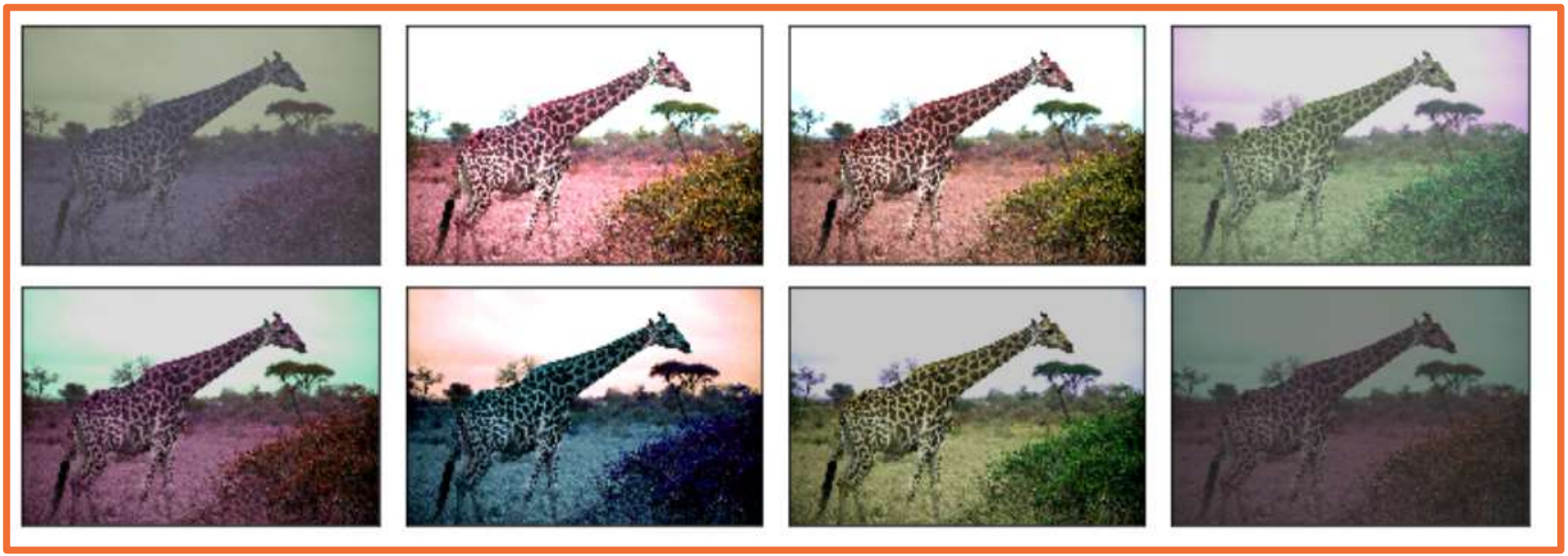
- Take different crops during training
- Can be used during inference too!





CutMix

Color Jitter

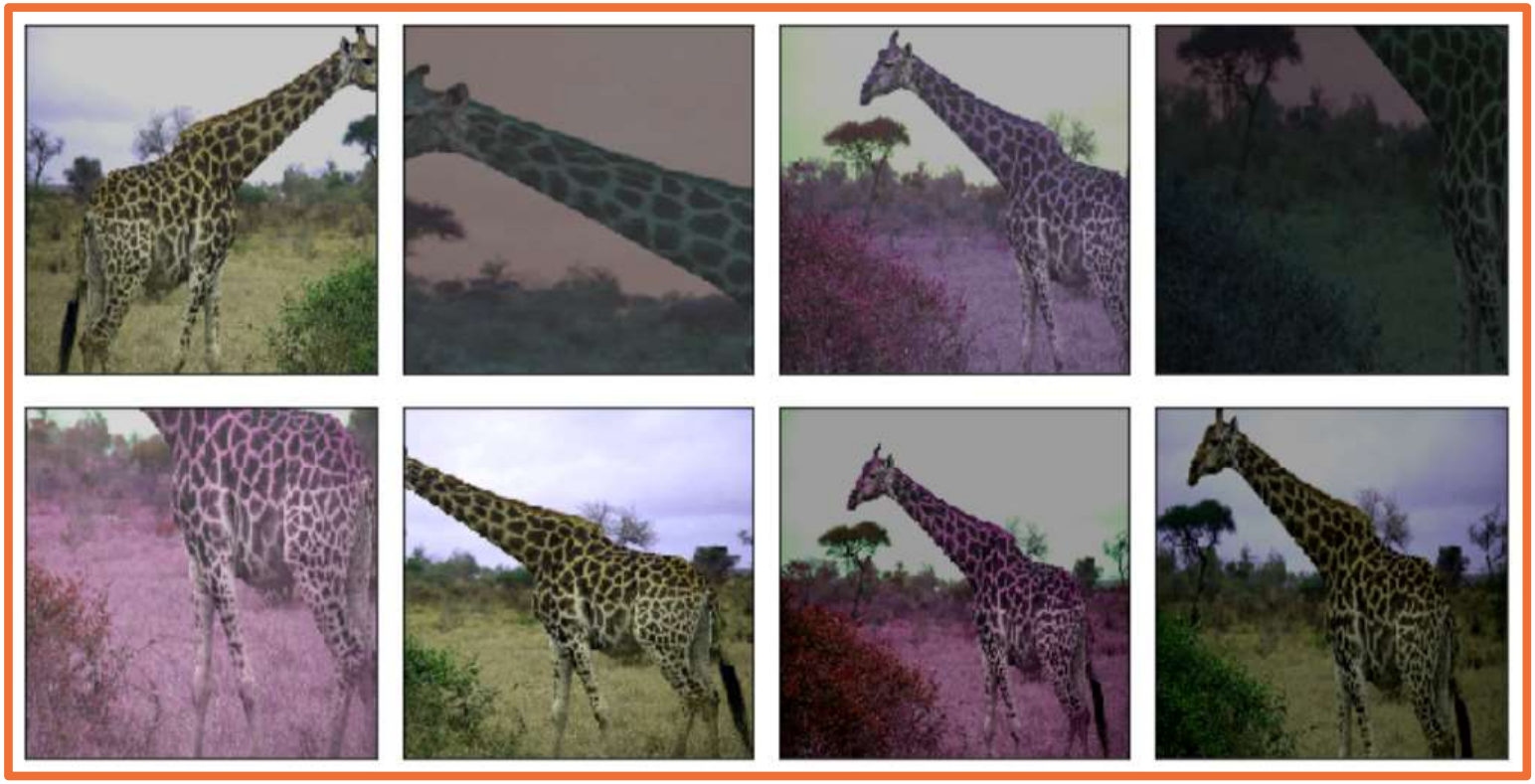


From https://mxnet.apache.org/versions/1.5.0/tutorials/gluon/data_augmentation.html

We can apply **generic affine transformations:**

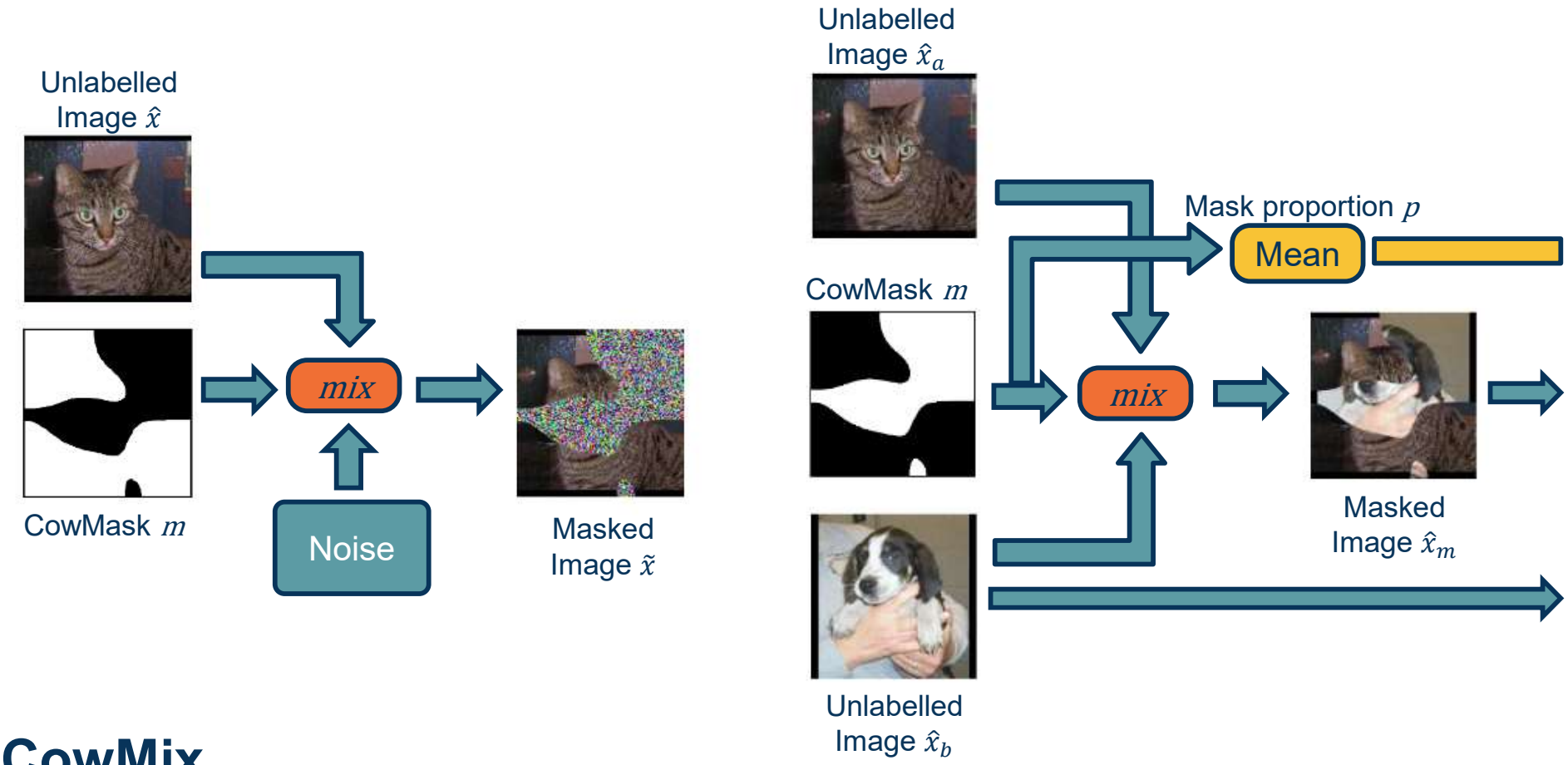
- **Translation**
- **Rotation**
- **Scale**
- **Shear**

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



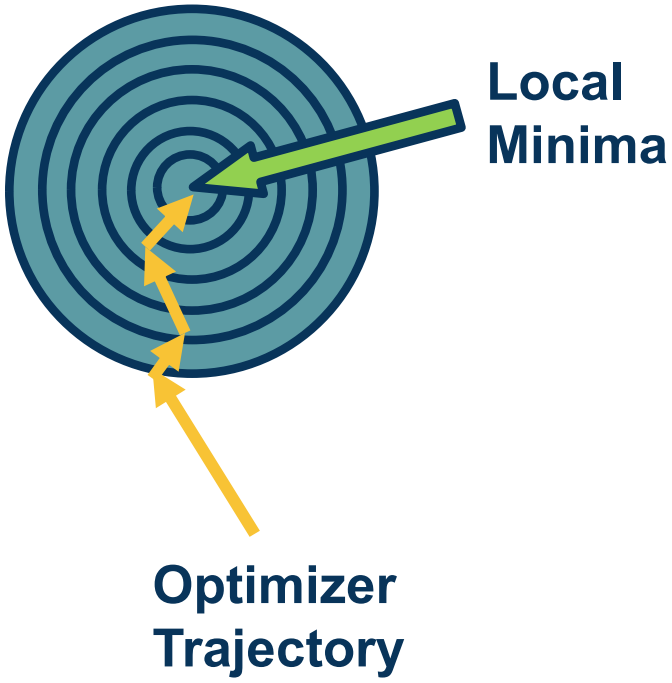
CowMix

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



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)

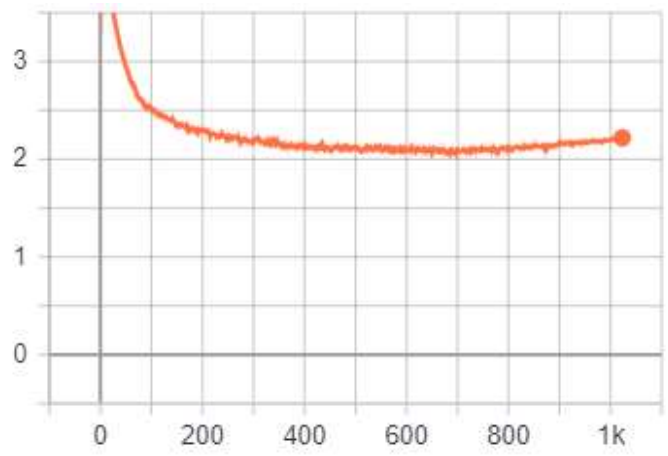
Use **cross-validation** to decide on hyper-parameters if amount of data is an issue

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. $-\mathbf{log}(\boldsymbol{p})$ for cross-entropy, where $\boldsymbol{p} = \mathbf{0.5}$

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

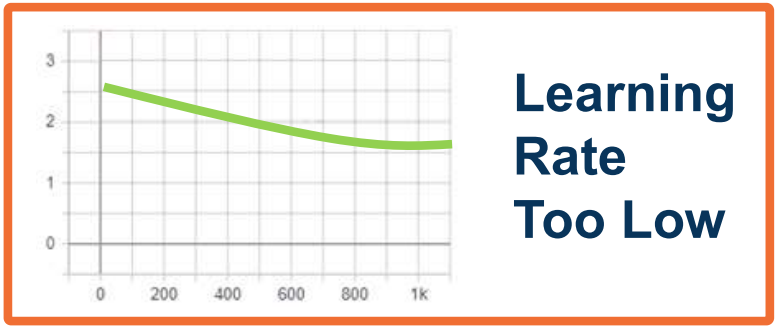


Validation Loss

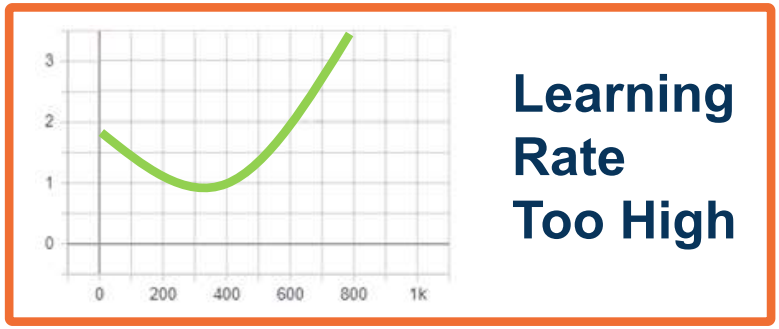
Loss and Not a Number (NaN)

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



Learning Rate Too Low



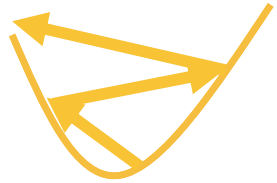
Learning Rate Too High

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

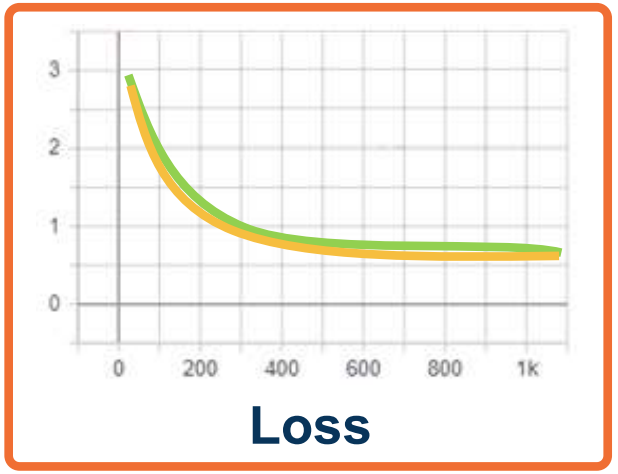
- Divide by zero
- Forgetting the log!

In pytorch, use autograd's detect anomaly to debug

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



- 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



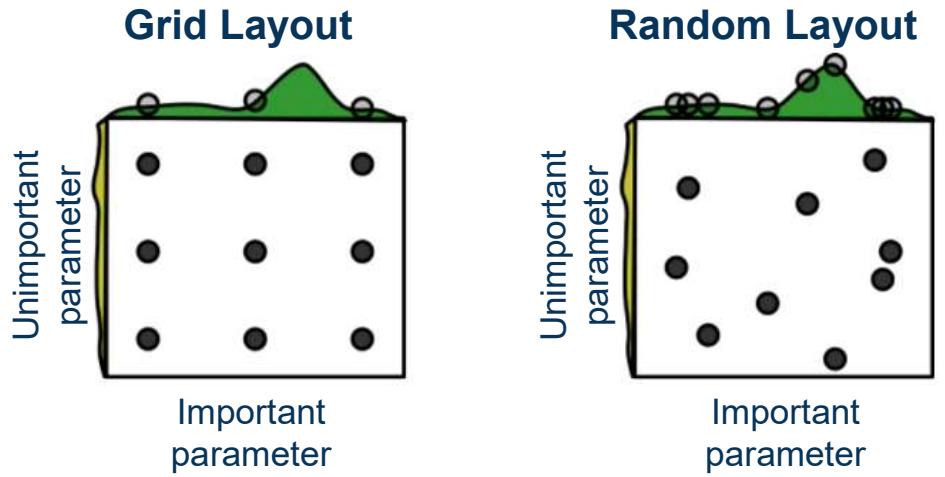
Overfitting

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

Inter-dependence of Hyperparameters

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

Examples:

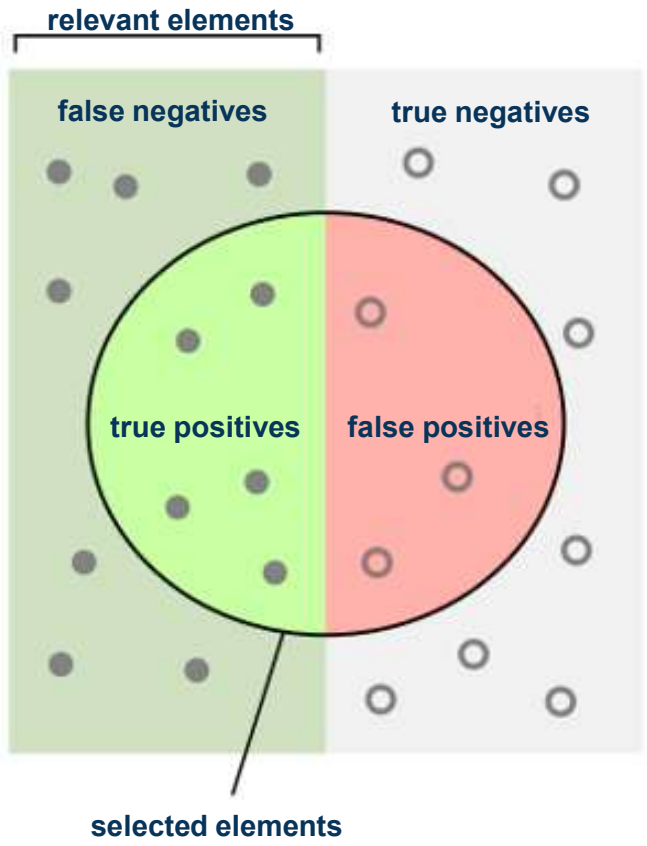
- Batch norm and dropout **maybe not be needed together** (and sometimes the combination is worse)
- The learning rate should be **changed proportionally to batch size** – increase the learning rate for larger batch sizes
  - **One interpretation:** Gradients are more reliable/smoother

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

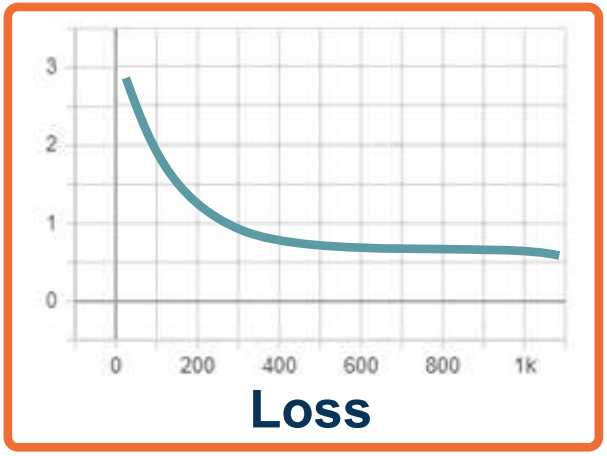
- **Example:** Cross entropy loss

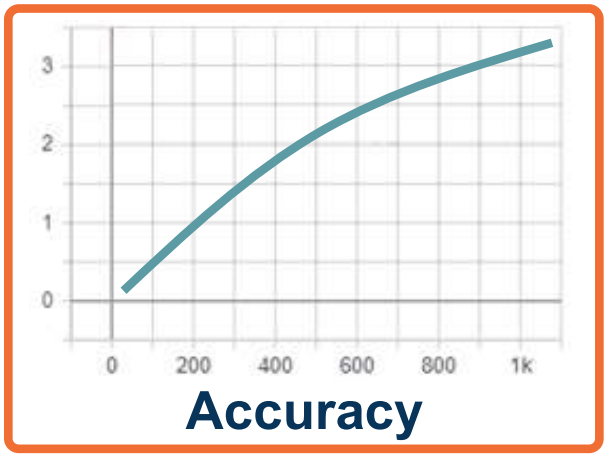
$$L = -log\, P(Y = y_i | X = x_i)$$

- **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**!



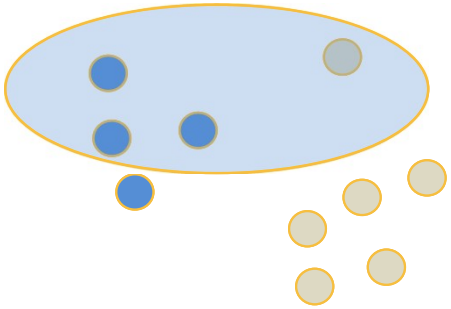


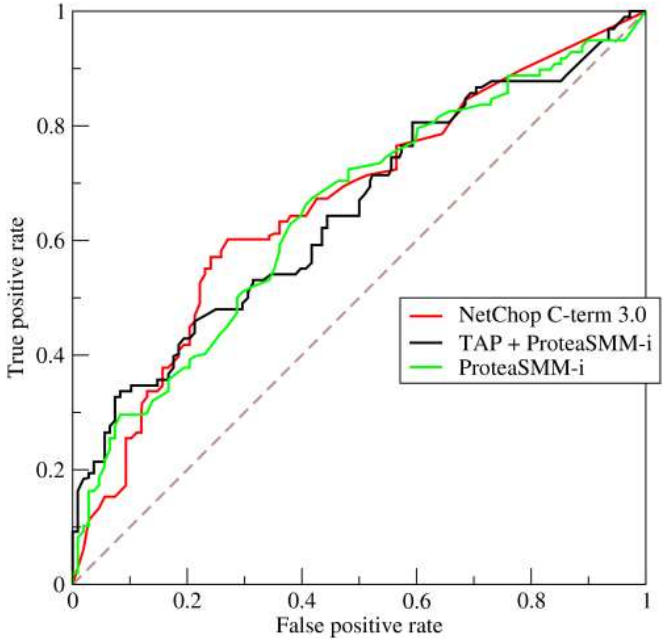
Simple Example: Cross-Entropy and Accuracy

Georgia Tech

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: $\boldsymbol{FPR} = \frac{\boldsymbol{fp}}{\boldsymbol{fp+tn}}$
  - $\boldsymbol{Accuracy} = \frac{\boldsymbol{tp+tn}}{\boldsymbol{tp+tn+fp+fn}}$

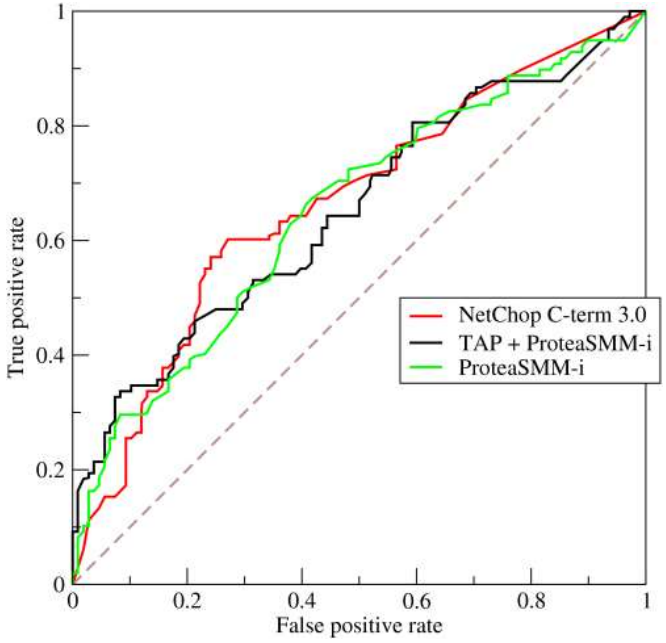




From https://en.wikipedia.org/wiki/Receiver_operating_characteristic

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: $\boldsymbol{FPR} = \frac{\boldsymbol{fp}}{\boldsymbol{fp+tn}}$
  - $\boldsymbol{Accuracy} = \frac{\boldsymbol{tp+tn}}{\boldsymbol{tp+tn+fp+fn}}$
- 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

Resource:

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