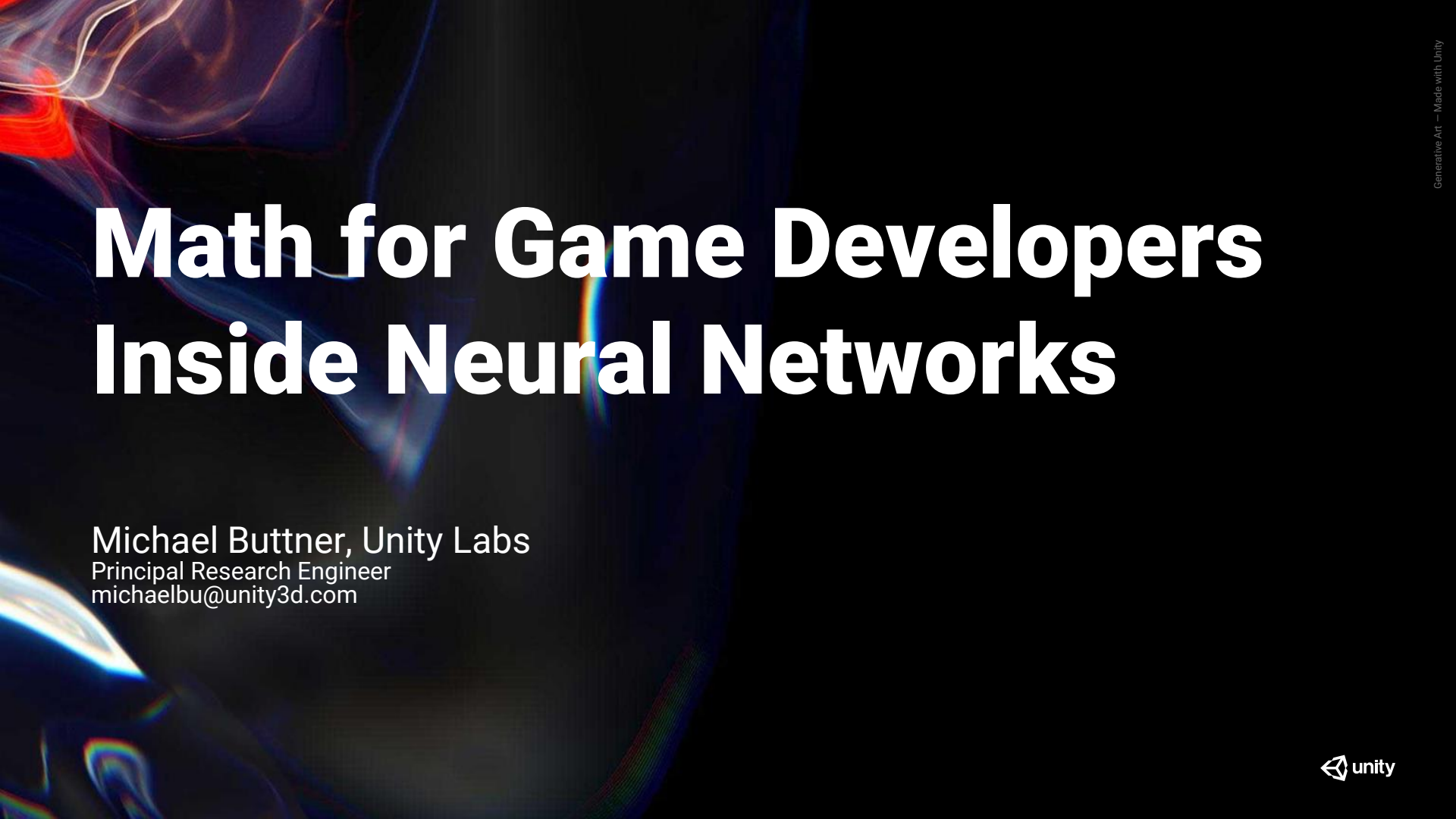
unity

GDC 2019



Math for Game Developers Inside Neural Networks

Michael Buttner, Unity Labs
Principal Research Engineer
michaelbu@unity3d.com

Generative Art – Made with Unity

unity

Image descriptions

Man in black shirt is playing guitar.

Construction worker in orange safety vest is working on road.

Two young girls are playing with lego toy.

Boy is doing backflip on wakeboard.

[Karpathy and Fei-Fei 2015]

Style transfer

+

=

[Gatys, Leon A and Ecker, Alexander S and Bethge, Matthias 2015]

Image enhancement

https://github.com/alexjc/neural-enhance

unity

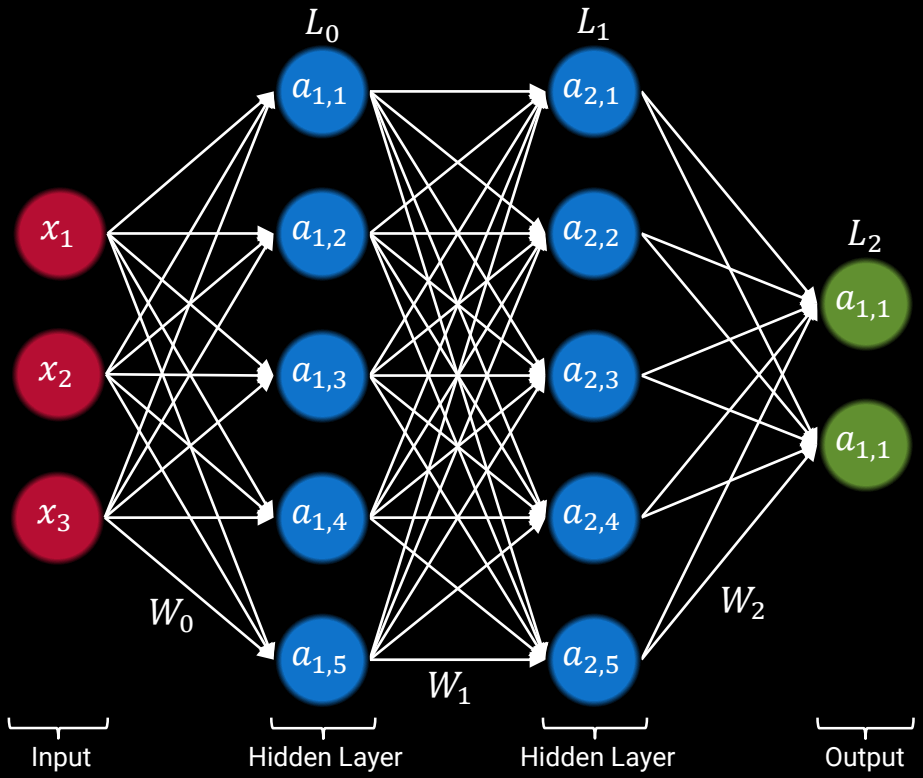
Beyond images and classification

Vast amount of neural network applications are “offline”
Majority of examples focus on image-related problems
NN inference is usually implemented inefficiently

- Neural Networks in performance critical code
- Inference and Training without frameworks
- Data-oriented design

unity

Data oriented NN Inference



Multilayer Perceptron (MLP) – DenseLayer - FullyConnectedLayer

Optimize data layout for memory access patterns

Continuous memory access to avoid cache misses

Minimize memory access

No dynamic memory allocation

128-bit SSE Load & Store instructions

$$a_l = \sigma(W_l a_{l-1} + b_l)$$

$$\underbrace{\hat{y} = \sigma\left(W_2\left(\sigma(W_1 + (\sigma(W_0 x + b_0))b_1)\right) + b_2\right)}_{\text{Forward Pass | Neural Network Inference}}$$

unity

Instruction Timings

AMD Ryzen

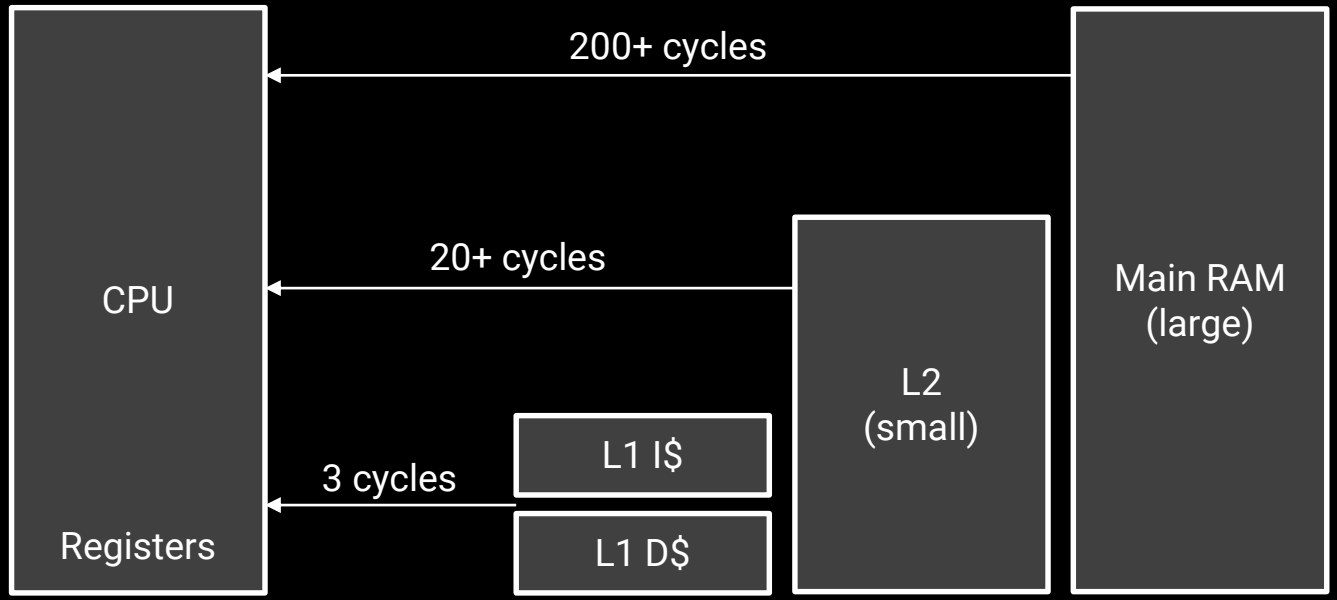
Instruction	Latency
SQRTSS/PS	9-10
VSQRTPS	9-10
SQRTSD/PD	14-15
VSQRTPD	14-15
RSQRTSS/PS	5
VRSQRTPS	5

Instruction	Latency
FSIN	50-170
FCOS	50-115
FSINCOS	60-120
FPTAN	~90
FPATAN	50-160

https://www.agner.org/optimize/instruction_tables.pdf

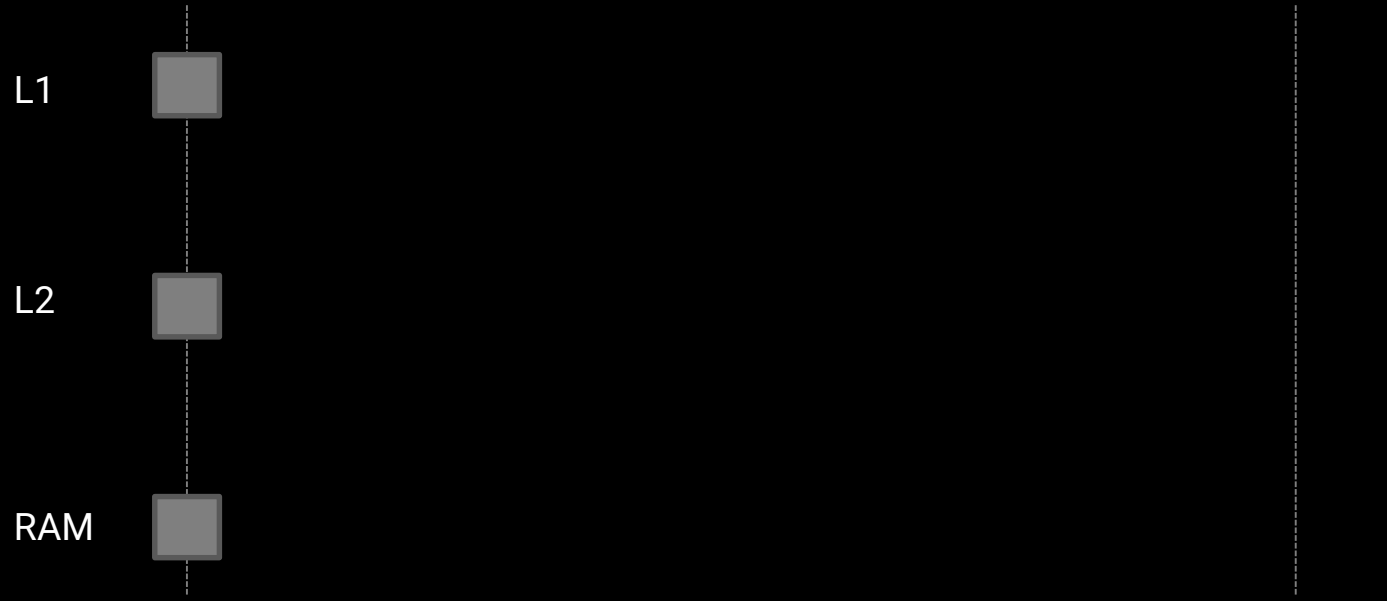
By Agner Fog. Technical University of Denmark.

Memory Caching



CppCon 2014: Mike Acton "Data-Oriented Design and C++"

Data Source Latency



CppCon 2014: Mike Acton "Data-Oriented Design and C++"

DirectX Math

Memory types

Register types

```
XMFLOAT4A _a(1.0f, 2.0f, 3.0f, 4.0f);
XMFLOAT4A _b(5.0f, 6.0f, 7.0f, 8.0f);

XMFLOAT4A _c;

void DirectXMath()
{
    const XMVECTOR a(XMLoadFloat4A(&_a));
    const XMVECTOR b(XMLoadFloat4A(&_b));

    const XMVECTOR c(XMVectorAdd(a, b));

    XMStoreFloat4A(&_c, c);
}
```

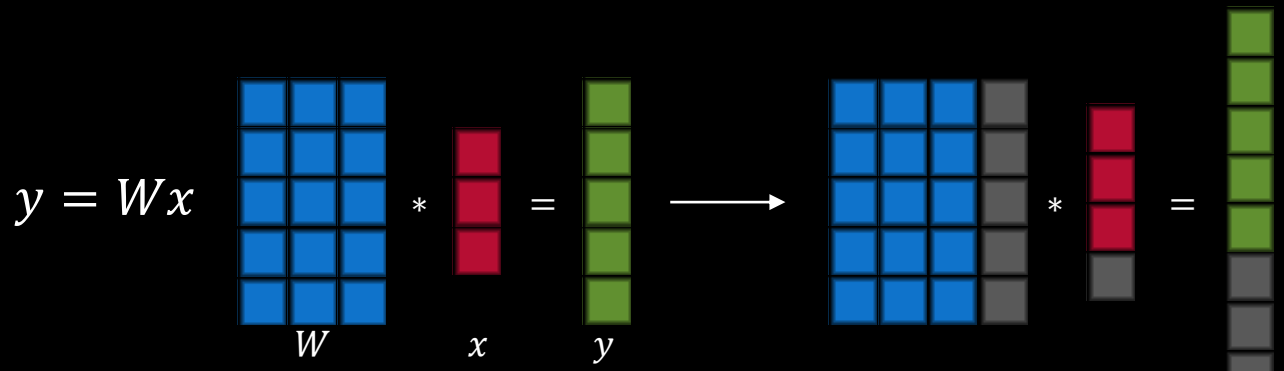
XMLoad/StoreFloat4A map directly to *_mm_load_ps* and *_mm_store_ps*

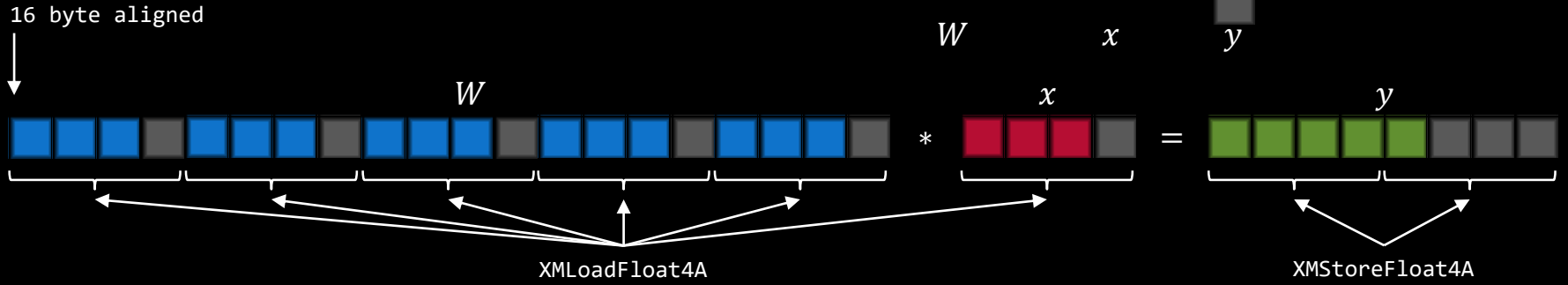
"An all inline SIMD C++ linear algebra library for use in games and graphics apps"

https://github.com/Microsoft/DirectXMath

Alignment & Padding

Row major layout & 128-bit padding for vectors and matrices





unity

MatrixMN & VectorN

```
struct MatrixMN
{
    float* ptr;
    uint16_t numRows;
    uint16_t numColumns;

    INLINE MatrixMN(const uint32_t rows, const uint32_t columns);

    INLINE MemoryRequirements GetMemoryRequirements() const;
    INLINE MemoryRequirements operator()() const;
    INLINE MemoryBlock& operator()(MemoryBlock& memoryBlock);

    INLINE uint32_t GetNumRows() const;
    INLINE uint32_t GetAlignedNumRows() const;
    INLINE uint32_t GetNumColumns() const;
    INLINE uint32_t GetAlignedNumColumns() const;

    INLINE float* GetPtr();
    INLINE const float* GetPtr() const;
    INLINE float* GetRowPtr(const uint32_t row);
    INLINE const float* GetRowPtr(const uint32_t row) const;
    INLINE XMFLOAT4A* GetPtrSSE();
    INLINE const XMFLOAT4A* GetPtrSSE() const;
};
```

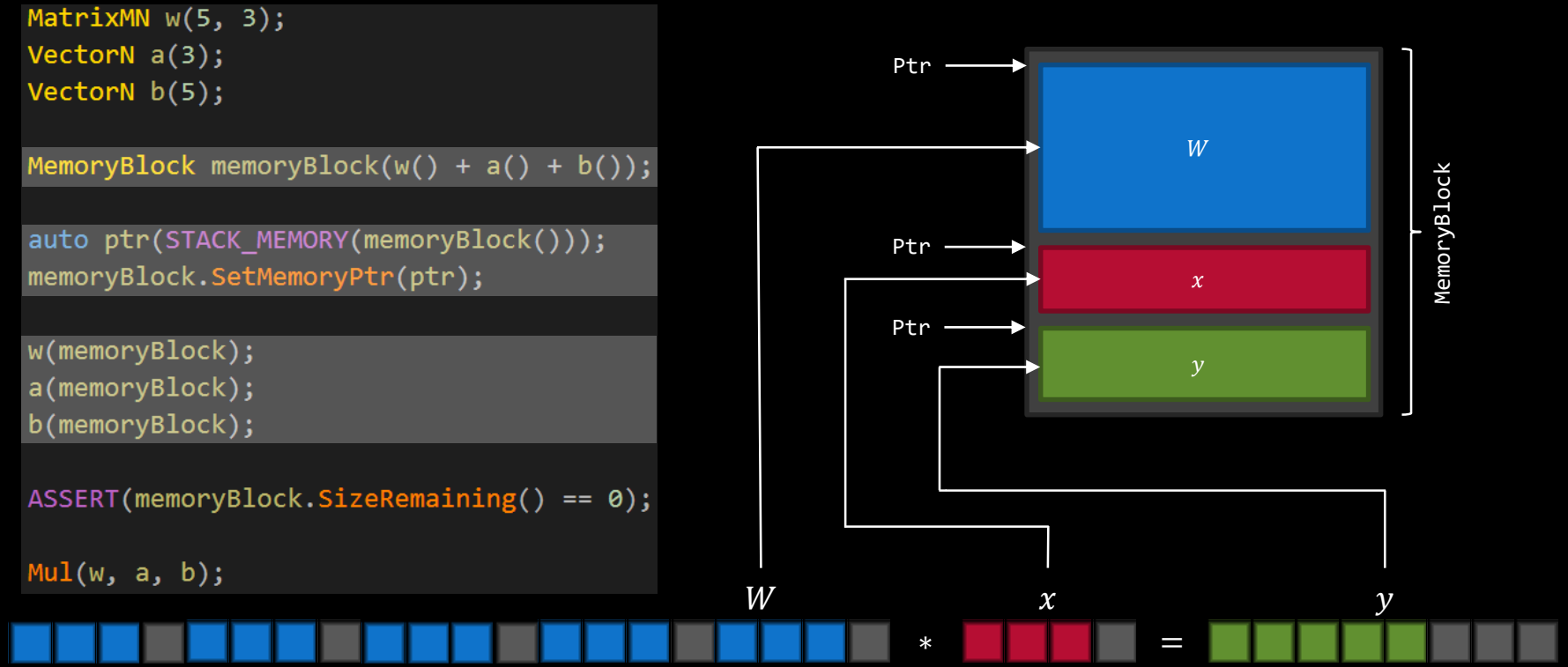
```
struct VectorN
{
    float* ptr;
    uint16_t numValues;

    INLINE VectorN(const uint32_t values);

    INLINE MemoryRequirements GetMemoryRequirements() const;
    INLINE MemoryRequirements operator()() const;
    INLINE MemoryBlock& operator()(MemoryBlock& memoryBlock);

    INLINE uint32_t GetNumValues() const;
    INLINE uint32_t GetAlignedNumValues() const;

    INLINE float* GetPtr();
    INLINE const float* GetPtr() const;
    INLINE XMFLOAT4A* GetPtrSSE();
    INLINE const XMFLOAT4A* GetPtrSSE() const;
};
```

Matrix and Vector reference memory

Decoupled memory management

MemoryBlocks & MemoryRequirements

```
MatrixMN w(5, 3);
VectorN a(3);
VectorN b(5);

MemoryBlock memoryBlock(w() + a() + b());

auto ptr(STACK_MEMORY(memoryBlock()));
memoryBlock.SetMemoryPtr(ptr);

w(memoryBlock);
a(memoryBlock);
b(memoryBlock);

ASSERT(memoryBlock.SizeRemaining() == 0);

Mul(w, a, b);
```



Stack allocation

Reasonably sized memory blocks can be allocated on the stack

```
sub rsp, <size>
```

```
#define ALIGNED_ALLOCA(size, alignment) align_ptr( \
    reinterpret_cast<size_t>(alloca((size) + alignment)), alignment)

#define STACK_MEMORY(requirements) \
    ((void*)ALIGNED_ALLOCA((requirements.Size()), \
        (requirements.Alignment())))
```

Requires tight control of scopes

~~`void* Allocate(size_t size) { return alloca(size); }`~~

SSE MatMul

```
const XMVECTOR v(XMLoadFl
```

```
const XMVECTOR
const XMVECTOR
const XMVECTOR

XMStoreFloat4A
```

```
void Mul(const MatrixMN& lhs, const VectorN& rhs, VectorN& result)
{
    ASSERT(lhs.GetNumRows() == result.GetNumValues());
    ASSERT(lhs.GetNumColumns() == rhs.GetNumValues());

    const uint32_t advance(lhs.GetAlignedNumColumns() >> 2);
    const uint32_t blockAdvance(advance * 3);

    XMFLOAT4A* writePtr(result.GetPtrSSE());

    const XMFLOAT4A* r0(lhs.GetPtrSSE());
    const XMFLOAT4A* r1(r0 + advance);
    const XMFLOAT4A* r2(r1 + advance);
    const XMFLOAT4A* r3(r2 + advance);

    const uint32_t numRows(lhs.GetAlignedNumRows() >> 2);
    for (uint32_t row = 0; row < numRows; ++row)
    {
        const XMFLOAT4A* readPtr(rhs.GetPtrSSE());

        XMVECTOR d0(XMVectorZero());
        XMVECTOR d1(XMVectorZero());
        XMVECTOR d2(XMVectorZero());
        XMVECTOR d3(XMVectorZero());

        for (uint32_t i = 0; i < advance; ++i)
        {
            const XMVECTOR v(XMLoadFloat4A(readPtr++));

            d0 = XMVectorAdd(d0, XMVector4Dot(XMLoadFloat4A(r0++), v));
            d1 = XMVectorAdd(d1, XMVector4Dot(XMLoadFloat4A(r1++), v));
            d2 = XMVectorAdd(d2, XMVector4Dot(XMLoadFloat4A(r2++), v));
            d3 = XMVectorAdd(d3, XMVector4Dot(XMLoadFloat4A(r3++), v));
        }

        const XMVECTOR d4(XMVectorPermute<0, 1, 4, 5>(d0, d1));
        const XMVECTOR d5(XMVectorPermute<0, 1, 4, 5>(d2, d3));
        const XMVECTOR d(XMVectorPermute<0, 2, 4, 6>(d4, d5));

        XMStoreFloat4A(writePtr++, d);

        r0 += blockAdvance;
        r1 += blockAdvance;
        r2 += blockAdvance;
        r3 += blockAdvance;
    }
}
```

```
ector4
ector4
ector4
ector4
```

```
oat4A(r0++), v))
oat4A(r1++), v))
oat4A(r2++), v))
oat4A(r3++), v))
```

```
>(d0,
>(d2,
(d4, d
```

unity

Composite operations

All this pain just for a simple matrix multiplication?

“Layer Fusion” | Composite operations

$\hat{y} = \sigma(Wx + b)$ can be combined
into a single operation

Reduce memory access & hide memory latency

$$\sigma(Wx + b) \longrightarrow \begin{aligned} u &= Wx \\ v &= u + b \\ \underbrace{s}_{\text{Temporary results}} &= \sigma(v) \end{aligned}$$

```
void Sigmoid(MatrixMN& lhs)
{
    const XMVECTOR one(XMVectorReplicate(1.0f));

    XMFLOAT4A* ptr(lhs.GetPtrSSE());

    const uint32_t numValues(lhs.GetAlignedNumColumns() >> 2);
    for (uint32_t i = 0; i < numValues; ++i)
    {
        const XMVECTOR x(XMLoadFloat4A(ptr));

        // f(x) = 1 / (1 + e^-x)
        const XMVECTOR result(
            XMVectorMax(x, XMVectorZero()) +
                XMVectorExpE(XMVectorMin(x, XMVectorZero())) - one);

        // f(x) = x / (1 + abs(x)) | "Fast sigmoid"
        //const XMVECTOR result(XMVectorDivide(x,
        //  XMVectorAdd(one, XMVectorAbs(x))));

        XMStoreFloat4A(ptr++, result);
    }
}
```

SSE FullyConnected Layer

```
const XMVECTOR d4(XMVectorPermute<0, 1, 4, 5>(d0, d1));
const XMVECTOR d5(XMVectorPermute<0, 1, 4, 5>(d2, d3));
const XMVECTOR d(XMVectorPermute<0, 2, 4, 6>(d4, d5));

XMStoreFloat4A(writePtr++, d);
```

```
const XMVECTOR d4(XMVectorPermute<0, 1, 4, 5>(d0, d1));
const XMVECTOR d5(XMVectorPermute<0, 1, 4, 5>(d2, d3));
const XMVECTOR d(XMVectorPermute<0, 2, 4, 6>(d4, d5));

const XMVECTOR x(XMVectorAdd(d, XMLoadFloat4A(biasPtr++)));

// f(x) = x / (1 + abs(x)) | "Fast sigmoid"
const XMVECTOR y(XMVectorDivide(x,
    XMVectorAdd(one, XMVectorAbs(x))));

XMStoreFloat4A(writePtr++, y);
```

```
void Mul(const MatrixMN& lhs, const VectorN& rhs, VectorN& result)
{
    ASSERT(lhs.GetNumRows() == result.GetNumValues());
    ASSERT(lhs.GetNumColumns() == rhs.GetNumValues());

    const uint32_t advance(lhs.GetAlignedNumColumns() >> 2);
    const uint32_t blockAdvance(advance * 3);

    XMFLOAT4A* writePtr(result.GetPtrSSE());

    const XMFLOAT4A* r0(lhs.GetPtrSSE());
    const XMFLOAT4A* r1(r0 + advance);
    const XMFLOAT4A* r2(r1 + advance);
    const XMFLOAT4A* r3(r2 + advance);

    const uint32_t numRows(lhs.GetAlignedNumRows() >> 2);
    for (uint32_t row = 0; row < numRows; ++row)
    {
        const XMFLOAT4A* readPtr(rhs.GetPtrSSE());

        XMVECTOR d0(XMVectorZero());
        XMVECTOR d1(XMVectorZero());
        XMVECTOR d2(XMVectorZero());
        XMVECTOR d3(XMVectorZero());

        for (uint32_t i = 0; i < advance; ++i)
        {
            const XMVECTOR v(XMLoadFloat4A(readPtr++));

            d0 = XMVectorAdd(d0, XMVector4Dot(XMLoadFloat4A(r0++), v));
            d1 = XMVectorAdd(d1, XMVector4Dot(XMLoadFloat4A(r1++), v));
            d2 = XMVectorAdd(d2, XMVector4Dot(XMLoadFloat4A(r2++), v));
            d3 = XMVectorAdd(d3, XMVector4Dot(XMLoadFloat4A(r3++), v));
        }

        const XMVECTOR d4(XMVectorPermute<0, 1, 4, 5>(d0, d1));
        const XMVECTOR d5(XMVectorPermute<0, 1, 4, 5>(d2, d3));
        const XMVECTOR d(XMVectorPermute<0, 2, 4, 6>(d4, d5));

        XMStoreFloat4A(writePtr++, d);

        r0 += blockAdvance;
        r1 += blockAdvance;
        r2 += blockAdvance;
        r3 += blockAdvance;
    }
}
```

Performance gain

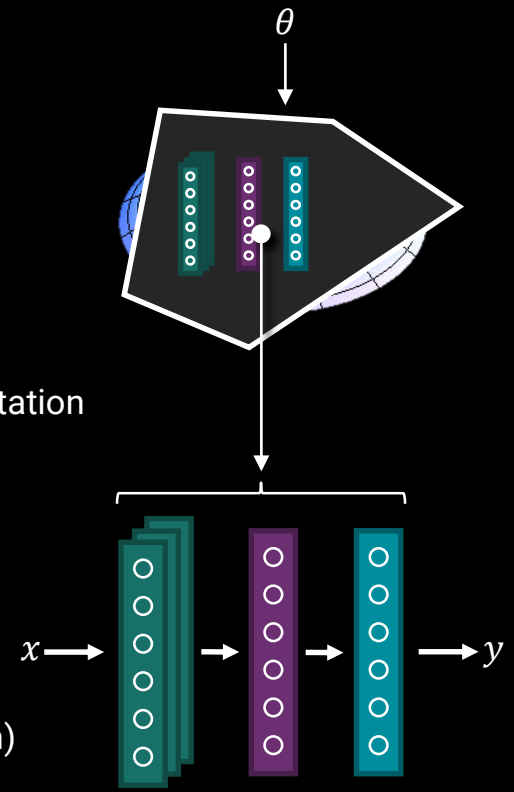
Performance gain depends on network architecture and feature vector size

Complex architectures gain the most

- Experiment with naïve implementation
- Lock down architecture
- Side-by-side implementation
- Profile and optimize

Complex manifold example

- Feature vectors size 400+ values
- From 3.0 ms to 0.7 ms (76% gain)




```
float* writePtr(result.GetPtr());

const XMFLOAT4A* w0(m_weights[index][index0].GetPtrSSE());
const XMFLOAT4A* w1(m_weights[index][index1].GetPtrSSE());
const XMFLOAT4A* w2(m_weights[index][index2].GetPtrSSE());
const XMFLOAT4A* w3(m_weights[index][index3].GetPtrSSE());

const uint32_t numDotProducts(m_weights[index][index0].GetRowStride() >> 2);

const uint32_t numRows(result.GetNumRows());
for (uint32_t row = 0; row < numRows; ++row)
{
    const XMFLOAT4A* readPtr(input.GetPtrSSE());

    XMVECTOR dot(XMVectorZero());
    for (uint32_t i = 0; i < numDotProducts; ++i)
    {
        const XMVECTOR a(XMLoadFloat4A(w0++));
        const XMVECTOR b(XMLoadFloat4A(w1++));
        const XMVECTOR c(XMLoadFloat4A(w2++));
        const XMVECTOR d(XMLoadFloat4A(w3++));

        dot = XMVectorAdd(dot,
            XMVector4Dot(Blend(a, b, c, d, t),
                XMLoadFloat4A(readPtr++)));
    }

    writePtr[row] = XMVectorGetX(dot);
}
```

Neural Network Training

The goal is to adjust all W_l, b_l such that *for any* training sample (x, y) the loss L becomes 0

$\frac{\partial L}{\partial W_l}, \frac{\partial L}{\partial b_l}$ tell us how to adjust W_l, b_l for any given (x, y)

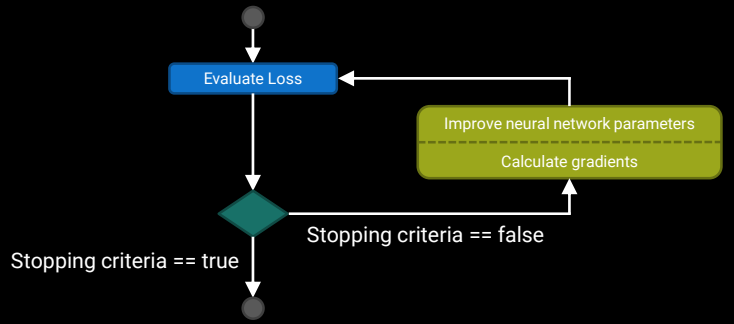
Loss function can include additional constraints
For example maintain unit quaternion property

$$N(x) = \sigma(Wx + b)$$

$$\hat{y} = N_2\left(N_1\left(N_0(x)\right)\right)$$

$$L = \frac{1}{2}(\hat{y} - y))^2$$

```
initialize(net)
for epoch = 1,…,K do
            for batch = 1,…,#data/b do
                        x,y ← randomly draw b samples
                        z ← forward(net, x)
                        l ← loss(z,y)
                        grad ← backward(l)
                        update(net, grad)
            end for
end for
```



unity

Neural Network Training

```
void Train(const Tensor& input, const Tensor& output,
           const uint32_t numEpochs, const Scalar eta)
{
    for (uint32_t i = 0; i < numEpochs; ++i)
    {
        Epoch(input, output, eta);
    }
}

void Epoch(const Tensor& input, const Tensor& output, const Scalar eta)
{
    ASSERT(input.GetNumRows() == output.GetNumRows());

    const uint32_t numLayers(uint32_t(weights.Length()));
    const uint32_t numTrainingSamples(uint32_t(input.GetNumRows()));
    for (uint32_t i = 0; i < numTrainingSamples; ++i)
    {
        CalculateGradients(input.GetRow(i), output.GetRow(i));

        UpdateWeightsAndBiases(eta);
    }
}
```

```
void UpdateWeightsAndBiases(const Scalar factor)
{
    ASSERT(weights.Length() == gradient_weights.Length());
    const uint32_t numWeights(uint32_t(weights.Length()));
    for (uint32_t i = 0; i < numWeights; ++i)
    {
        ElementwiseSub(weights[i], gradient_weights[i], factor);
    }

    ASSERT(biases.Length() == gradient_biases.Length());
    const uint32_t numBiases(uint32_t(biases.Length()));
    for (uint32_t i = 0; i < numBiases; ++i)
    {
        ElementwiseSub(biases[i], gradient_biases[i], factor);
    }
}
```

Calculate Gradients

$$\delta^L = (a^L - y) \odot \sigma' z^L$$

$$\delta^l = \left((w^{l+1})^T \delta^{l+1} \right) \odot \sigma' z^l$$

$$\frac{\partial C}{\partial b^l} = \delta^l$$

$$\frac{\partial C}{\partial w^l} = a^{l-1} \delta^l$$

```
void CalculateGradients(const Tensor& x, const Tensor& y)
{
    const uint32_t numLayers(uint32_t(weights.Length()));
    const uint32_t numLayersPlusOne(numLayers + 1);
    const uint32_t numLayersMinusOne(numLayers - 1);

    STACK_ARRAY(Tensor, zs, numLayers);
    STACK_ARRAY(Tensor, activations, numLayersPlusOne);

    activations[0] = x;

    for (uint32_t i = 0; i < numLayers; ++i)
    {
        zs[i] = Add(Mul(weights[i], activations[i]), biases[i]);
        activations[i + 1] = Sigmoid(zs[i]);
    }

    Tensor delta(
        ElementwiseMul(Sub(activations[numLayers], y),
            SigmoidPrime(zs[numLayersMinusOne])));

    gradient_biases[numLayersMinusOne] = delta;
    gradient_weights[numLayersMinusOne] =
        OuterProduct(delta, activations[numLayersMinusOne]);

    for (uint32_t i = numLayersMinusOne; i > 0; --i)
    {
        const uint32_t num_zs(uint32_t(zs.Size()));
        const uint32_t num_activations(uint32_t(activations.Size()));

        const Tensor& z(zs[i - 1]);
        const Tensor sp(SigmoidPrime(z));

        delta = ElementwiseMul(Mul(Transpose(weights[i]), delta), sp);

        gradient_biases[i - 1] = delta;
        gradient_weights[i - 1] = OuterProduct(delta, activations[i - 1]);
    }
}
```

Forward Pass

Backward Pass

Michael Nielsen
http://neuralnetworksanddeeplearning.com/chap2.html
Sully Chen
https://www.youtube.com/watch?v=gl3lfL-g5mA

Beyond Backpropagation

What if we are not using a simple MLP network?

What if we want to experiment with different architectures?

$$y(\theta) = \sigma\big(u(\theta)x + v(\theta)\big)$$
$u(\theta)$=…
$v(\theta)$=…

```
Scalar x1(2);
Scalar x2(3);

Scalar loss = cos(x1) + x1 * exp(x2);

const float dydx1(x1.derivative());
const float dydx2(x2.derivative());
```

Backpropagation is just a special case of reverse-mode automatic differentiation, applied to a neural network

“AD allows exact and efficient calculation of derivatives, by systematically invoking the chain rule of calculus at the elementary operator level during program execution”

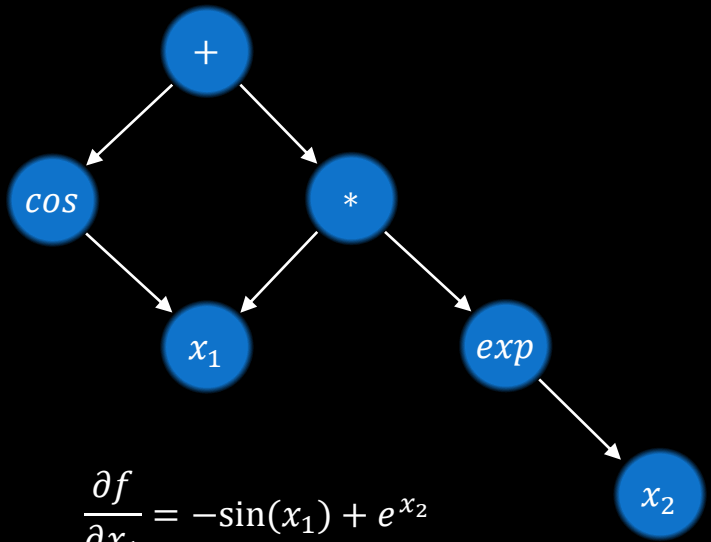
Neural networks are more than just a series of “layers”

Algorithmic Differentiation

$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$

Expressions can be divided into elementary operations

Computational graphs are a subset of abstract syntax trees



$$
\begin{aligned}
f(x_1, x_2) &= w_1 & \dot{f}(x_1, x_2) &= \dot{w}_1 \\
w_1 &= w_2 + w_3 & \dot{w}_1 &= \dot{w}_2 + \dot{w}_3 \\
w_2 &= \cos(w_5) & \dot{w}_2 &= -\sin(w_5)\,\dot{w}_5 \\
w_3 &= w_4 w_5 & \dot{w}_3 &= \dot{w}_4 w_5 + \dot{w}_5 w_4 \\
w_4 &= e^{w_6} & \dot{w}_4 &= e^{w_6}\dot{w}_6 \\
w_5 &= x_1 & \dot{w}_5 &= \text{seed} \in \{0,1\} \\
w_6 &= x_2 & \dot{w}_6 &= \text{seed} \in \{0,1\}
\end{aligned}
$$

Forward Primal Trace

$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$

$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Forward primal trace calculates rate of change of f with respect to *one* of its inputs

Adjoint Mathematics

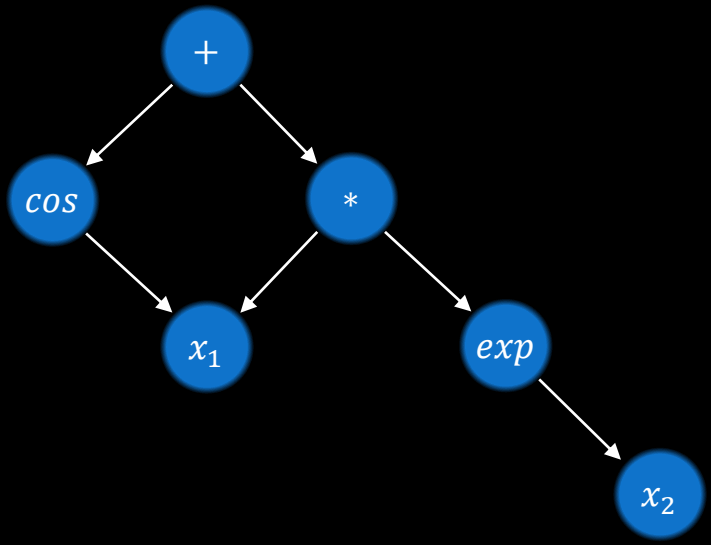
We call f_i the result of the operation on n_i, we denote $f = f_N$ the final result on the op node n_N and C_i the set of arguments to node n_i.

The adjoint a_i of the result f_i on node n_i is the partial derivative of the final result $\mathrm{f} = f_N$ to f_i

$$a_i = \frac{df}{df_i}$$

Since obviously $a_N = \frac{\partial a_N}{\partial a_N} = 1$ and directly from the derivative chain rule, the fundamental adjoint equation is:

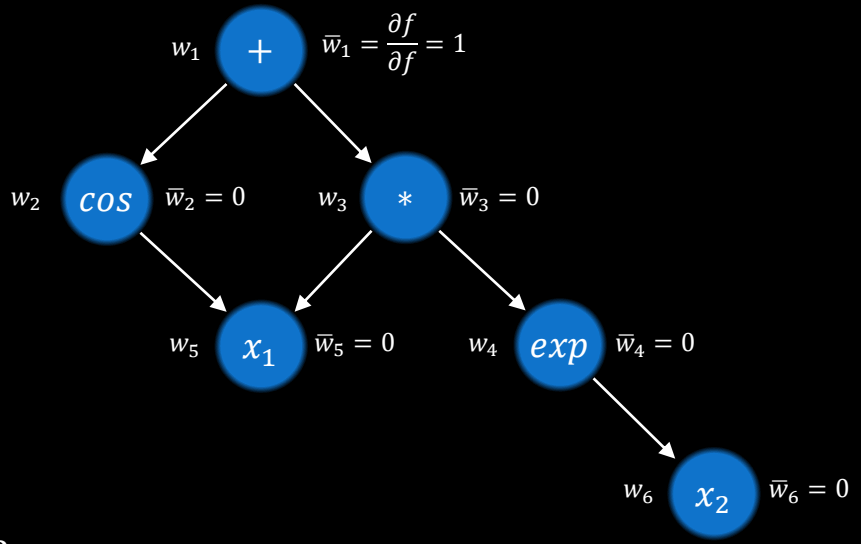
$$a_j = \sum_{i/n_j \in C_i} \frac{\partial f_i}{\partial f_j} a_i$$



Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



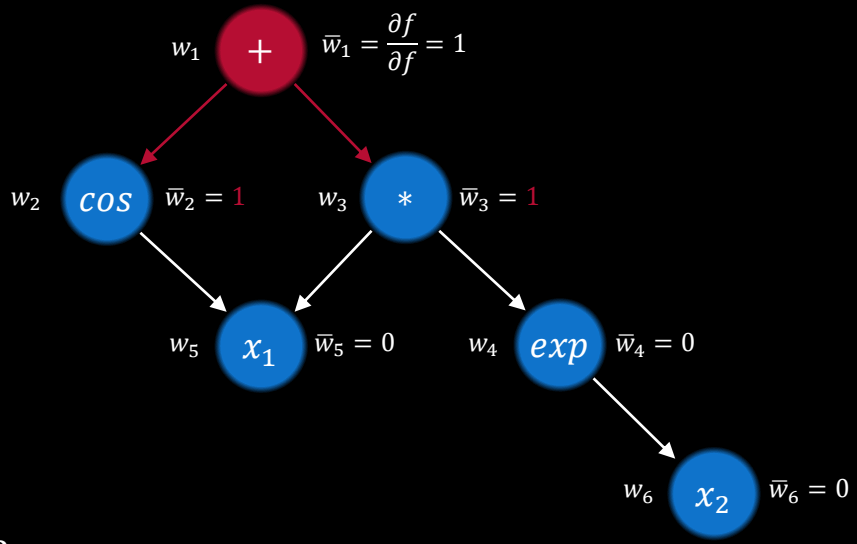
$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



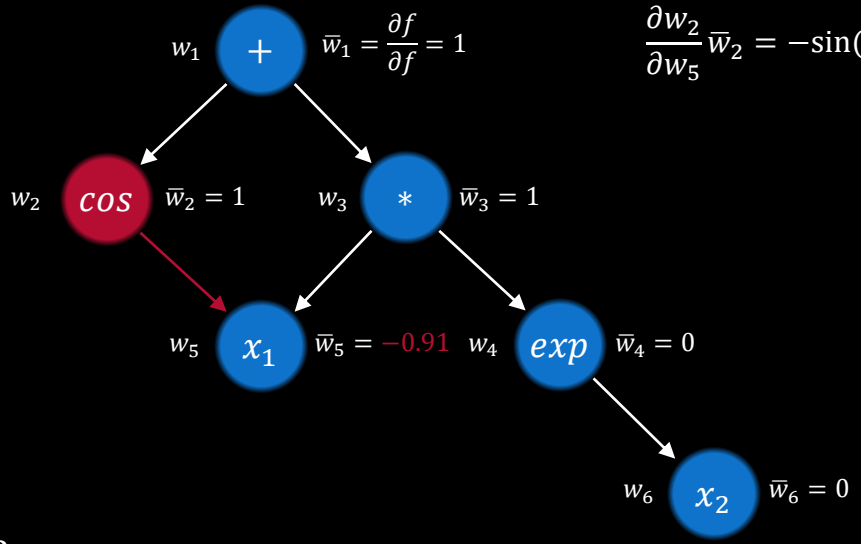
$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



$$\frac{\partial w_2}{\partial w_5}\bar{w}_2 = -\sin(w_5) * 1 = -0.91$$

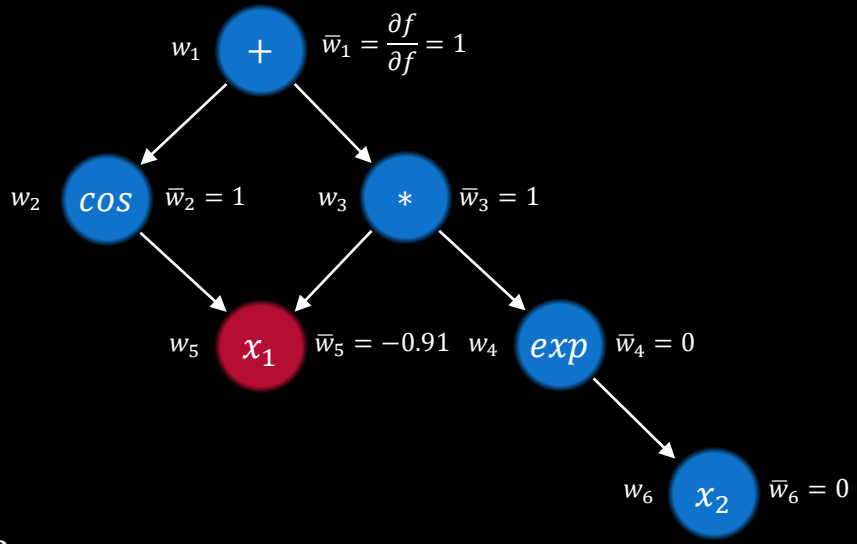
$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



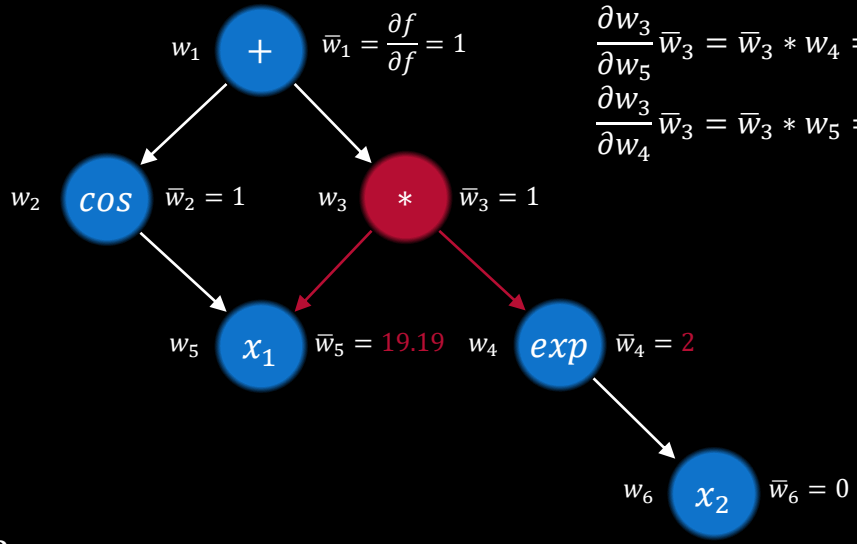
$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



$$\frac{\partial w_3}{\partial w_5}\bar{w}_3 = \bar{w}_3 * w_4 = 1 * e^3 = 1 * 20.1$$
$$\frac{\partial w_3}{\partial w_4}\bar{w}_3 = \bar{w}_3 * w_5 = 1 * 2 = 2$$

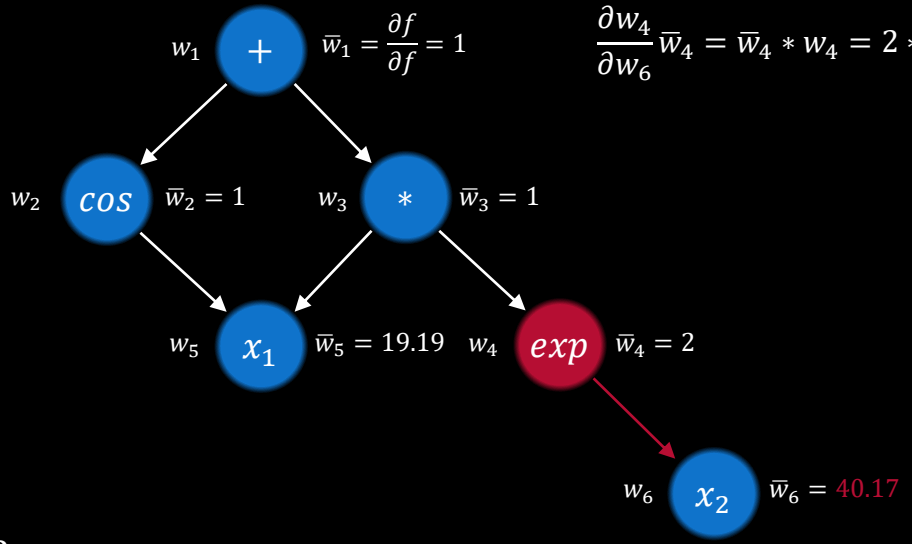
$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



$$\frac{\partial w_4}{\partial w_6} \bar{w}_4 = \bar{w}_4 * w_4 = 2 * e^3 = 40.17$$

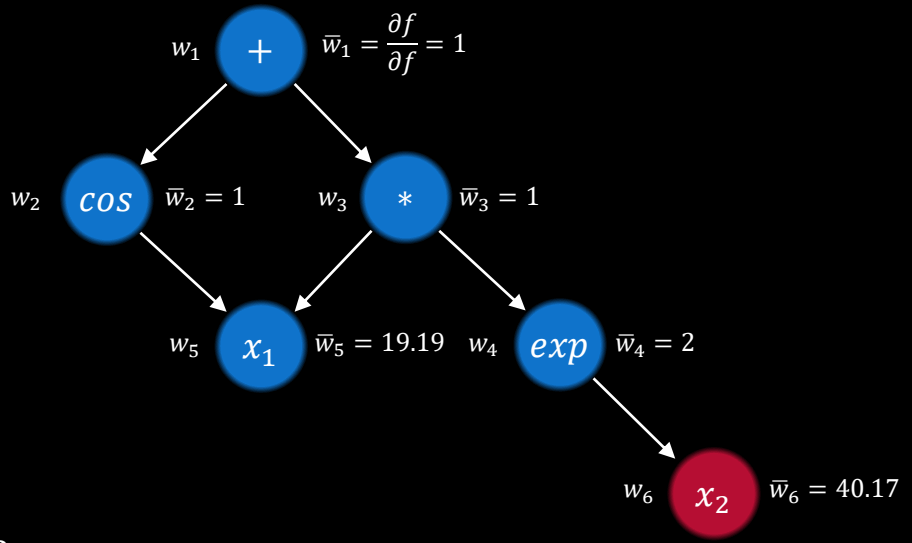
$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



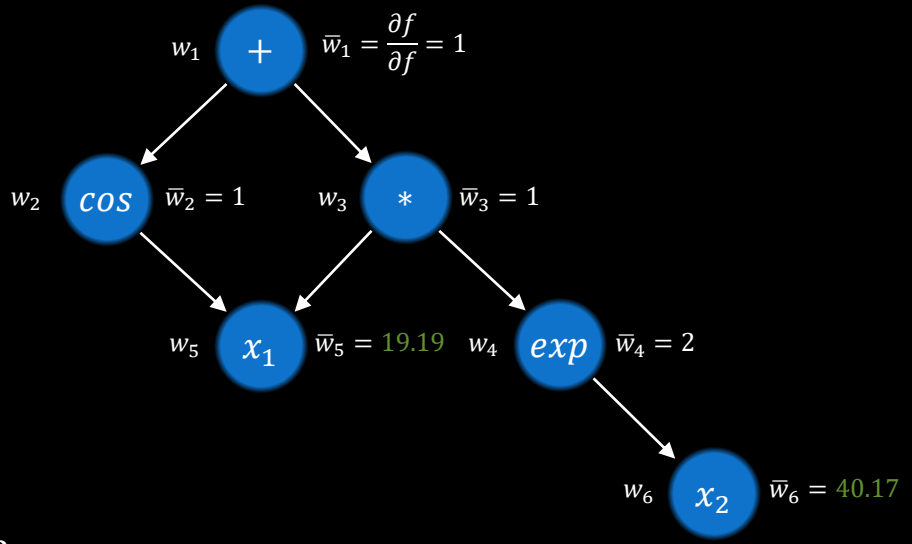
$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Reverse Mode Differentiation

$$f(x_1, x_2) = \cos(x_1) + x_1 \exp(x_2)$$

$$f(x_1, x_2) = w_1$$
$$w_1 = w_2 + w_3$$
$$w_2 = \cos(w_5)$$
$$w_3 = w_4 w_5$$
$$w_4 = e^{w_6}$$
$$w_5 = x_1$$
$$w_6 = x_2$$



$$\frac{\partial f}{\partial x_1} = -\sin(x_1) + e^{x_2}$$
$$\frac{\partial f}{\partial x_2} = e^{x_2} x_1$$

Let $x_1 = 2$ and $x_2 = 3$,
then $\frac{\partial f}{\partial x_1} = 19.19$ and $\frac{\partial f}{\partial x_2} = 40.17$

Algorithmic Adjoint Differentiation

Reverse adjoint trace calculates rate of change of f with respect to *all* of its inputs

Evaluating $\nabla f(x)$ is as fast evaluating $f(x)$

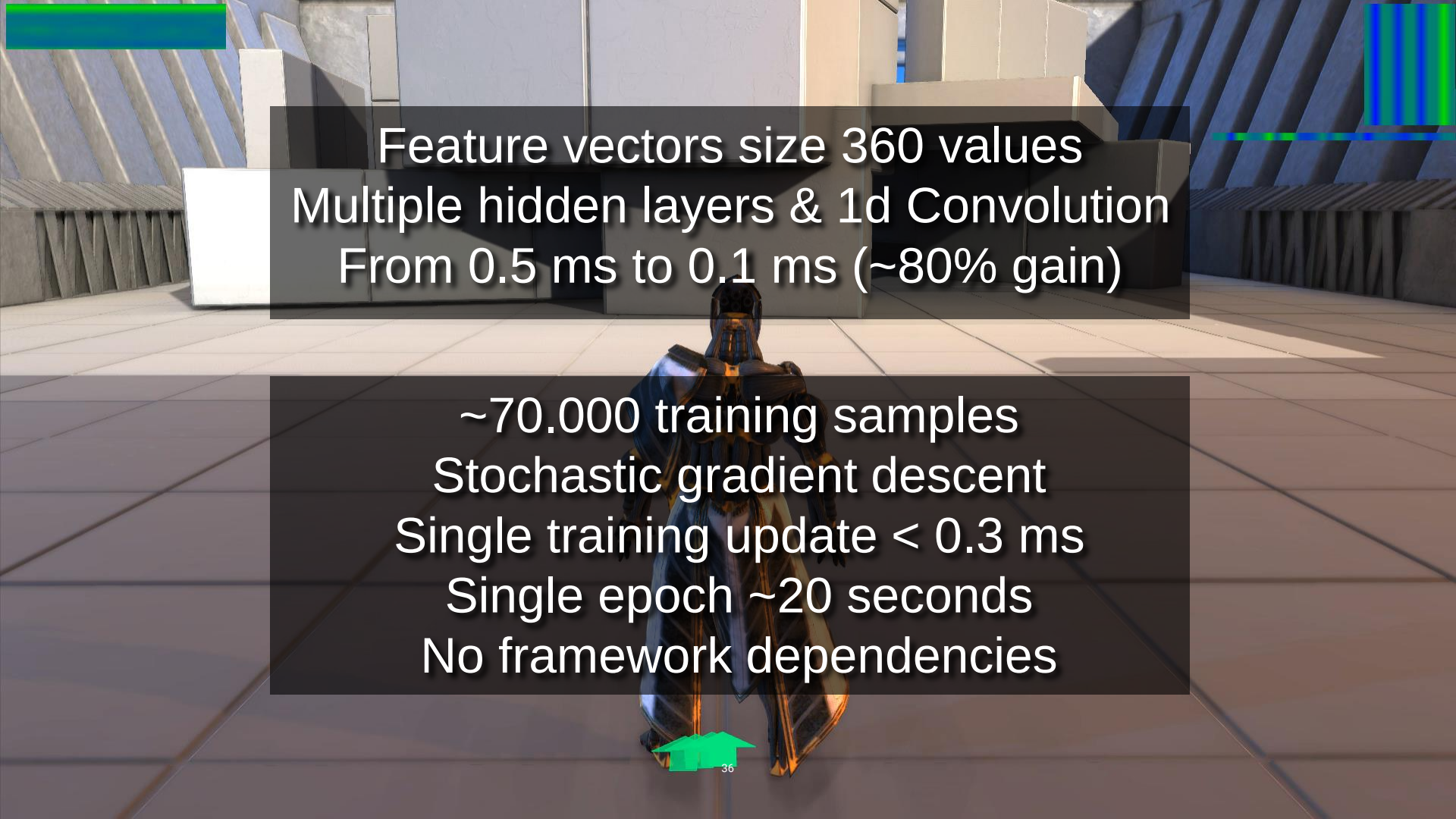
Forward pass evaluates the expression

Reverse pass evaluates $\nabla f(x)$

AAD allows us to use arbitrary control structures like loops or branches, intermediate variables and functions

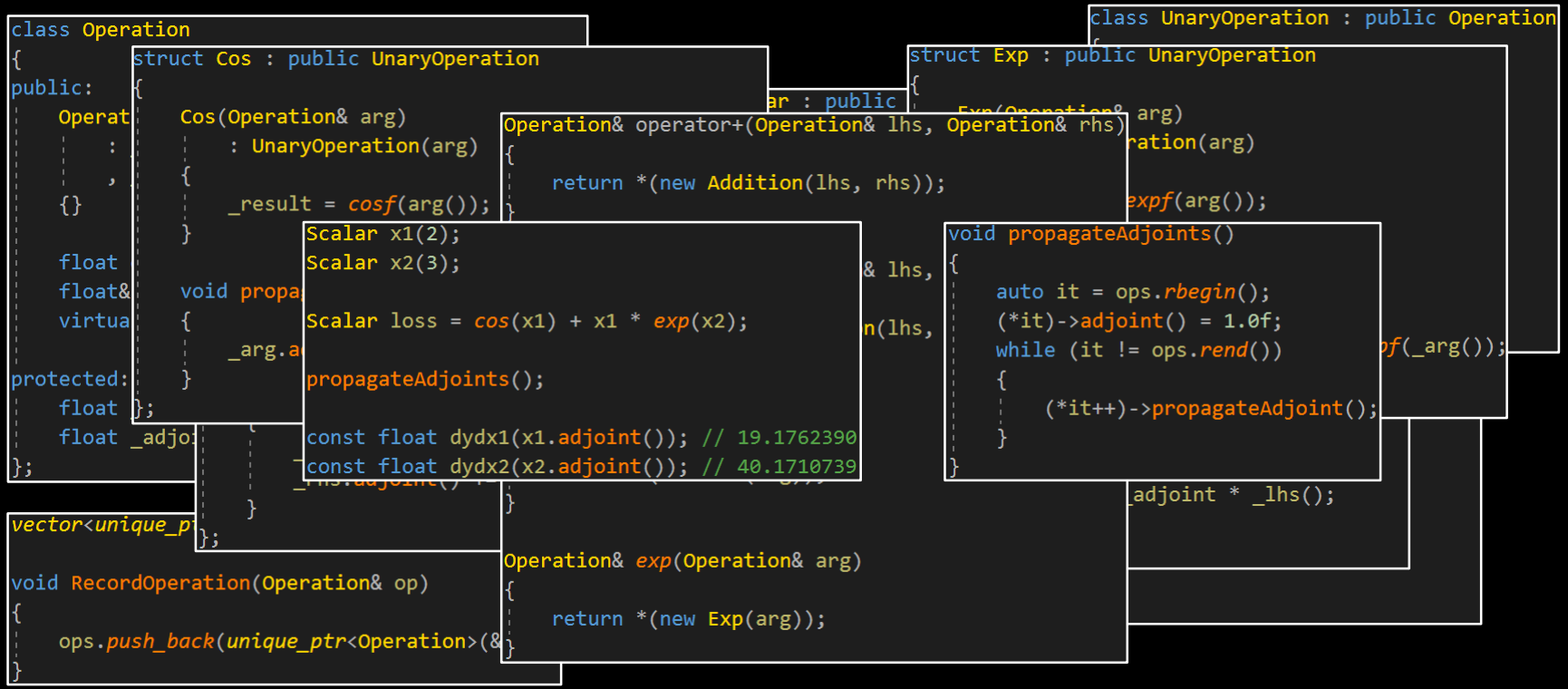
Data-oriented design & SSE

```
void Train(const Tensor& input, const Tensor& output,
           const uint32_t numEpochs, const Scalar eta)
{
    for (uint32_t i = 0; i < numEpochs; ++i)
    {
        Epoch(input, output, eta);
    }
}

void Epoch(const Tensor& input, const Tensor& output, const Scalar eta)
{
    ASSERT(input.GetNumRows() == output.GetNumRows());

    const uint32_t numLayers(uint32_t(weights.Length()));
    const uint32_t numTrainingSamples(uint32_t(input.GetNumRows()));
    for (uint32_t i = 0; i < numTrainingSamples; ++i)
    {
        CalculateGradients(input.GetRow(i), output.GetRow(i));

        UpdateWeightsAndBiases(eta);
    }
}
```


Feature vectors size 360 values
Multiple hidden layers & 1d Convolution
From 0.5 ms to 0.1 ms (~80% gain)
~70.000 training samples
Stochastic gradient descent
Single training update < 0.3 ms
Single epoch ~20 seconds
No framework dependencies
36

Michael Buttner, Unity Labs
Principal Research Engineer
michaelbu@unity3d.com

Algorithmic Adjoint Differentiation



unity