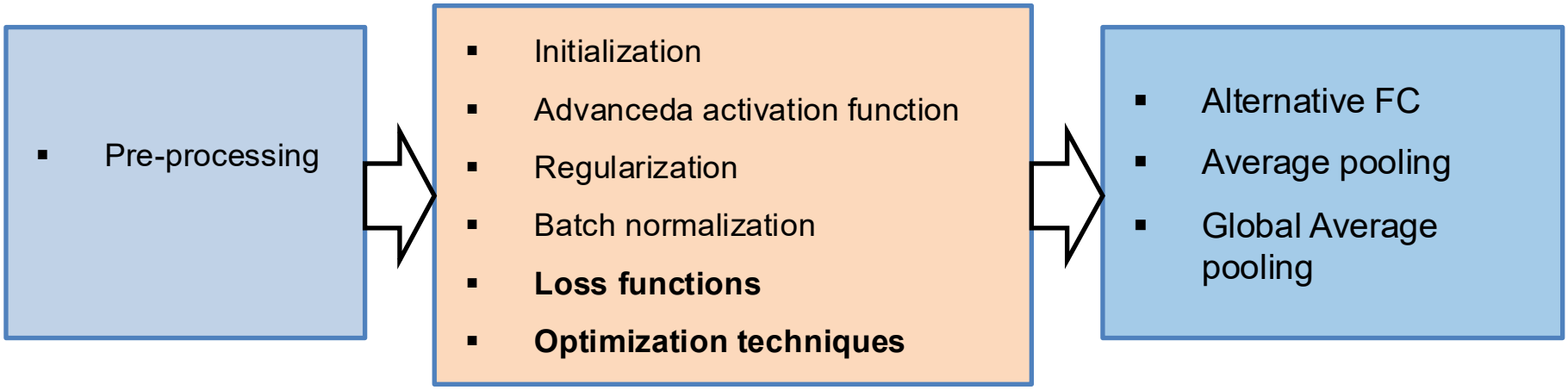
COMP/EECE 7/8740 Neural Networks

Topics: Advanced Training Techniques

- Loss functions
  - Zero-one loss (per class)
  - Binary and Categorical Cross Entropy Loss
  - Foca Loss
- Optimization techniques:
  - SGD, Adagrad, AdaDelta, RMSProp, Adam, and AdamW
  - Sophia and Muon losses

Md Zahangir Alom
Department of Computer Science
University of Memphis, TN

Overview : Learning with DNN



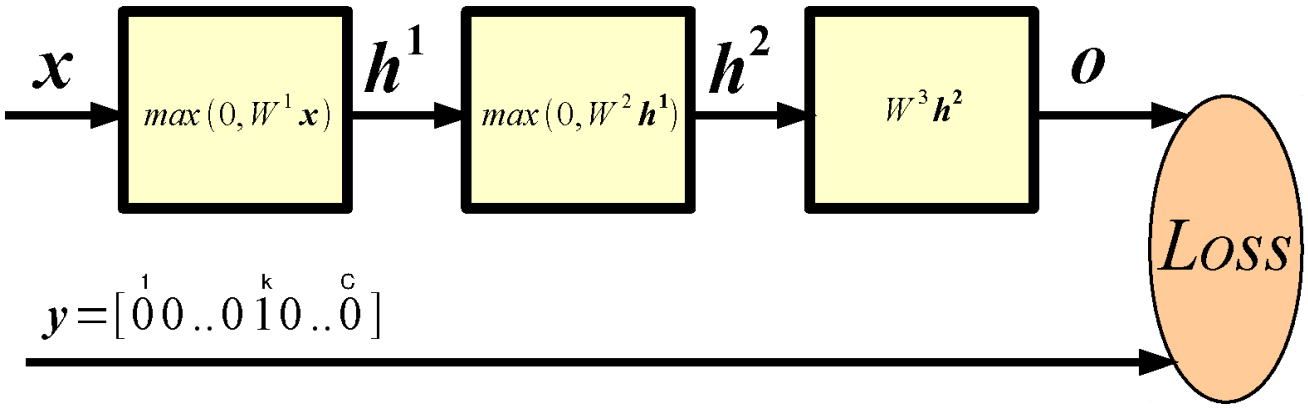
Inputs

Related to the DNN models implementation and ensuring efficient training

Related to classification & output layers

Loss Functions

How Good is a Network?



- A **loss function** measures how far a model's predictions are from the true targets
- Training aims to **minimize this loss**, guiding parameter updates so the model makes more accurate predictions.

What is an appropriate loss?

- Compare training class to output class
- Zero-one loss (per class)

$$L(\hat{y}, y) = I(\hat{y} \neq y),$$

- **Is it good?**
  - Nope – **it's a step function**.
  - We need to compute the gradient of the loss.
  - This loss is **not differentiable**, and 'flips' easily.

Entropy Loss function

$$P(y \mid x) = \begin{cases} \hat{p} & \text{if } y = 1 \\ 1 - \hat{p} & \text{if } y = 0 \end{cases}$$

Max/Min occours at points where derivative is equal to 0:

$$\frac{\partial P(y \mid x)}{\partial p} = 0$$

Our examples can only be part of one class at a time either 1 or 0 so,

- when $y = 1$ then $\hat{p}$ should be the output and $1 - \hat{p}$ should be ignored,
- similarly, when $y = 0$ then $1 - \hat{p}$ should be the output and $\hat{p}$ should be ignored.

We can combine this as follows:

$$P(y \mid x) = \hat{p}^y (1 - \hat{p})^{1-y}$$

if $y = 1$ then:

$$P(y \mid x) = \hat{p}^y (1 - \hat{p})^{1-y} = \hat{p}^1 (1 - \hat{p})^{1-1} = \hat{\mathbf{p}}$$

if $y = 0$ then:

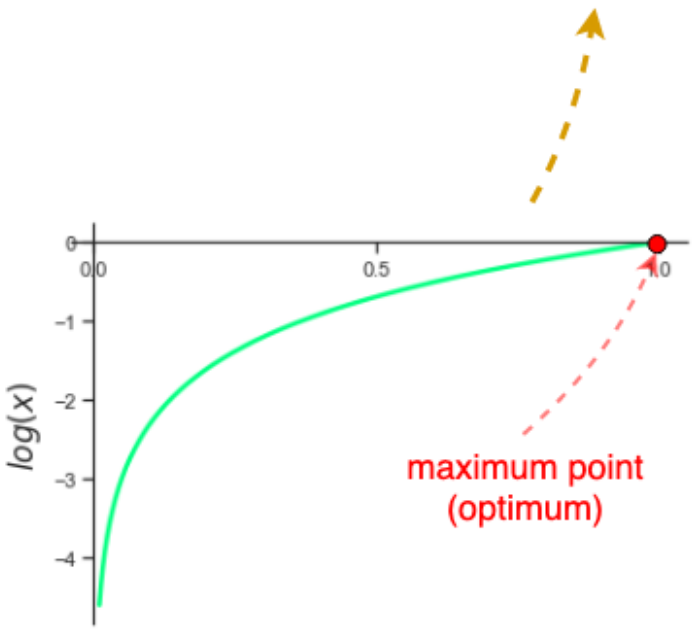
$$P(y \mid x) = \hat{p}^y (1 - \hat{p})^{1-y} = \hat{p}^0 (1 - \hat{p})^{1-0} = \mathbf{1} - \hat{\mathbf{p}}$$

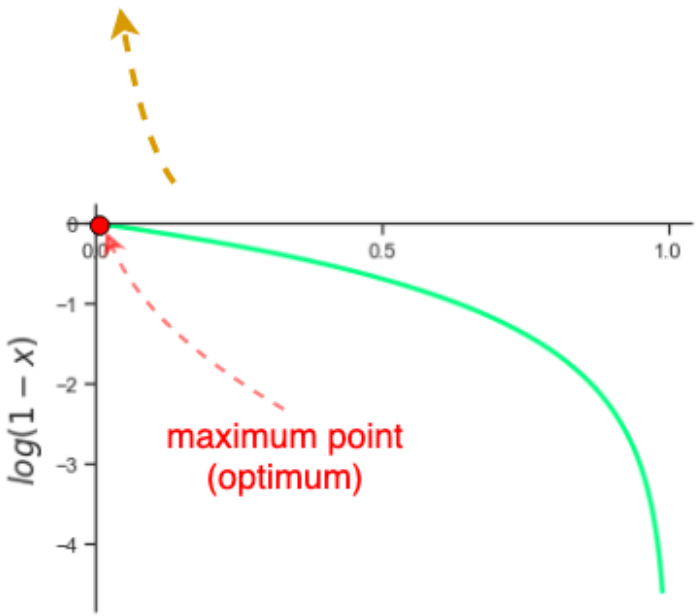
Entropy Loss function

Recall the properties of natural log:

- $\log(a * b) = log(a) + \log(b)$
- $log(a^x = x \log(a))$

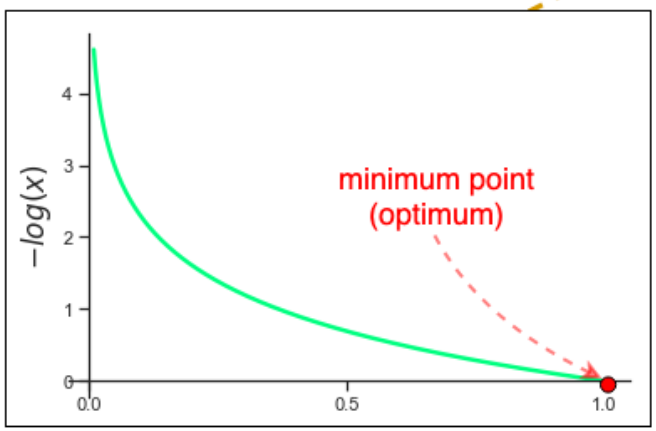
$$\log(P(y \mid x) = \log(\hat{p}^y(1-\hat{p})^{(1-y)})$$
$$= \log(\hat{p}^y) + \log((1-\hat{p})^{(1-y)})$$
$$= \mathbf{y}\log(\hat{\mathbf{p}}) + (\mathbf{1}-\mathbf{y})\log(\mathbf{1}-\hat{\mathbf{p}})$$

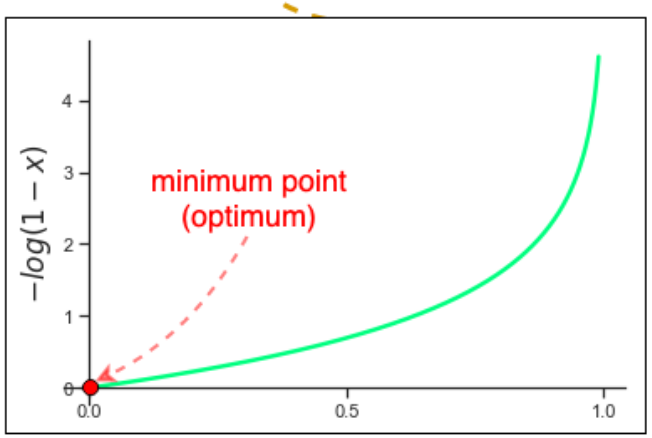




Entropy Loss function

$$\begin{aligned} \textit{Binary Cross Entropy Loss}(y, \hat{p}) &= -\log(P(y \mid x) \\ &= -(y \log(\hat{p}) + (1 - y) \log(1 - \hat{p})) \\ &= \mathbf{-y \log(\hat{p}) - (1 - y) \log(1 - \hat{p})} \end{aligned}$$





Binary Cross Entropy Loss/Cost function

$$\text{Binary Cross-Entropy Cost} = \frac{1}{m}\left(-\log\left(\mathcal{L}(\hat{p}^{(1)}, \hat{p}^{(2)}, \cdots, \hat{p}^{(m)})\right)\right)$$

$$= \frac{\mathbf{1}}{\mathbf{m}} \sum_{\mathbf{i=1}}^{\mathbf{m}} -\mathbf{y^{(i)}} \log(\mathbf{\hat{p}^{(i)}}) - (\mathbf{1} - \mathbf{y^{(i)}}) \log(\mathbf{1} - \mathbf{\hat{p}^{(i)}})$$

$$-\log\left(\mathcal{L}(\hat{p}^{(1)}, \hat{p}^{(2)}, \cdots, \hat{p}^{(m)})\right) = -\sum_{i=1}^{m} y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})$$

$$= \sum_{i=1}^{m} -y^{(i)} \log(\hat{p}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})$$

Categorical Cross Entropy Loss/Cost function

$$\mathcal{L}(\hat{p}^{(1)}, \hat{p}^{(2)}, \cdots, \hat{p}^{(m)} \mid (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})) = \prod_{i=1}^{m} \hat{p}^{(i)\,y^{(i)}} (1 - \hat{p}^{(i)})^{1-y^{(i)}}$$

This is the ***likelihood($\mathcal{L}$)*** function, where we are trying to maximize the *each* probability, $\hat{p}$, given the examples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})$.

$\prod_{i=1}^{m}$ represents product over examples 1 to m

$$\begin{aligned}
\log\left(\mathcal{L}(\hat{p}^{(1)}, \hat{p}^{(2)}, \cdots, \hat{p}^{(m)} \mid (x^{(1)}, y^{(1)}, (x^{(2)}, y^{(2)}, \cdots, (x^{(m)}, y^{(m)}))\right) &= \log\left(\prod_{i=1}^{m} \hat{p}^{(i)\,y^{(i)}} (1 - \hat{p}^{(i)})^{1-y^{(i)}}\right) \\
&= \sum_{i=1}^{m} \log\left(\hat{p}^{(i)\,y^{(i)}} (1 - \hat{p}^{(i)})^{1-y^{(i)}}\right) \\
&= \sum_{i=1}^{m} y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})
\end{aligned}$$

This is called the ***log-likelihood*** function

Why cross-entropy **loss function**?

Entropy :

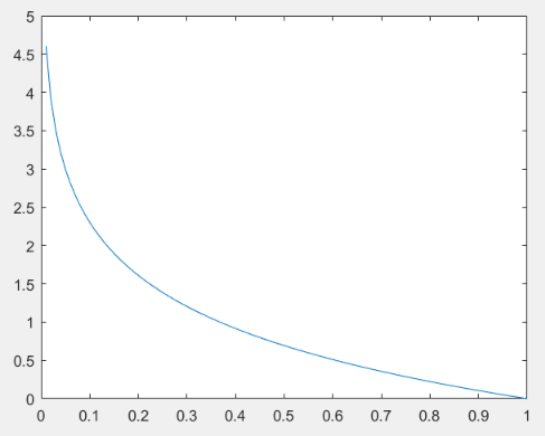
$$H(X) = \begin{cases} -\int_x p(x) \log p(x), & \text{if } X \text{ is continous} \\ -\sum_x p(x) \log p(x), & \text{if } X \text{ is discrete} \end{cases}$$

Cross-entropy is defined as

$$L_{\text{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

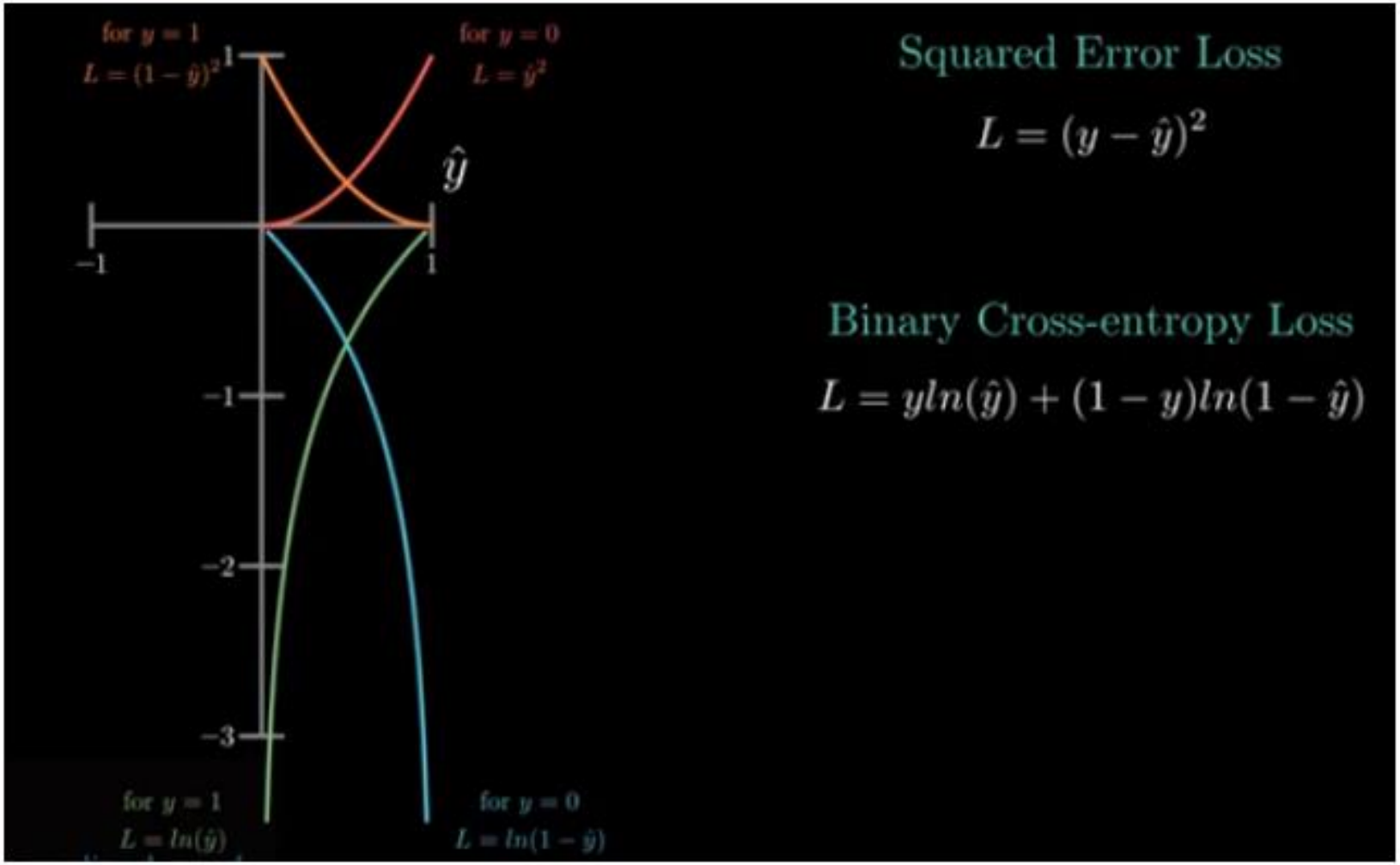
where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

- **Is it a good loss?**
  - Differentiable
  - Cost decreases as probability increases



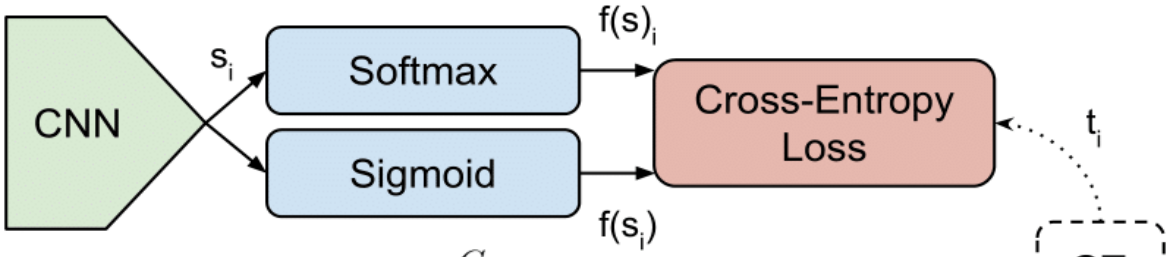
Why cross-entropy **loss function**?

Gradient Scope in Loss functions



Types of Cross-Entropy Loss

- Categorical Cross-Entropy Loss
- Binary Cross-Entropy Loss and

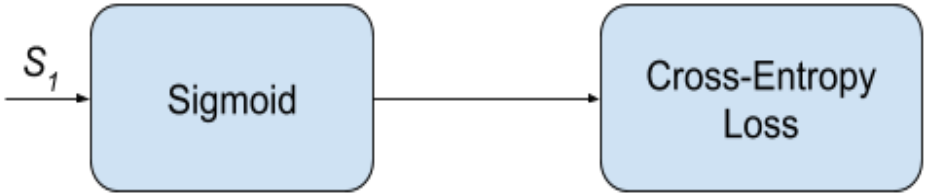


$$CE = -\sum_{i}^{C} t_i log(f(s)_i)$$

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

Binary Cross-Entropy Loss

- Also called **Sigmoid Cross-Entropy loss**. It is a **Sigmoid activation** plus a **Cross-Entropy loss**.
- It's called **Binary Cross-Entropy Loss** because it sets up a binary classification problem between C′=2 classes for every class in C.



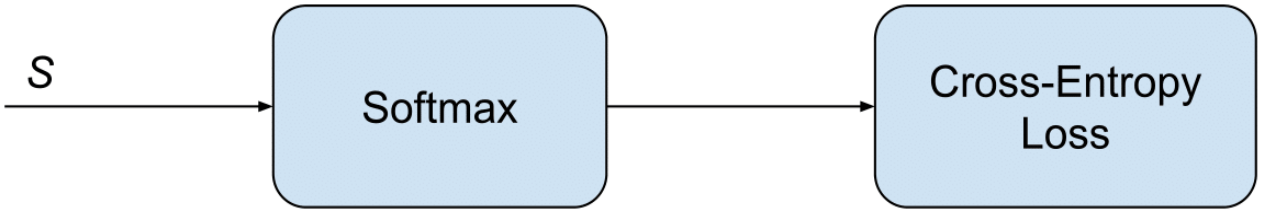
$$CE = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

$$L = -\sum_{i=1}^{2} t_i \log(p_i)$$
$$= -[t_1 \log(p_1) + t_2 \log(p_2)]$$
$$= -[t \log(p) + (1 - t) \log(1 - p)]$$

$$P = f(s_i) = \frac{1}{1 + e^{-s_i}}$$

Categorical Cross-Entropy Loss

- Also called **Softmax Loss**. It is a **Softmax activation** plus a **Cross-Entropy loss**
- We use this loss when we train a DNN or CNN to output a probability over the C classes for each image (multi-class classification).

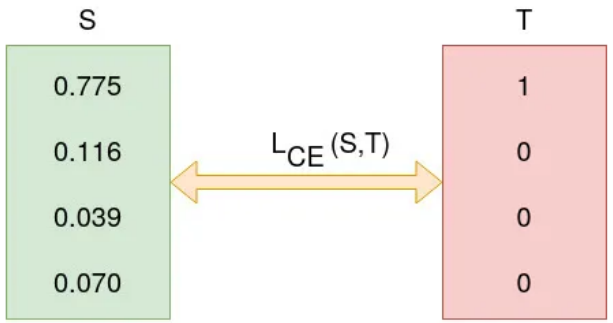


$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

Examples

Categorical Cross-Entropy Loss

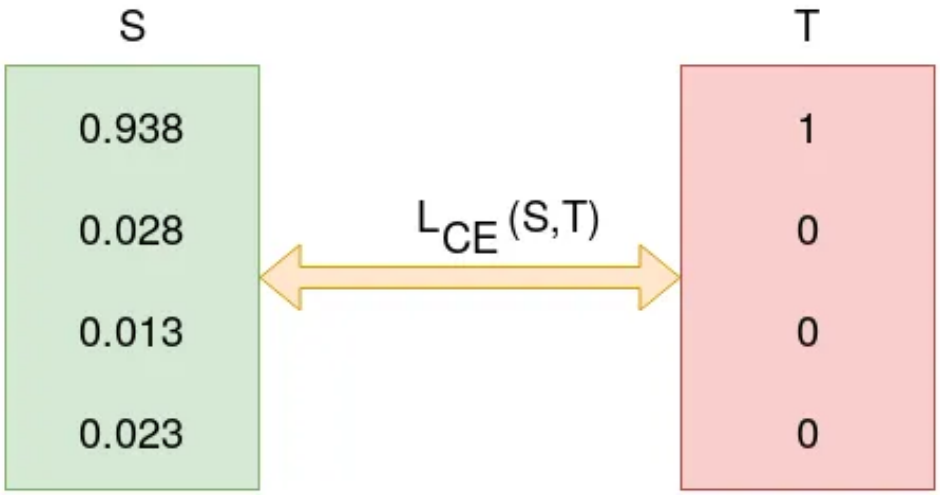
$$L = -\frac{1}{N}\left[\sum_{j=1}^{N}\left[t_j \log(p_j) + (1 - t_j)\log(1 - p_j)\right]\right]$$



$$\begin{aligned} L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\ &= -\left[1\log_2(0.775) + 0\log_2(0.126) + 0\log_2(0.039) + 0\log_2(0.070)\right] \\ &= -\log_2(0.775) \\ &= 0.3677 \end{aligned}$$

Examples:

Categorical Cross-Entropy Loss

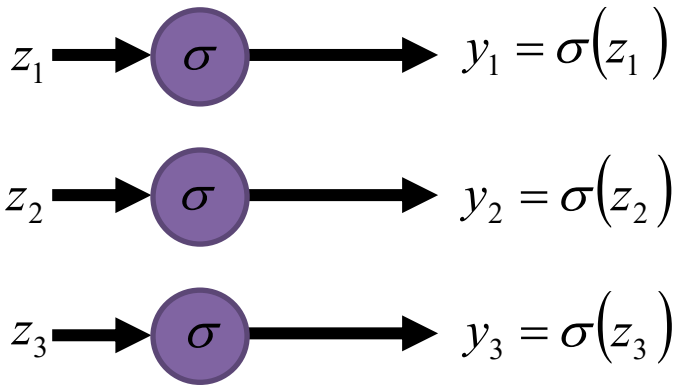


$$L_{CE} = -1 \log_2(0.936) + 0 + 0 + 0$$
$$= 0.095$$

Softmax Classifier (Multinomial Logistic Regression)

- Softmax layer as output layer

Ordinary Layer



- In general, the output of network can be any value.
- May not be easy to interpret

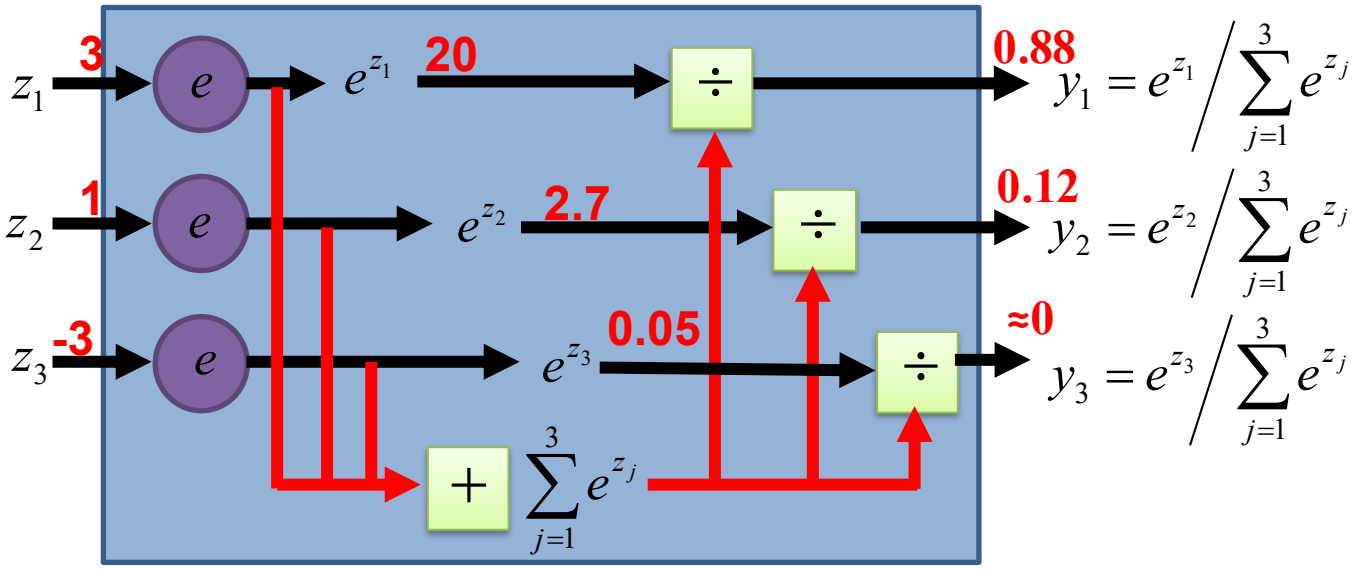
Softmax Classifier (Multinomial Logistic Regression)

- Softmax layer as the output layer

Probability:

- $1 > y_i > 0$
- $\sum_i y_i = 1$

Softmax Layer



Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

cat	**3.2**
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

cat	**3.2**
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Probabilities must be >= 0

cat	**3.2**	exp →	**24.5**
car	5.1		164.0
frog	-1.7		0.18

unnormalized probabilities

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

		exp		normalize	
cat	**3.2**	→	**24.5**	→	**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7	→	0.18	→	0.00

unnormalized probabilities

probabilities

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

	Unnormalized log-probabilities / logits	exp	unnormalized probabilities	normalize	probabilities
cat	**3.2**	→	**24.5**	→	**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7	→	0.18	→	0.00

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

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

Probabilities must be >= 0

Probabilities must sum to 1

	Unnormalized log-probabilities / logits	exp	unnormalized probabilities	normalize	probabilities
cat	**3.2**		**24.5**		**0.13**
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

→ L_i = -log(0.13) = **0.89**

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

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

Probabilities must be >= 0

Probabilities must sum to 1

	Unnormalized log-probabilities / logits	exp	unnormalized probabilities	normalize	probabilities
cat	**3.2**		**24.5**		**0.13**
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

→ L_i = -log(0.87) = **0.0604**

Maximum Likelihood Estimation :
Choose probabilities to maximize the likelihood of the observed data

Softmax Classifier (Multinomial Logistic Regression)

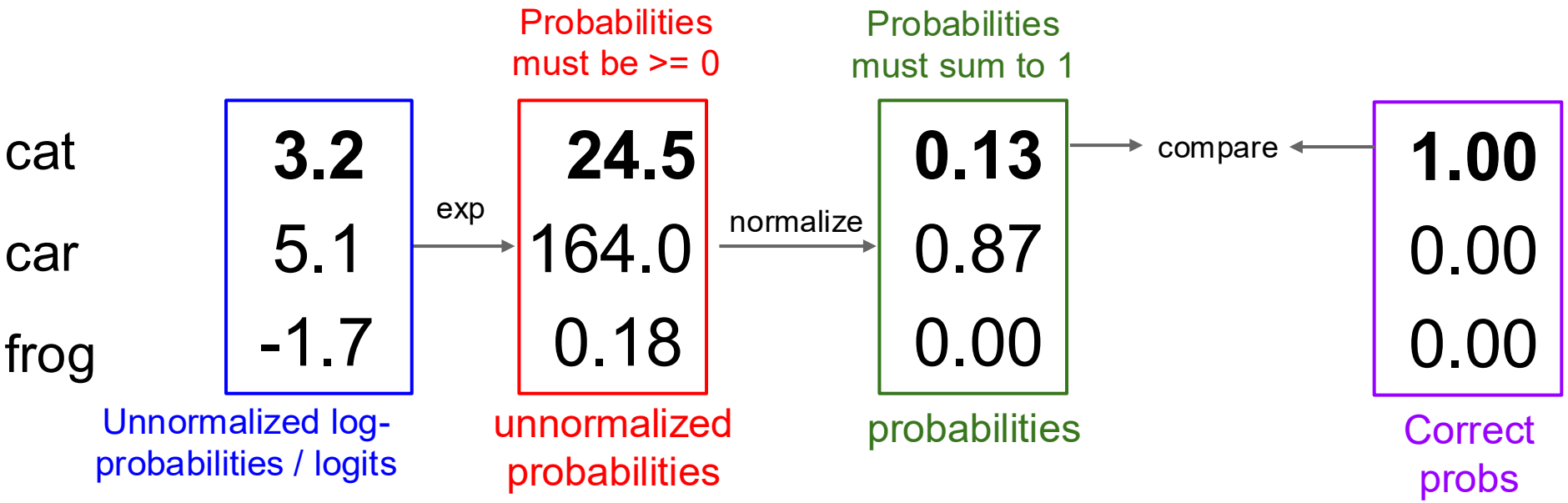
Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

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



Softmax Classifier (Multinomial Logistic Regression)

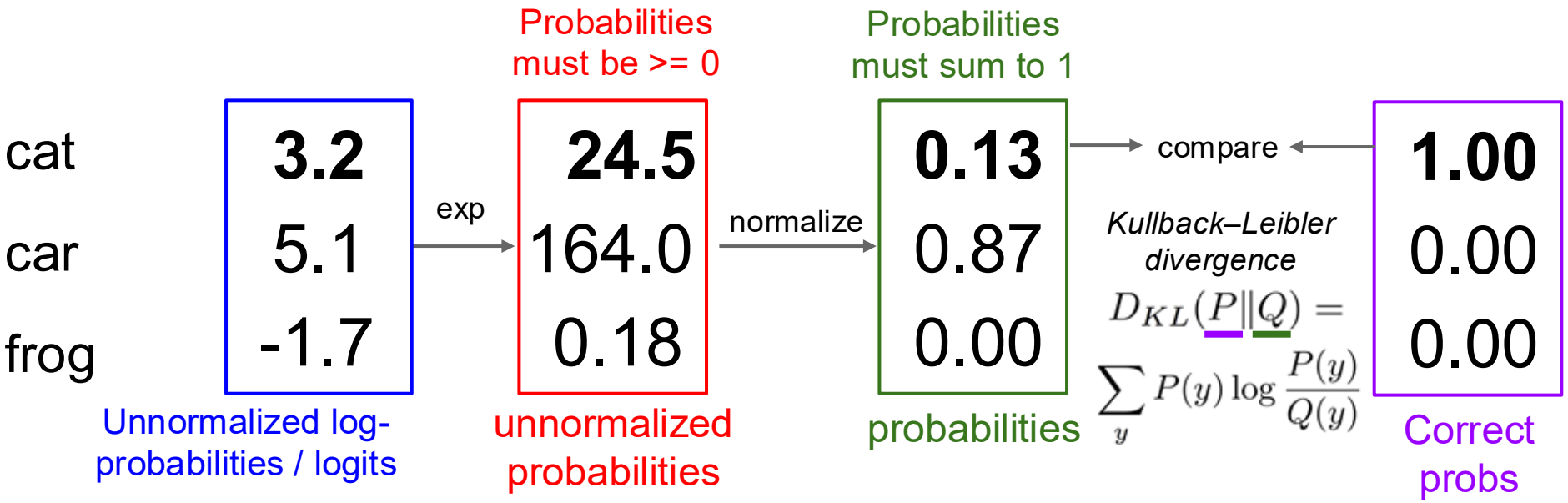
Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

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



Softmax Classifier (Multinomial Logistic Regression)

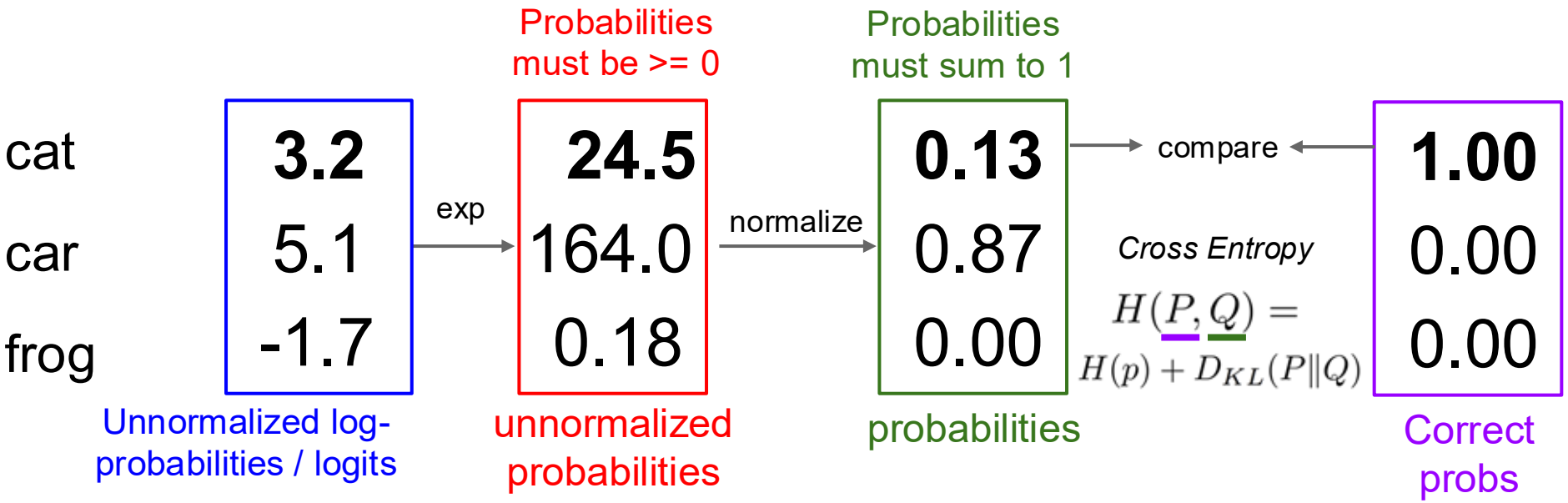
Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

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



Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Maximize probability of correct class

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

Putting it all together:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

cat	**3.2**
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Maximize probability of correct class

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

Putting it all together:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

cat	**3.2**
car	5.1
frog	-1.7

Q: What is the min/max possible loss L_i?

Softmax Classifier (Multinomial Logistic Regression)

cat **3.2**

car 5.1

frog -1.7

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Maximize probability of correct class

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

Putting it all together:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

Q: What is the min/max possible loss L_i?
A: min 0, max infinity

Softmax Classifier (Multinomial Logistic Regression)

cat **3.2**

car 5.1

frog -1.7

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Maximize probability of correct class

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

Putting it all together:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

Q2: At initialization all s will be approximately equal; what is the loss?

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Maximize probability of correct class

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

Putting it all together:

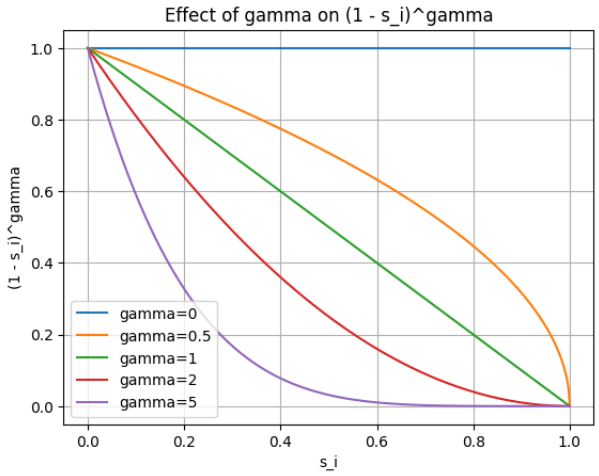
$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

cat	**3.2**
car	5.1
frog	-1.7

Q2: At initialization all s will be approximately equal; what is the loss?
A: log(C), eg log(10) ≈ 2.3

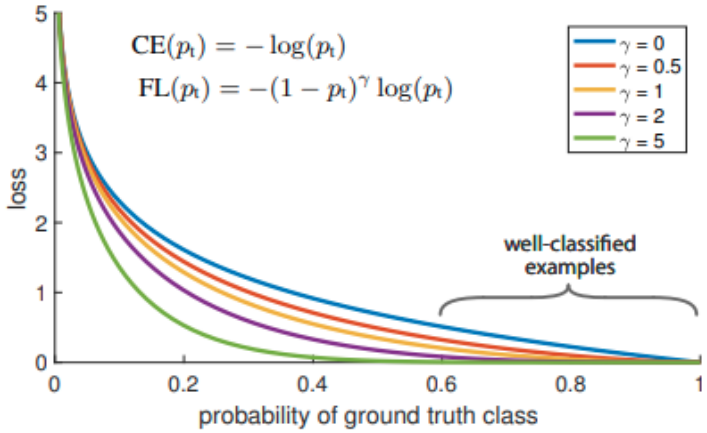
Focal Loss

- **Focal Loss** was introduced by Lin et al., from Facebook. They claim to improve one-stage object detectors using **Focal Loss** to train a detector named “RetinaNet”.
- **Focal loss** is a **Cross-Entropy Loss** that weights the contribution of each sample to the loss based in the classification error.
- **Focal loss** could also be considered a **Binary Cross-Entropy Loss** (Sigmoid activations + Cross-Entropy Loss)



$$FL = -\sum_{i=1}^{C=2}(1-s_i)^{\gamma} t_i log(s_i) \qquad \gamma >= 0$$

γ=0, Focal Loss is equivalent to Binary Cross Entropy Loss.



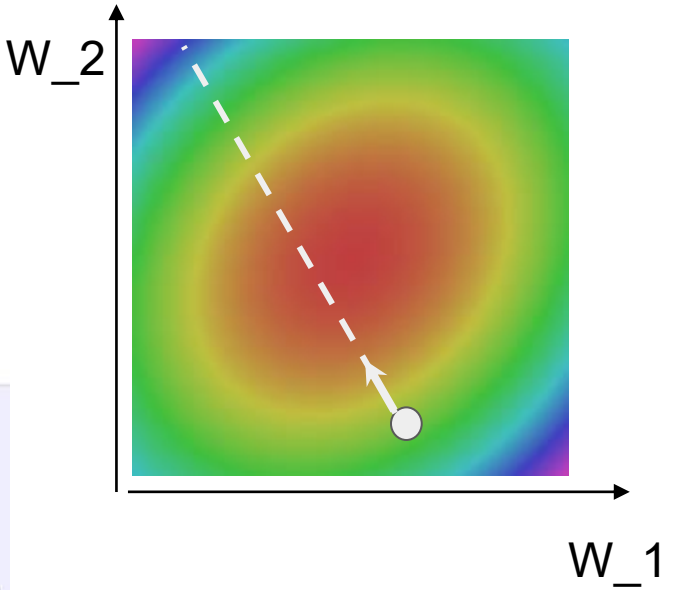
Lin, Tsung-Yi, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. "Focal loss for dense object detection." In *Proceedings of the IEEE international conference on computer vision*, pp. 2980-2988. 2017.

DNN optimization methods

DNN optimization methods are algorithms (like SGD, Adam, or AdamW) that **iteratively update a neural network's parameters to minimize the loss** function during training.

- **Stochastic Gradient Descent (SGD)**
- **SGD+Momentum**
- Adagrad
- Nesterov's Accelerated Gradient (NAG)
- RMSProp
- Adam
- AdamW and
- Many more …

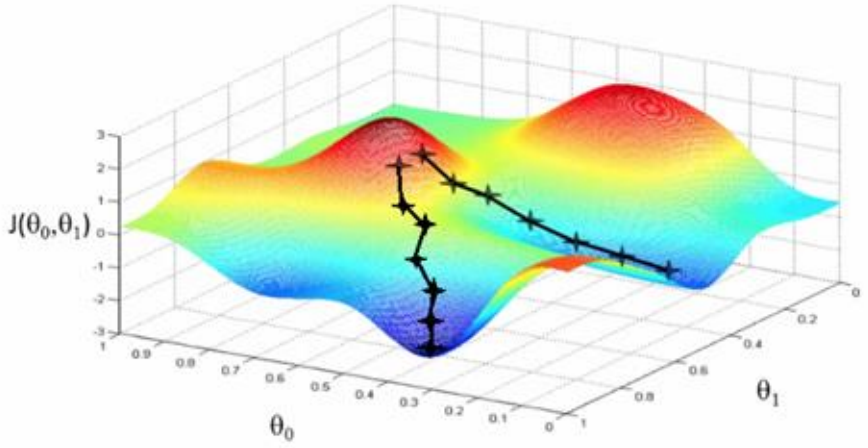
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Stochastic Gradient Descent

- Goal:
  - Optimize parameters to minimize loss
- Step along the direction of steepest descent (negative gradient)

$$\theta_t = \theta_{t-1} - \alpha \nabla_{\theta_{t-1}} f(\theta_{t-1})$$

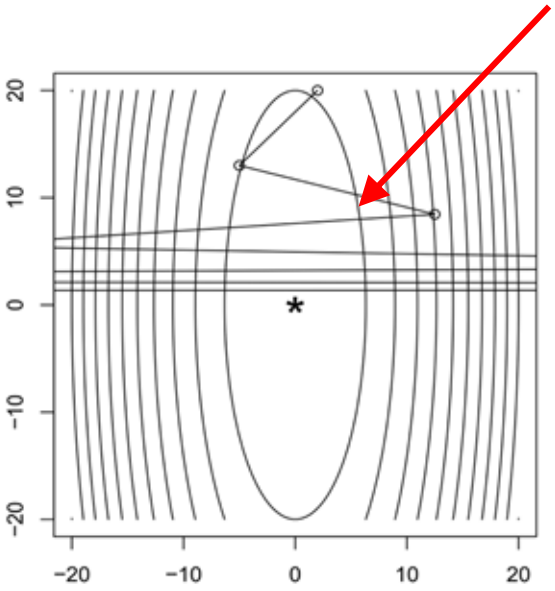


Picture from Andrew Ng's Machine Learning Course

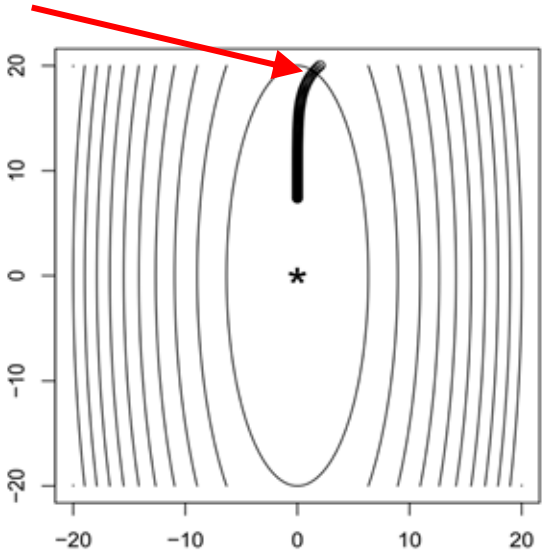
What does SGD do?

- Gradient over **entire dataset is impractical**
- Better to **take quick, noisy steps**
- Go fast in **along the consistent direction**

- Estimate **gradient over a mini-batch of samples**
  - Increasing batch **size on GPU is essentially free up to a point**



Too big learning rate



Too small learning rate

https://www.cs.cmu.edu/~ggordon/10725-F12/slides/05-gd-revisited.pdf

SGD with Momentum

$$\theta_t = \theta_{t-1} - \alpha \nabla_{\theta_{t-1}} f(\theta_{t-1})$$

$$v_t = \mu v_{t-1} - \alpha \nabla f(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

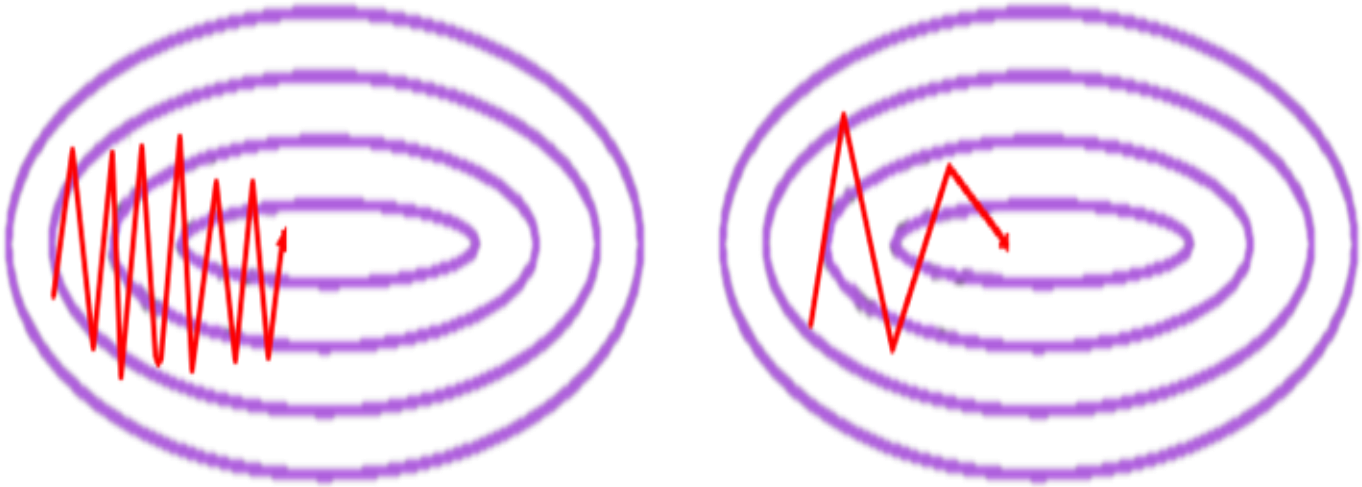
$$v_t = \mu v_{t-1} - \alpha \nabla f(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

$$\theta_t = \theta_{t-1} + (0 \cdot v_{t-1} - \alpha \nabla f(\theta_{t-1}))$$

Same as vanilla gradient descent

Impact of SGD with momentum



SGD without Momentum

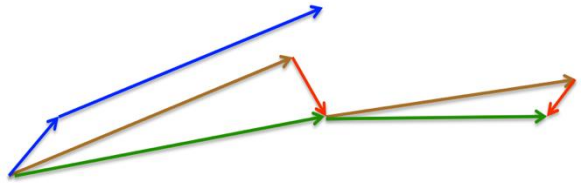
SGD with Momentum

- **Cancels out oscillation**
- **Gathers speed** in direction that matters
- We can adapt our **updates to the slope of our error function and speed up SGD in turn**

Nesterov's Accelerated Gradient (NAG)

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

γ is usually set to 0.9 or a similar value



Nesterov update (Source: G. Hinton's lecture 6c)

- First computes the **current gradient (small blue vector in figure)** and then takes **a big jump in the direction of the updated accumulated gradient (big blue vector)**
- NAG first makes a big jump in the **direction of the previous accumulated** gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the **complete NAG update (green vector).**

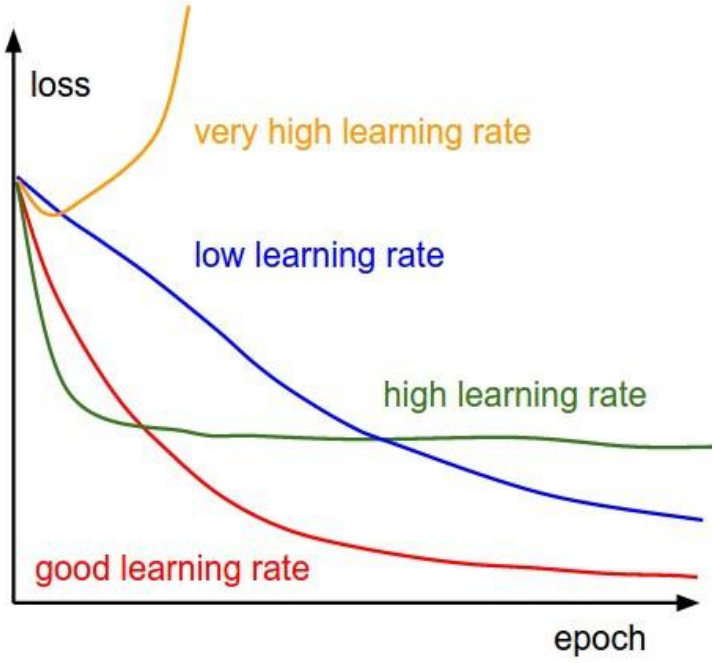
- Advantages:
  - Prevents **from going too fast and results in increased** responsiveness
  - Significantly **increased the performance of RNNs**

Challenges : per parameter learning rate

- Gradients of **different layers have different magnitudes**
- Different neuron units **have different firing rates**

- So, what we want :
  - We would also like to adapt our updates to each individual parameter to **perform larger or smaller updates depending on their importance** (different learning rates for different parameters)

How to pick the learning rate?



- **Too big = diverge**, too small = slow convergence
- No **"one learning rate to rule them all"**
- Start from a high value and **keep cutting by half if model diverges**
- Learning rate schedule:
  - decay learning rate over time

http://cs231n.github.io/assets/nn3/learningrates.jpeg

Adagrad

- **Adagrad** is a adapts the learning rate to the parameters, performing
  - Smaller updates(low learning rates) for parameters associated with frequently occurring features, and
  - Larger updates (high learning rates) for parameters associated with infrequent features.

- But how?
  - Gradient update depends on **history of magnitude of gradients**
  - Parameters with **small or sparse updates have larger learning rates**
  - **Square root** important for good performance
  - More **tolerance for learning rate**

Duchi et al 2011. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”

Adagrad

$$\theta_t = \theta_{t-1} - \alpha \nabla_{\theta_{t-1}} f(\theta_{t-1})$$

$$g_t = \sum_{\tau=1}^{t-1} (\nabla f(\theta_\tau))^2$$

$$= g_{t-1} + (\nabla f(\theta_{t-1}))^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{g_t + \epsilon}} \odot \nabla f(\theta_{t-1})$$

$$g_t = \sum_{\tau=1}^{t-1} (\nabla f(\theta_\tau))^2$$
$$= g_{t-1} + (\nabla f(\theta_{t-1}))^2$$

What happens as *t* increases?

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{g_t + \epsilon}} \odot \nabla f(\theta_{t-1})$$

Adagrad

- Advantages :
  - **Eliminates** the need to **manually tune the learning rate**.
  - Most implementations use **a default value of 0.01**
  - **Ensure faster and better training process**

- **Dis-advantages:**
  - Maintain **entire history of gradients**
  - As a results, the **sum of magnitude of gradients always increasing** during training process.
  - **Learning rate to shrink and eventually become infinitesimally small**, at which point the algorithm is **no longer able to acquire additional knowledge** (Forces learning rate to 0 over time)
  - **Hard to compensate** for in advance

Duchi et al 2011. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”

Solution: don't maintain all history

- **Monotonically increasing** because we hold all the history instead, **forget gradients far in the past**
- In practice, **downweight previous gradients exponentially**

- The following algorithms aim to resolve this flaw.
  - **AdaDelta, and**
  - **RMSProp**

Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701

Adadelta

- **Adadelta** is an extension of Adagrad that seeks to **reduce its aggressive, monotonically decreasing learning rate**
- Instead of accumulating all past squared gradients, Adadelta **restricts the window of accumulated past gradients to some fixed size w**
- The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) **only on the previous average and the current gradient**

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2.$$

γ to a similar value as the momentum term, around 0.9.

Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701

Adadelta

- What and where to replace:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t. \qquad \Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

- As the denominator is just the **root mean squared (RMS)** error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701

RMSProp

- RMSprop is an unpublished, **adaptive learning rate method proposed by Geoff Hinton**
- RMSprop and Adadelta have both been developed independently around the same time stemming from the need to **resolve Adagrad's radically diminishing learning rates.**
- RMSprop **only cares about recent gradients** (otherwise like Adagrad)
  - Good property because **optimization landscape changes**
- Standard **gamma (γ) is 0.9**

Hinton et al. 2012, http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

$$g_t = \sum_{\tau=1}^{t-1} (\nabla f(\theta_\tau))^2$$
$$= g_{t-1} + (\nabla f(\theta_{t-1}))^2$$

$$g_t = (1 - \gamma) \sum_{\tau=1}^{t-1} \gamma^{t-1-\tau} (\nabla f(\theta_\tau))^2$$
$$= \gamma g_{t-1} + (1 - \gamma) \nabla (f(\theta_{t-1}))^2$$

$$g_t = (1-\gamma)\sum_{\tau=1}^{t-1}\gamma^{t-1-\tau}\left(\nabla f(\theta_\tau)\right)^2$$

$$= \gamma g_{t-1} + (1-\gamma)\nabla\left(f(\theta_{t-1})\right)^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{g_t + \epsilon}} \odot \nabla f(\theta_{t-1})$$

- RMSprop also divides the learning rate by an exponentially decaying average of squared gradients.
- **Hinton suggests γ to be set to 0.9**, while a good default value for the **learning rate η is 0.001**.

Adam optimizer

- Adam is derived from **Adaptive Moment Estimation (Adam)** for each parameter
- Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum
- Essentially, **combine RMSProp + Momentum**
- Includes bias correction term from **initializing m and v to 0**

Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

$$\mu v_{t-1} - \alpha \nabla f(\theta_{t-1})$$ Momentum

$$\gamma g_{t-1} + (1-\gamma) \nabla (f(\theta_{t-1}))^2$$ RMSProp

---

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) \nabla f(\theta_{t-1})$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) (\nabla f(\theta_{t-1}))^2$$

Ref. : The relationship between second moment $E[X^2]$ and the variance $Var(X)$:

$Var(X)=E[X^2]-(E[X])^2$

$E[X^2] = Var(X) + (E[X])^2$

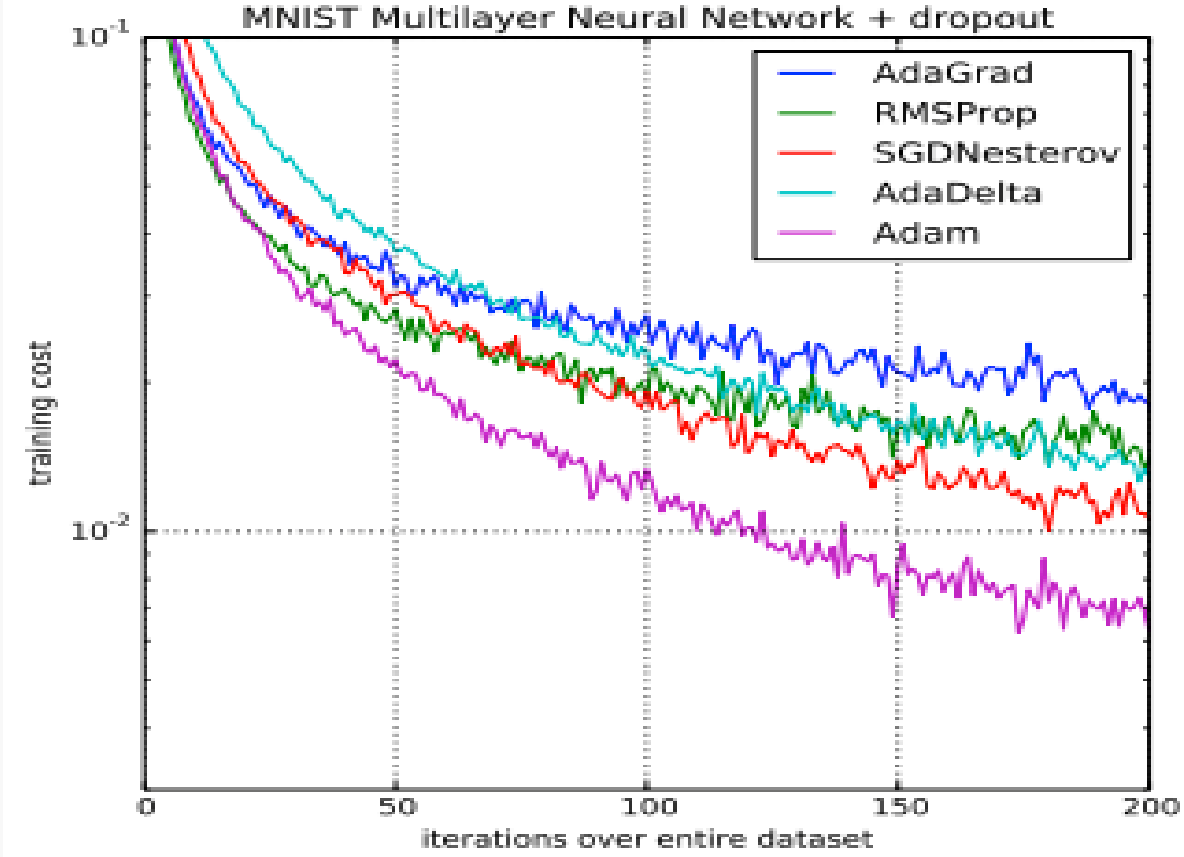
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f(\theta_{t-1})$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla f(\theta_{t-1}))^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{v_t + \epsilon}} \odot m_t$$

As m_t **and** v_t **are initialized as vectors of 0's**, the authors of Adam observed that **they are biased towards zero**, especially during the initial time steps, and when the decay rates are small (i.e. β1 and β2 are close to 1)

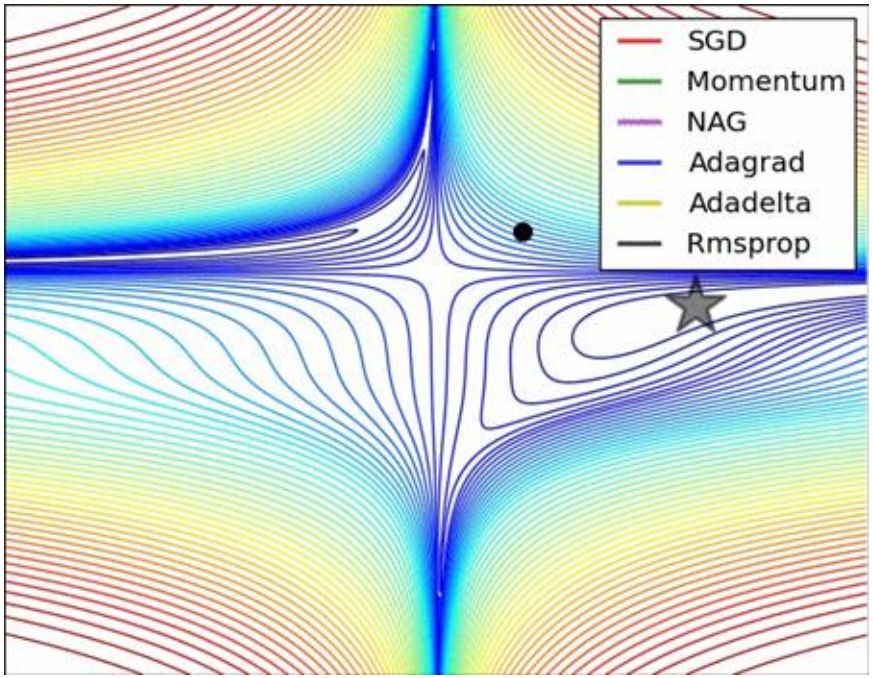
Comparison

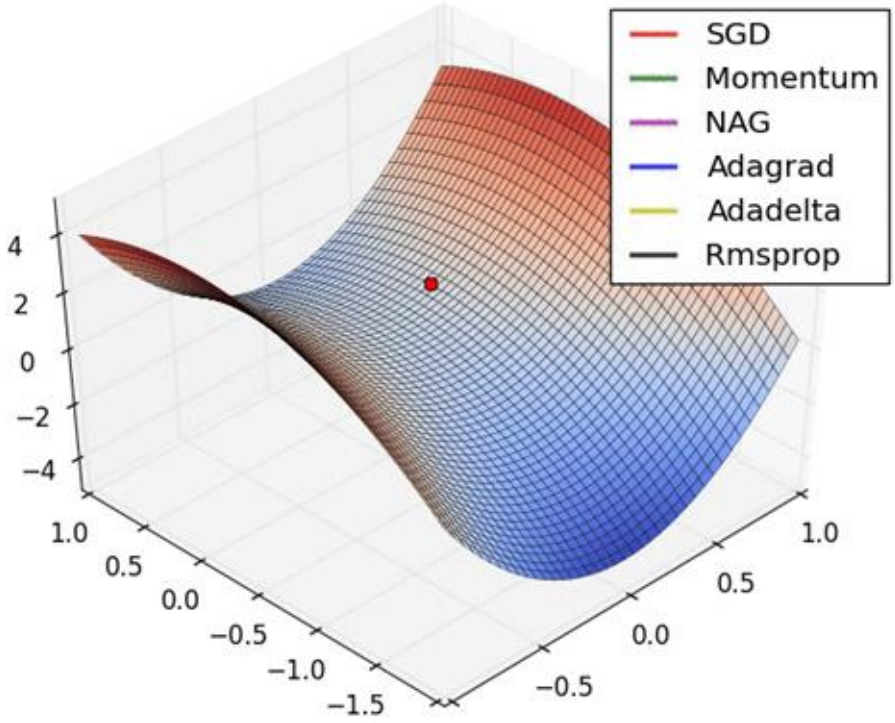


Comparison of Adam to Other Optimization Algorithms Training a Multilayer Perceptron

Taken from Adam: A Method for Stochastic Optimization, 2015.

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." *arXiv preprint arXiv:1609.04747* (2016).





Alec Radford

Why AdamW matters

- The idea behind **L2 regularization or weight decay** is that networks with smaller weights (all other things being equal) are observed
- **Help to overfit less and generalize better**

Algorithm 2 Adam with L_2 regularization and Adam with weight decay (AdamW)

1: **given** $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, w \in \mathbb{R}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: **repeat**
4: $t \leftarrow t + 1$
5: $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
6: $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w\boldsymbol{x}_{t-1}$
7: $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
8: $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9: $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
10: $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
12: $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + w\boldsymbol{x}_{t-1} \right)$
13: **until** *stopping criterion is met*
14: **return** optimized parameters $\boldsymbol{x}_t$

Loshchilov, Ilya, and Frank Hutter. "Fixing weight decay regularization in adam." (2018).

Other recent optimizers

- **AMSGrad**
- QHAdam
- AggMo
- **EVE**

- What and how to use it?
  - Adaptive optimizers like Adam have become **a default choice** for training neural networks
  - Researchers often prefer stochastic gradient descent **(SGD) with momentum** because models **trained with Adam have been observed to not generalize as well**
  - **SGD + momentum and Adam** are good first steps
  - Just use Adam with default parameters
  - **Learning rate decay** always good

1.Loshchilov, I., & Hutter, F. (2019). Decoupled Weight Decay Regularization. In Proceedings of ICLR 2019. ↩

2.Ma, J., & Yarats, D. (2019). Quasi-hyperbolic momentum and Adam for deep learning. In Proceedings of ICLR 2019. ↩

3.Lucas, J., Sun, S., Zemel, R., & Grosse, R. (2019). Aggregated Momentum: Stability Through Passive Damping. In Proceedings of ICLR 2019. ↩

4.Niu, F., Recht, B., Christopher, R., & Wright, S. J. (2011). Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, 1–22.

Recent development: why new optimizers?

- Large deep models (LLMs, multimodal learning) suffer from:
  - Poor curvature awareness (Adam ≠ true second order)
  - Instability at scale
  - Overconfidence and sharp minima (important for clinical AI)

- **Sophia and Muon** are **next-generation optimizers** designed to approximate **second-order information efficiently** while remaining scalable.

Sophia (Second-order Clipped Stochastic Optimization) Optimizer

Classic optimizers:

Optimizer	Uses gradient	Uses curvature
SGD	✅	❌
Adam	✅ (adaptive)	❌
Newton	✅	✅ (Hessian)

Problem:
True Newton updates require computing or inverting the Hessian H, which is infeasible for deep networks.

Sophia approximates **diagonal Hessian information** using stochastic estimates, yielding:

Second-order behavior at Adam-like cost

Liu et al., *Sophia: A Scalable Stochastic Second-Order Optimizer*, arXiv:2305.14342, 2023

Mathematical Foundation

Objective

$$\min_{\theta} \quad \mathcal{L}(\theta)$$

Taylor expansion

$$\mathcal{L}(\theta + \Delta) \approx \mathcal{L}(\theta) + g^{\top}\Delta + \frac{1}{2}\Delta^{\top}H\Delta$$

Newton step:

$$\Delta = -H^{-1}g$$

1

Sophia's Key Approximation

Sophia replaces H with a **diagonal estimate**:

$$H \approx diag(h_1, \ldots, h_d)$$

Each diagonal term is estimated via **gradient outer products**:

$$h_t \approx \mathbb{E}[g_t^2]$$

This is *similar to Adam*, but **interpreted as curvature**, not variance.

2

Sophia Update Rule

Let:

- $g_t = \nabla_{\theta}\mathcal{L}$
- h_t =running estimate of Hessian diagonal
- m_t =momentum

Hessian estimate

$$h_t = \beta h_{t-1} + (1-\beta)g_t^2$$

Parameter update

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{m_t}{\max(h_t, \epsilon)}$$

Liu et al., *Sophia: A Scalable Stochastic Second-Order Optimizer*, arXiv:2305.14342, 2023

Why Sophia is powerful

Key difference from Adam

Adam	**Sophia**
g^2 =variance	g^2 =curvature
Scale learning rate	Scale Newton step

- Approximates **diagonal Newton method**
- Naturally **prevents sharp minima**
- Improves **generalization**

- Works extremely well for:
  - LLMs
  - Vision Transformers
  - Multimodal Learning

Muon Optimizer

Motivation : Sophia handles **curvature**, but large models also suffer from:

- Gradient misalignment across layers
- Poor signal propagation
- Feature collapse

Muon introduces geometry-aware updates preserve **directional structure** in parameter space.

Core Idea : treats optimization as **vector transport in parameter space**, not just **scalar learning rates**. It enforces:

- Orthogonality
- Directional consistency
- Controlled norm growth

•Bernstein et al., *Norm-based Optimization and Implicit Bias*, NeurIPS 2020
•Wang et al., *Understanding Gradient Norms in Deep Learning*, ICLR 2022

Mathematical Intuition and update rule

1

Let gradient matrix $G \in \mathbb{R}^{d \times k}$(e.g., attention layers).

Muon applies **orthogonal projection**:

$$\tilde{G} = \text{Proj}_{\mathcal{O}}(G)$$

This ensures:

$$\tilde{G}^{\top}\tilde{G} \approx I$$

Meaning updates preserve **feature diversity**.

2

Step 1: Normalize gradient direction

$$\hat{g} = \frac{g}{\| g \|}$$

Step 2: Apply momentum and geometry correction

$$\theta_{t+1} = \theta_t - \eta \cdot \hat{g}$$

In practice, Muon uses **block-wise normalization** for:

- Attention heads
- Convolution kernels
- Transformer layers

Muon is an optimizer for 2D parameters of neural network hidden layers

Algorithm 2 Muon

Require: Learning rate η, momentum μ

1: Initialize $B_0 \leftarrow 0$
2: **for** $t = 1, \ldots$ **do**
3: Compute gradient $G_t \leftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1})$
4: $B_t \leftarrow \mu B_{t-1} + G_t$
5: $O_t \leftarrow \text{NewtonSchulz5}(B_t)$
6: Update parameters $\theta_t \leftarrow \theta_{t-1} - \eta O_t$
7: **end for**
8: **return** θ_t

```
# Pytorch code

def newtonschulz5(G, steps=5, eps=1e-
7):

     assert G.ndim == 2

     a, b, c = (3.4445, -4.7750,
     2.0315)

     X = G.bfloat16()

     X /= (X.norm() + eps)

     if G.size(0) > G.size(1):

          X = X.T

          for _ in range(steps): A = X
          @ X.T B = b * A + c * A @ A X
          = a * X + B @ X

     if G.size(0) > G.size(1):

          X = X.T return X
```

Sophia vs Muon (Comparison)

Aspect	Sophia	Muon
Focus	Curvature	Geometry
Second-order	Approximate Hessian	Directional control
Main benefit	Faster + stable convergence	Better representations
Best for	LLMs, classification	Transformers, fusion

Sophia + Muon hybrid:

- Sophia for **encoder weights**
- Muon for **fusion & attention layers**

Benefits : Newton for stability and Riemannian optimization for geometry

Summary

- Efficient training of DNN models
  - Loss functions:
    - Cross-entropy Loss
      - Zero-one loss
      - Binary Cross Entropy Loss and
      - Categorical Cross Entropy Loss
    - Focal Loss
  - Optimization methods
    - SGD + momentum
    - RMSprop
    - Adam
    - AdamW
    - Sophia and muon many more …
- What's next?
  - Deep Neural Network architectures