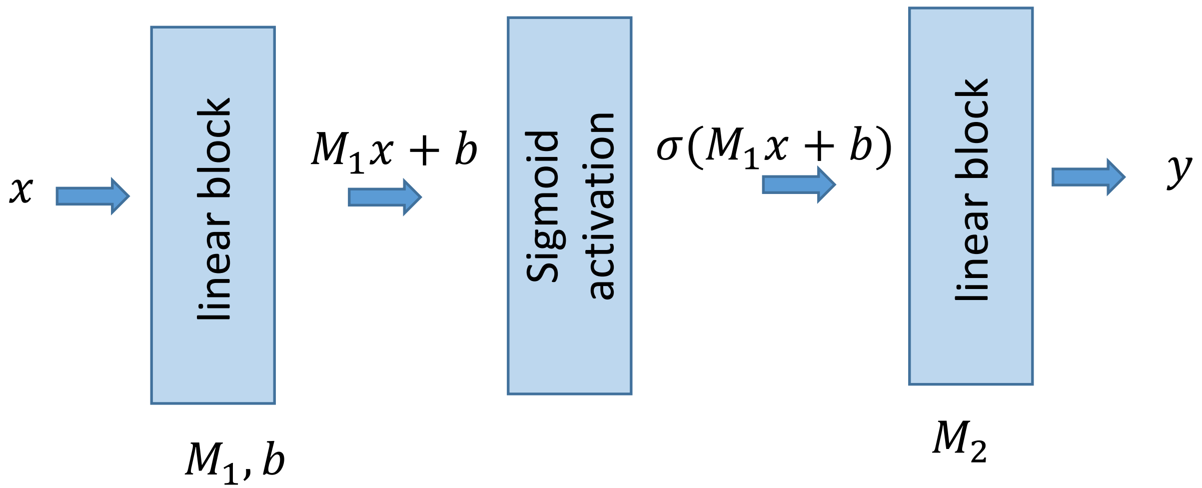
Neural Network Basics II

Today

- Multilayer perceptron
- Activation functions
- Surrogate loss functions
- A closer look at softmax/cross entropy loss
- Short intro to PyTorch

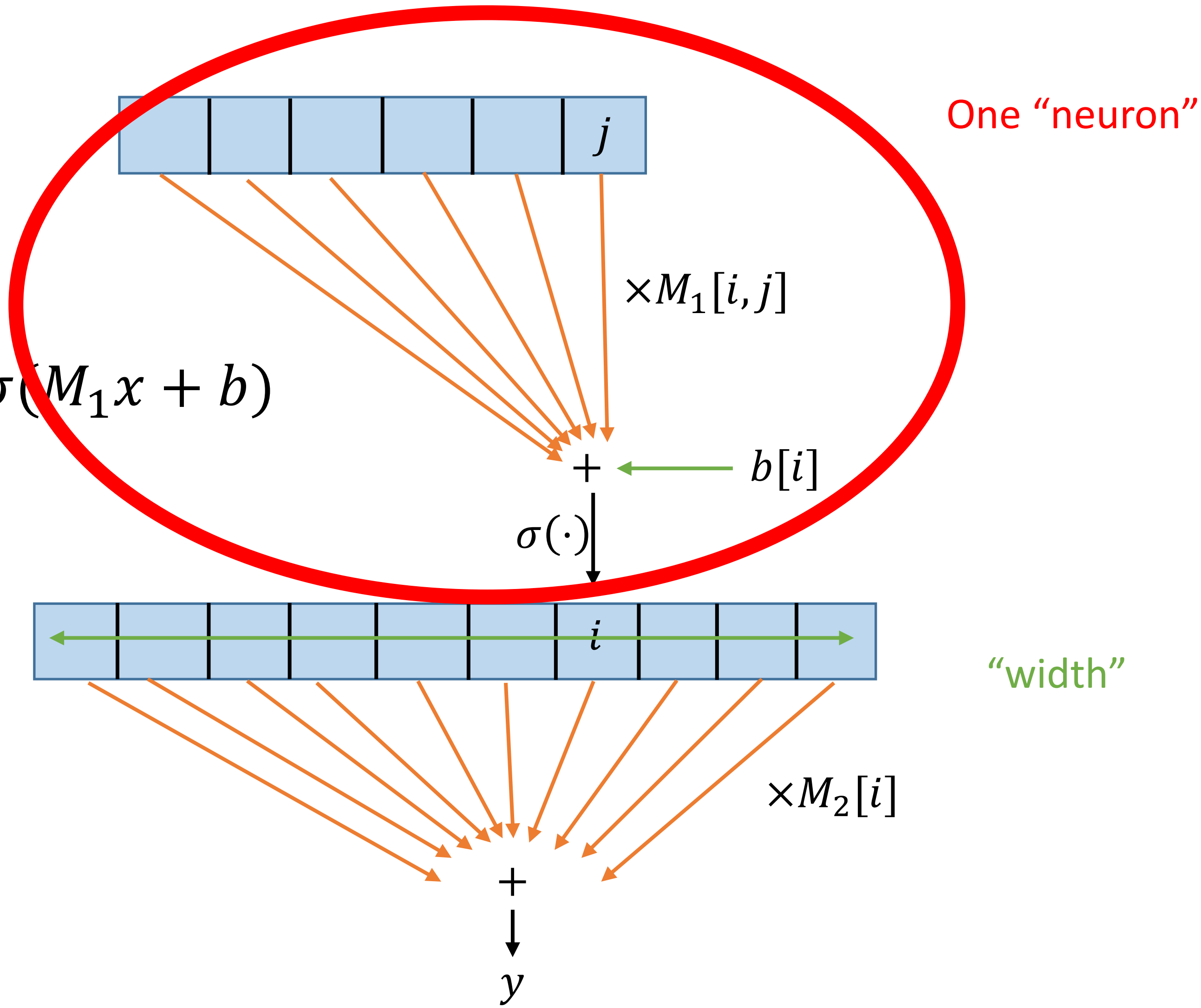
Last time: Block Diagrams



- Activation function $\sigma(x)$ is applied per-coordinate.

Last time: Another Visualization

$$y = M_2\sigma(M_1x + b)$$



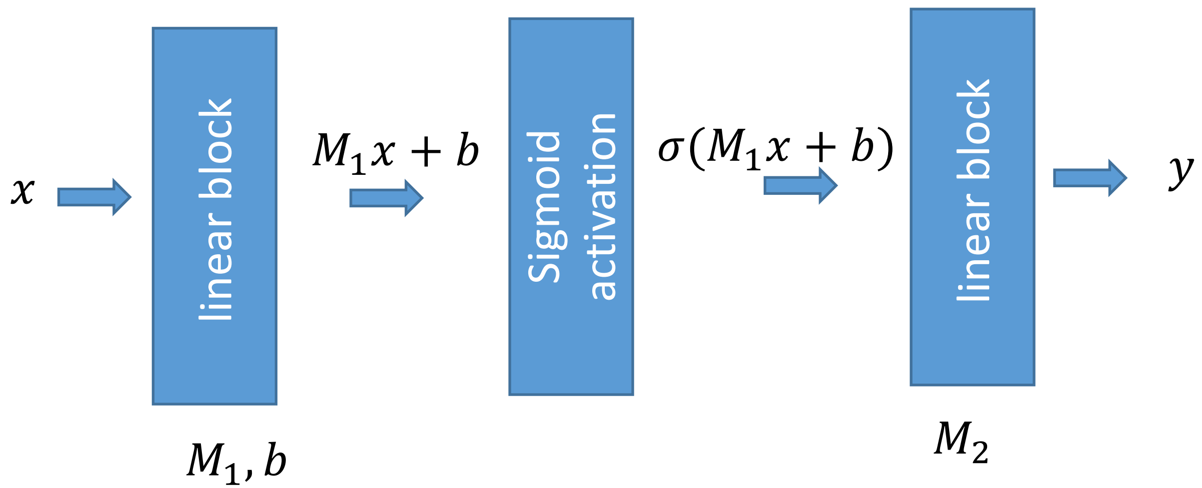
Last Time: Universal Approximation

- Given any function $f: [-1,1]^d \to \mathbb{R}$, any $\epsilon \geq 0$, and any non-polynomial activation function σ, there is some sufficiently large h such that there is a single-layer neural network $W_2\sigma(W_1x + b)$ with activation σ, hidden size h such that for all $x \in [-1,1]^d$:
$$|f(x) - W_2\sigma(W_1x + b)| \leq \epsilon$$

- In practice, h might have to be absurdly large in order for this theorem to be useful.

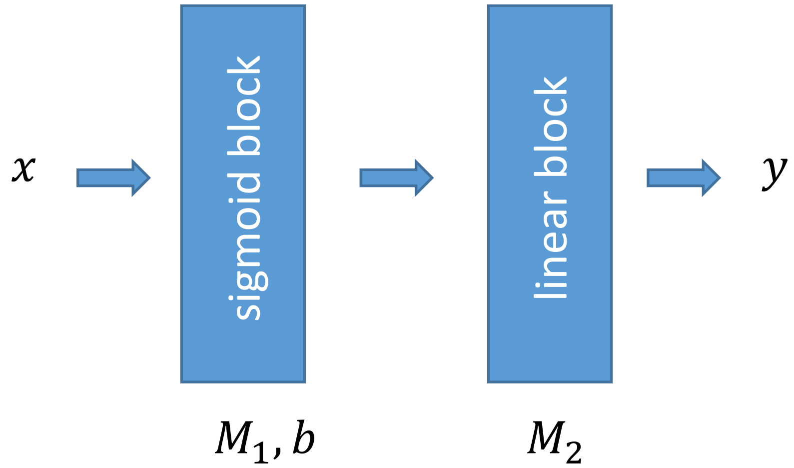
Multi-Layer Perceptron (MLP)

- Last time we talked about a *single layer perceptron*:



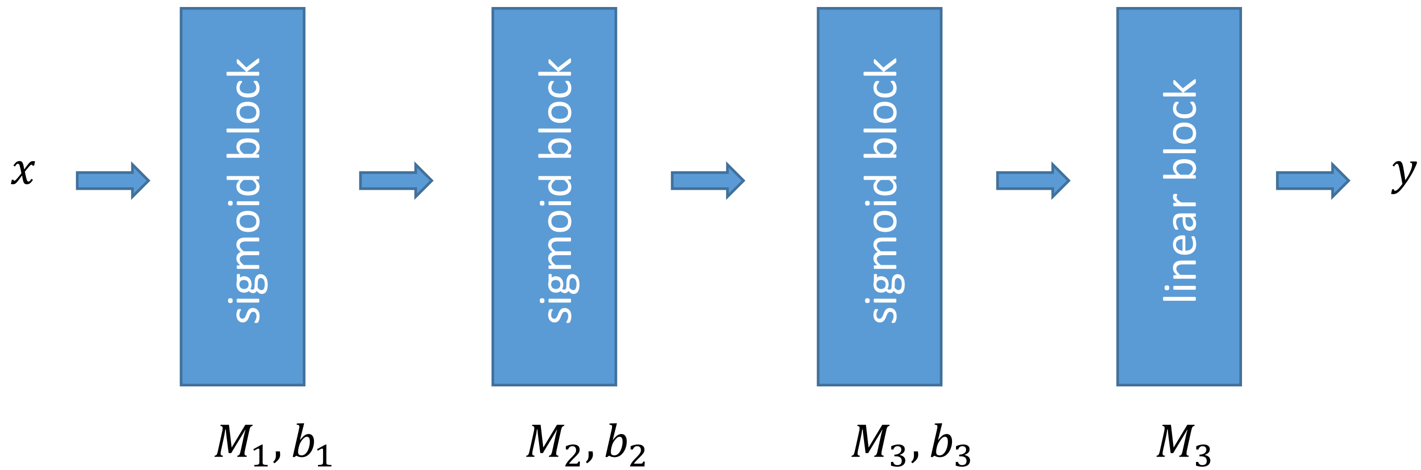
Multi-Layer Perceptron (MLP)

- A more concise diagram of a *single layer perceptron*:



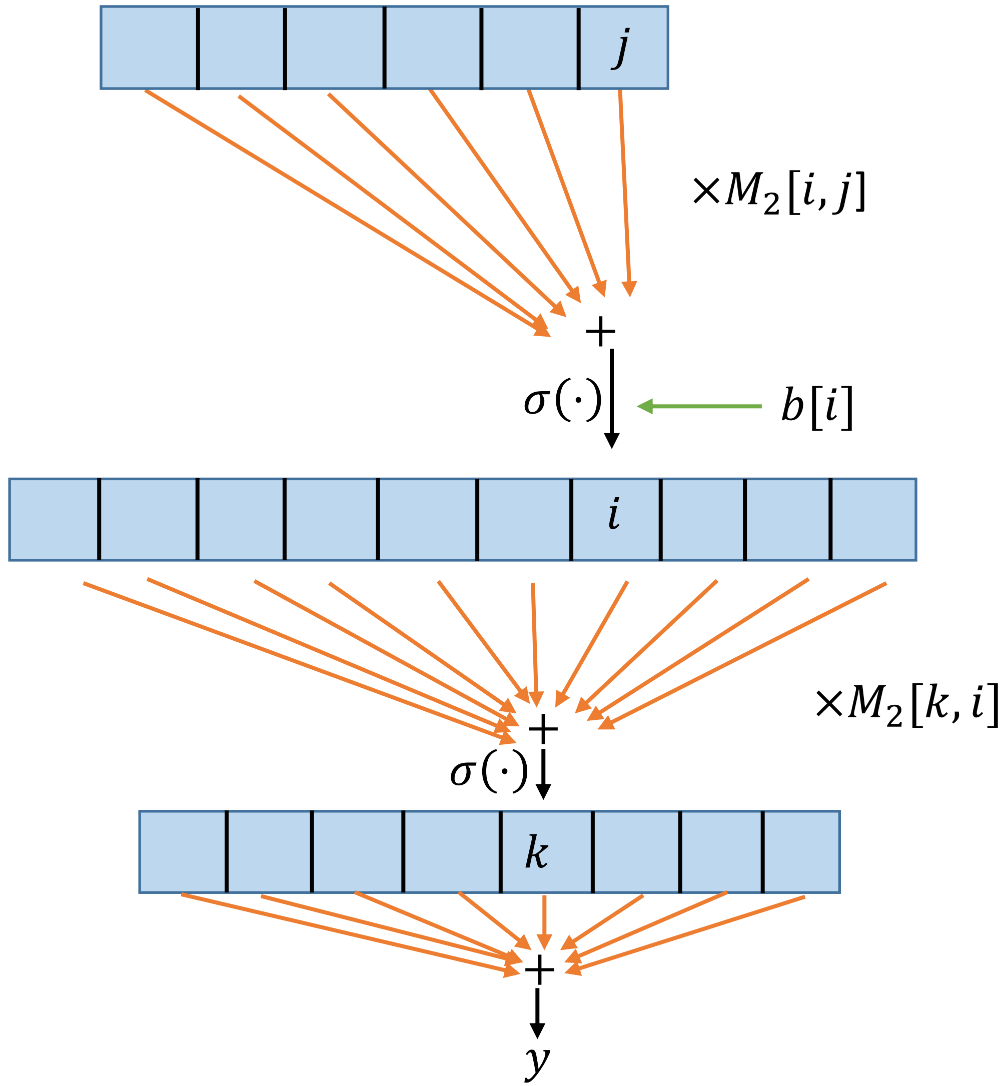
Multi-Layer Perceptron (MLP)

- The MLP just does this many times in a row:



- This is called *depth*. “Deep learning” means using deep architectures.

Multi-Layer Perceptron



Activation Functions

Activation Functions

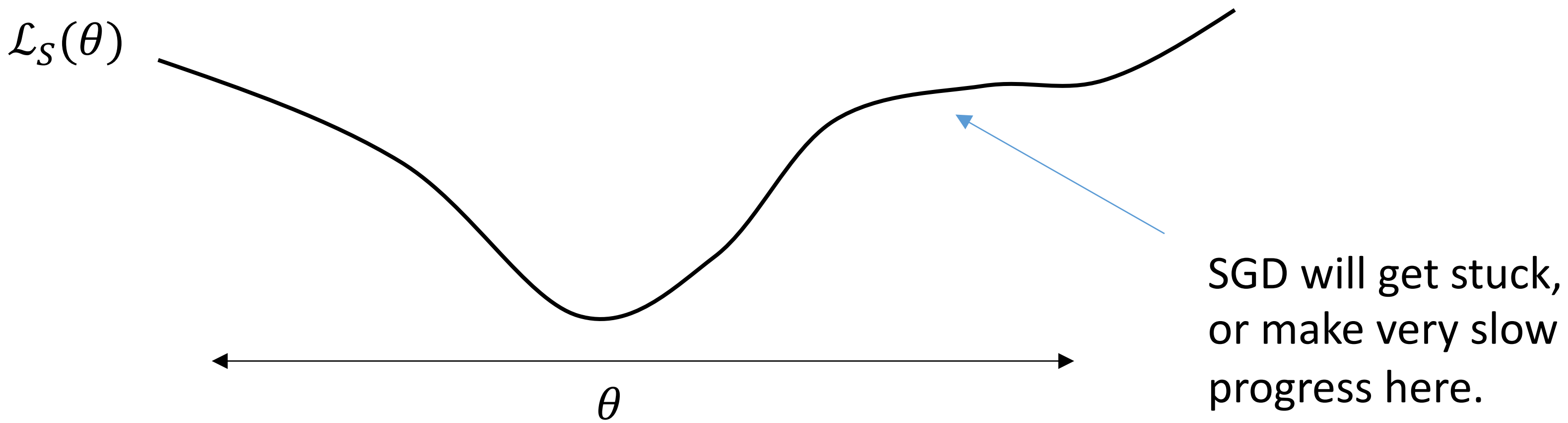
- Last class we covered the *sigmoid activation*:

$$\sigma(x) = \frac{\exp(x)}{1 + \exp(x)}$$

- This activation is no longer so popular because of *vanishing gradients*.

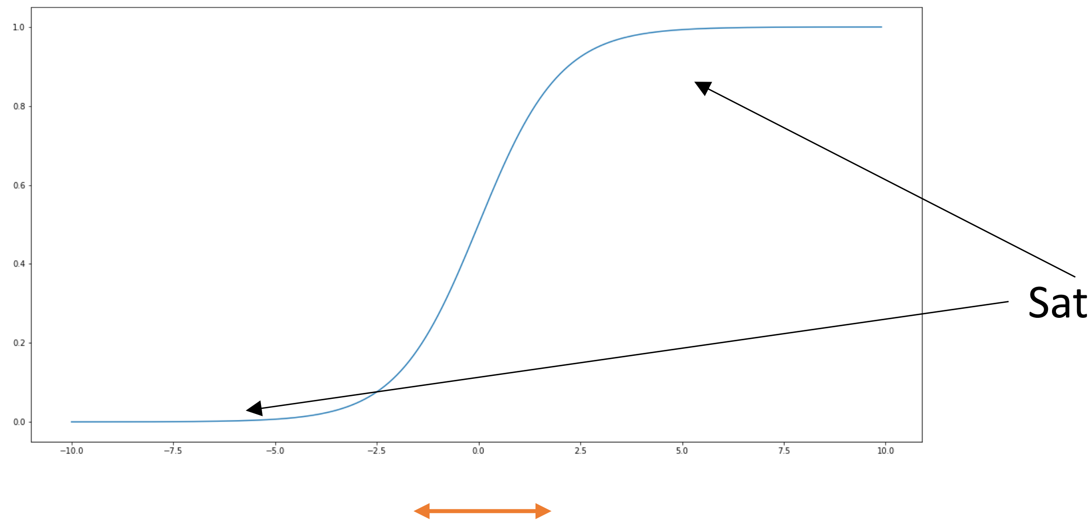
Vanishing Gradients

- Gradient Descent is a “local search” algorithm.



- Small gradient = changing θ by a little does not change the output much.

Vanishing Gradients and Sigmoid



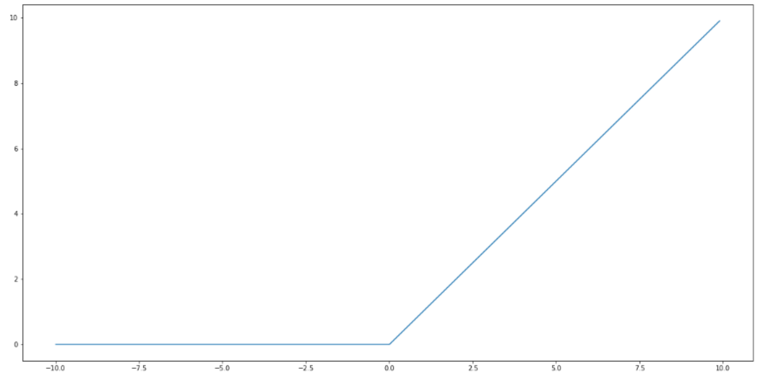
Saturation regions

Range of “learning”

- If the network parameters put the sigmoid into the saturated region, then the gradient will be very small.
- The deeper the network, the more likely one is to hit the saturated region.
- This makes it harder to train deep networks with sigmoid activation.

ReLU activation

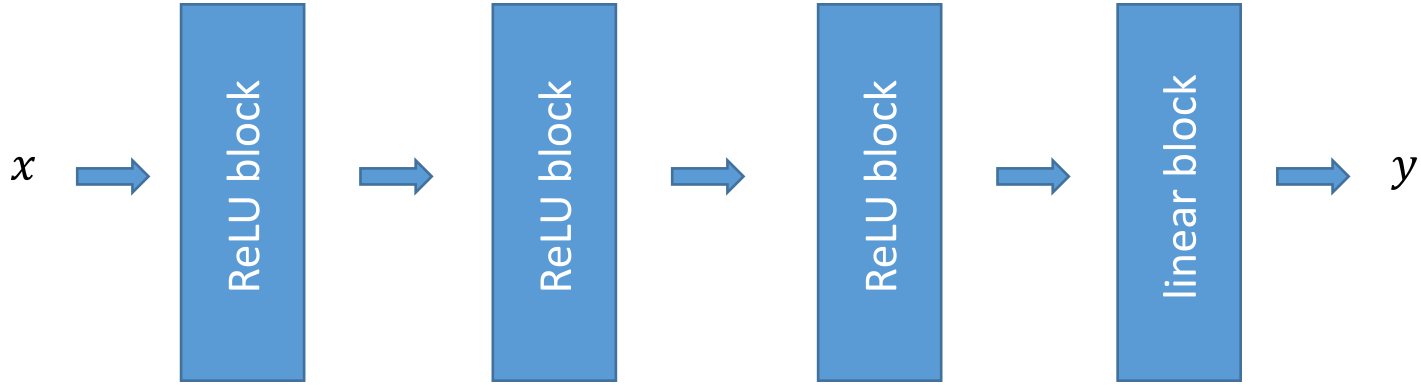
- Modern deep networks usually use some variant of the *rectified linear unit* activation:
$$\sigma(x) = \max(x, 0)$$



- This still shares some inspiration with sigmoid: it is "off" when the input is negative.
- Since it does not saturate on the positive end, it has a much larger range of "learning".

ReLU activation MLP

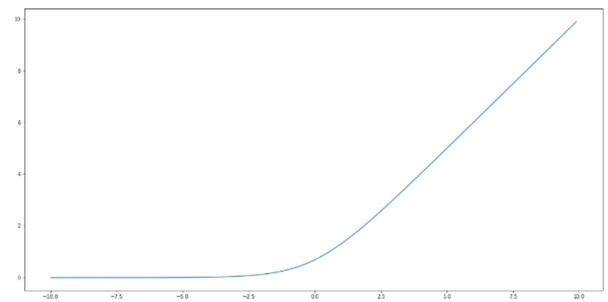
- A more modern MLP looks like:



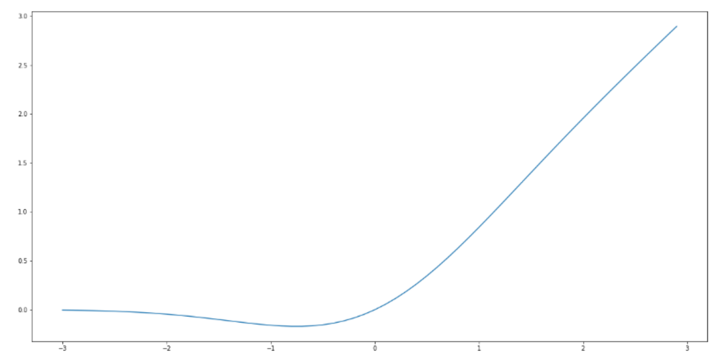
- ReLU networks are piece-wise linear.

Variations on ReLU

- Softplus: $\sigma(x) = \log(1 + \exp(x))$
  - "smooth ReLU"



- GeLU: $\sigma(x) = \dfrac{x\left(1+\mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)}{2}$
  - Much more recent.



Loss Functions

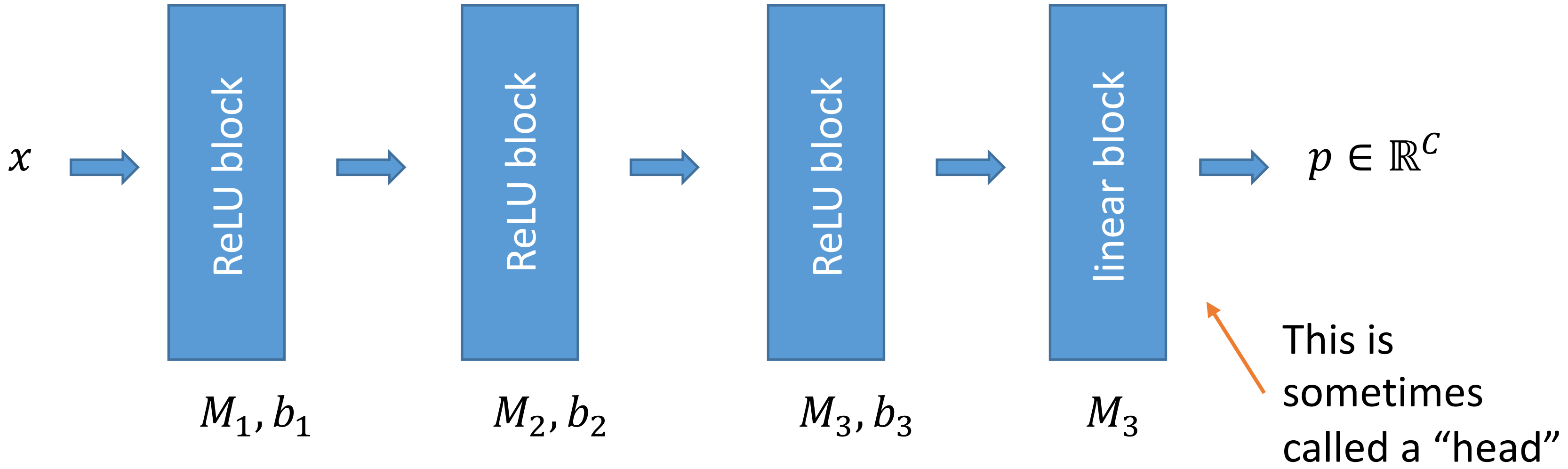
Classification Losses

Cat?

Dog?

Horse?

Classification Architectures

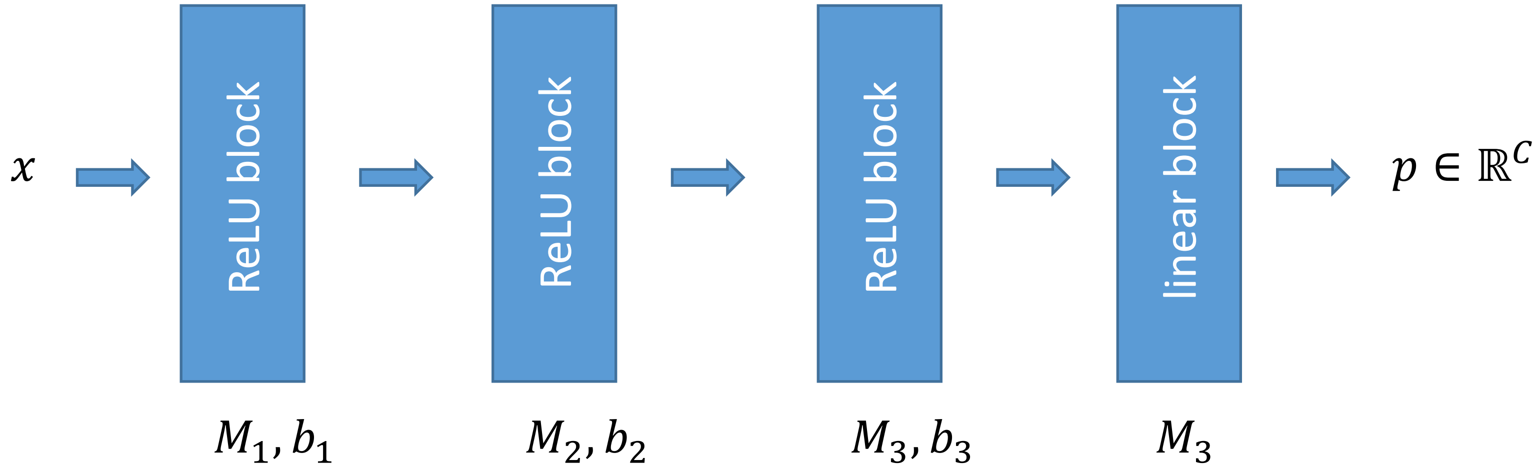


- Prediction is the coordinate of p with the highest value: $prediction = \mathrm{argmax}_j \, p[j]$

True Classification Loss

- We want to optimize the true train error:
$$\sum_{dataset} 1[prediction \neq label]$$
- Let $S = \{(x_1, y_1), \dots, (x_N, y_N)\}$ where $y_i \in \mathbb{R}^C$ is a 1-hot vector:
$$y_i = (0, \dots, 1, \dots, 0)$$
$y_i[k] = 1$ only if x_i has label k.

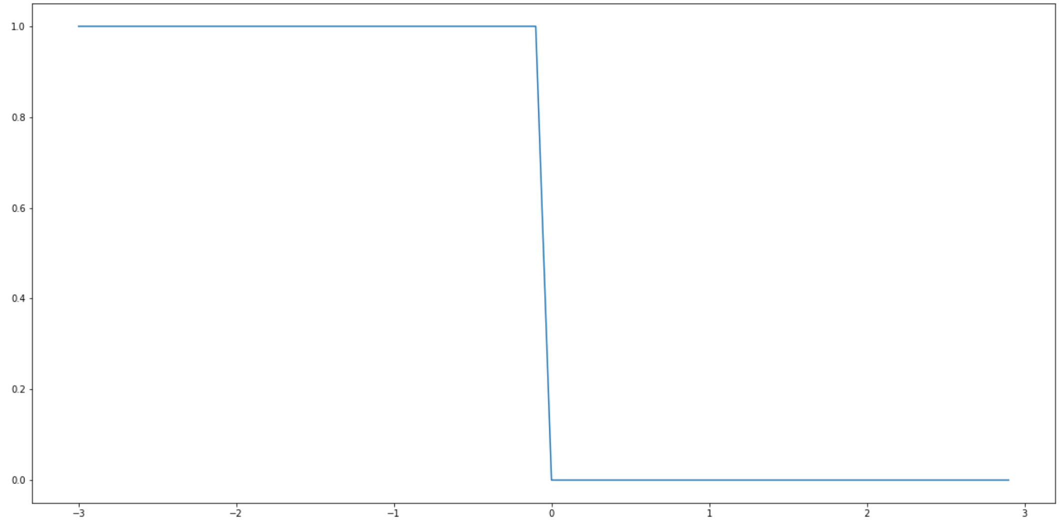
True Classification Loss



- Prediction is the coordinate of p with the highest value: $prediction = \mathrm{argmax}_j p[j]$
- Loss is $\ell(y, p) = 1 - y[prediction]$
- Why can't we use this loss?
  - Non-differentiable
  - Even if it were, there would be vanishing gradients.

True Classification Loss

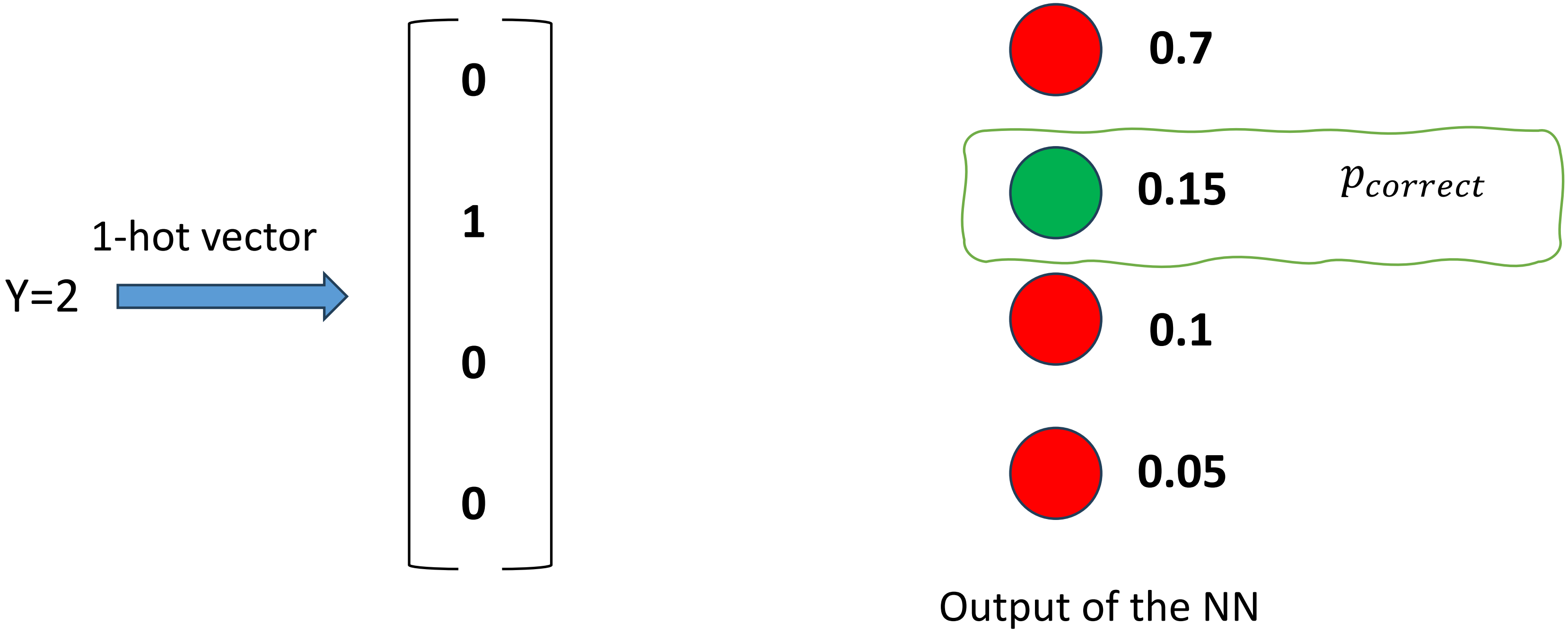
- It's best to visualize this loss for binary classification:



$$p_{correct} - p_{incorrect}$$

Hinge Loss

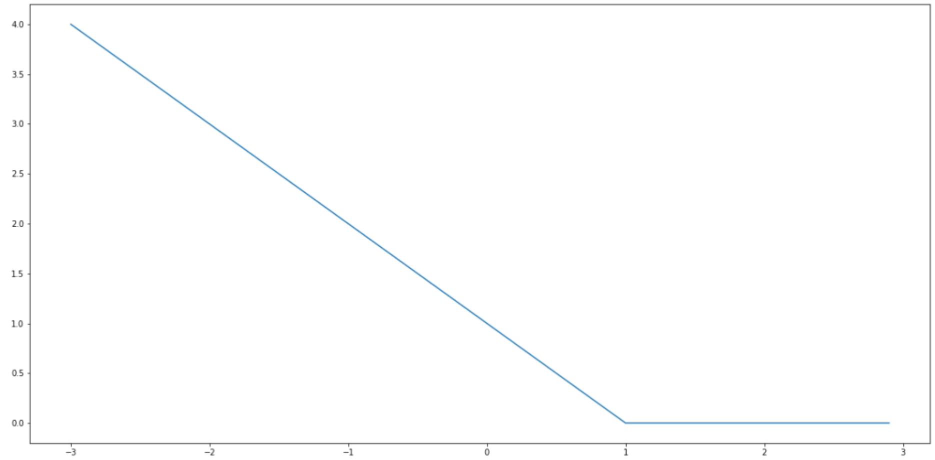
- Hinge is to true loss as ReLU is to sigmoid:
- $hinge(y,p) = \max[\max_{i \neq correct} p_i - p_{correct} + 1\,, 0]$



$$hinge(y,p) = 0.7\ - 0.15 + 1$$

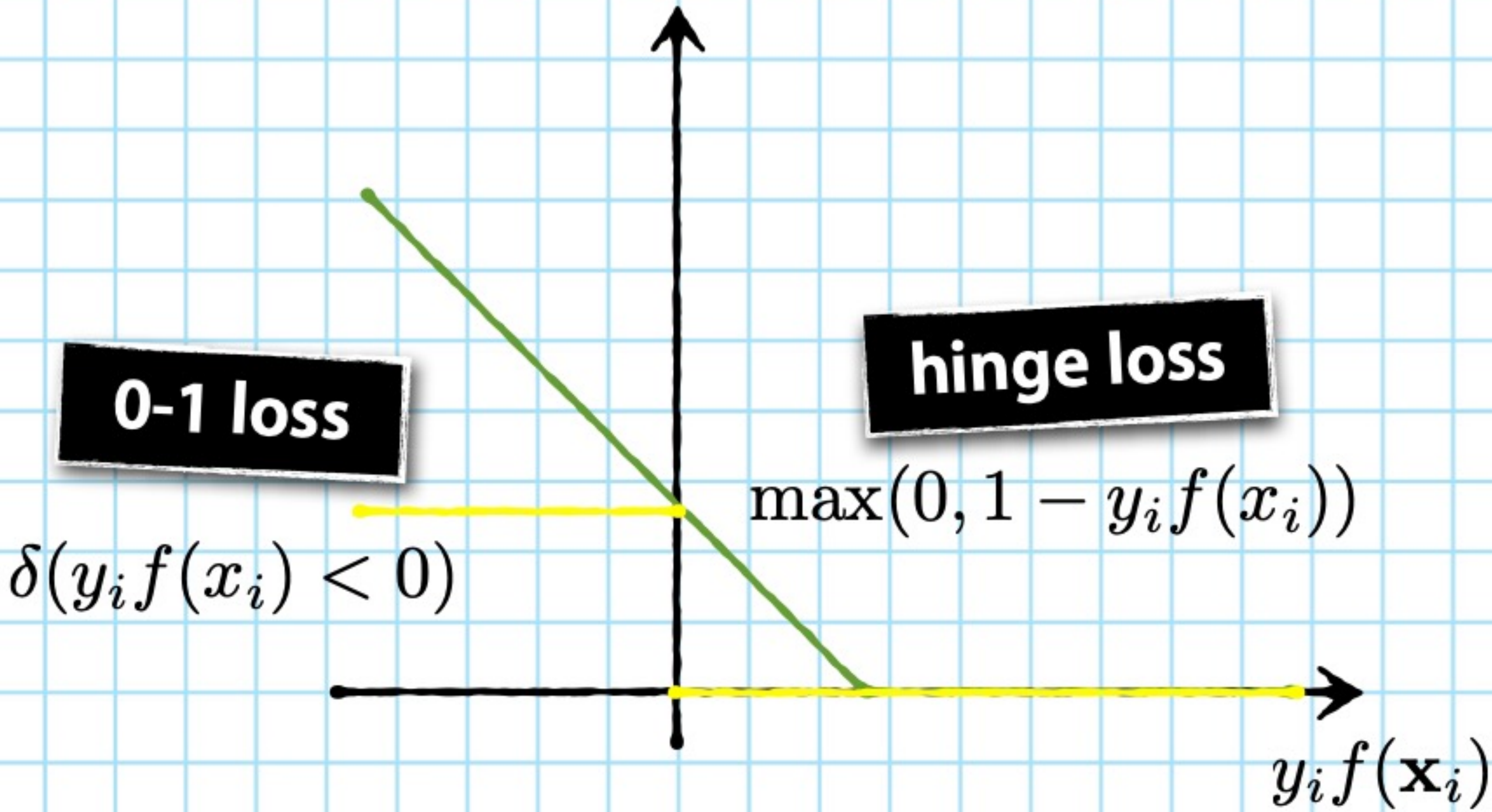
Hinge Loss

- Hinge is to true loss as ReLU is to sigmoid:
- $hinge(y, p) = \max[\max_{i \neq correct} p_i - p_{correct} + 1, 0]$
- The score for the "true" label needs to be 1 more than all the other scores to have zero loss.



$p_{correct} - p_{incorrect}$

Hinge Loss vs True Loss



Hinge Loss vs True Loss

- If the average hinge loss is 0.1, what can you say about the average number of mistakes?
- If the average hinge loss is 0.99, what can you say about the average number of mistakes?

Cross-Entropy Loss

- Input 1: a *distribution* over the C possible output classes

$$p = (p_1, \ldots, p_C), \qquad \sum_{i=1}^{C} p_i = 1$$

- Input 2: the true class value as a 1-hot vector:

$$y = (0, \ldots, 1, \ldots, 0)$$

- The cross-entropy loss is:

$$\ell(p, y) = \sum_i -1[y = i] \log(p_i)$$
$$= -\log(p_{correct\ class})$$

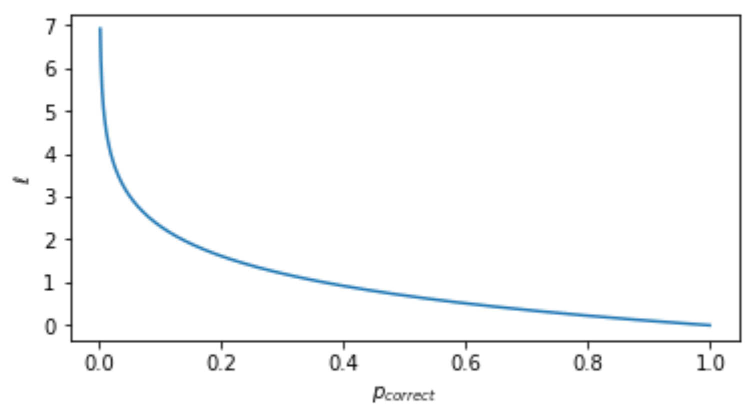
Cross-Entropy Loss

- If the correct class has probability $p_{correct} = 1$,

$$\ell(p, y) = -\log(p_{correct}) = 0$$

- If the correct class has probability $p_{correct} = 0$,

$$\ell(p, y) = -\log(0) = \infty$$



Cross-Entropy Loss

- The hinge loss is the smallest convex upper-bound of the classification loss.
- The cross-entropy is also convex, and positive.
- How is it related to the classification loss?
  - When do we not choose class k?
  - When $k \neq \text{argmax}_i(p_i)$.
  - If $k \neq \text{argmax}_i(p_i)$, then $p_k \leq \frac{1}{2}$.
- If incorrect,

$$\ell(p, y) = -\log(p_k) \geq -\log\left(\frac{1}{2}\right) = \log(2)$$

Cross-Entropy and Classification Loss

- Since $\ell(p, y) \geq 0$, when the prediction is correct we have $\ell(p, y) \geq 1[prediction \neq true\ label]$.
- Otherwise, we have
$$\ell(p, y) = -\log(p_k) \geq -\log\left(\frac{1}{2}\right) = \log(2)$$
- So, in general
$$\ell(p, y) \geq \log(2) 1[prediction \neq true\ label]$$

Softmax

- Cross-Entropy loss requires the input to be a *distribution*.
- The output of a linear layer is never normalized to 1.
- The *softmax layer* turns arbitrary vectors into probability distributions:

$$softmax(x)[i] = \frac{\exp x[i]}{\sum_j \exp x[j]}$$

$$\sum_i softmax(x)[i] = 1$$

UNSTABLE SOFTMAX

$$s_i = \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}$$

```python
import numpy as np

def softmax(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

x = softmax([1, 2, 3])
y = softmax([1000, 2000, 3000])

print('x = ', x)
print('y = ', y)
```

```python
import numpy as np

def softmax(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

x = softmax([1, 2, 3])
y = softmax([1000, 2000, 3000])

print('x = ', x)
print('y = ', y)
```

```python
import numpy as np

def softmax(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

x = softmax([1, 2, 3])
y = softmax([1000, 2000, 3000])

print('x = ', x)
print('y = ', y)
```

```python
import numpy as np

def softmax(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

x = softmax([1, 2, 3])
y = softmax([1000, 2000, 3000])

print('x = ', x)
print('y = ', y)
```

```python
import numpy as np

def softmax(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

x = softmax([1, 2, 3])
y = softmax([1000, 2000, 3000])

print('x = ', x)
print('y = ', y)
```

```python
import numpy as np

def softmax(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

x = softmax([1, 2, 3])
y = softmax([1000, 2000, 3000])

print('x = ', x)
print('y = ', y)
```

Console

```
x = [0.09003057 0.24472847 0.66524096]
```

```python
import numpy as np

def softmax(x):
    exps = np.exp(x)
    return exps / np.sum(exps)

x = softmax([1, 2, 3])
y = softmax([1000, 2000, 3000])

print('x = ', x)
print('y = ', y)
```

Console

```
x = [0.09003057 0.24472847 0.66524096]

y = [nan nan nan]
```

$$
\begin{aligned}
s_i &= \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}} \\
&= \frac{Ce^{a_i}}{\sum_{j=1}^{N} Ce^{a_j}} \\
&= \frac{e^{a_i + \log C}}{\sum_{j=1}^{N} e^{a_j + \log C}}
\end{aligned}
$$

Cross Entropy and KL-Divergence

- Entropy: For a distribution π:

$$H(\pi) = -\sum_i \pi_i \log(\pi_i)$$

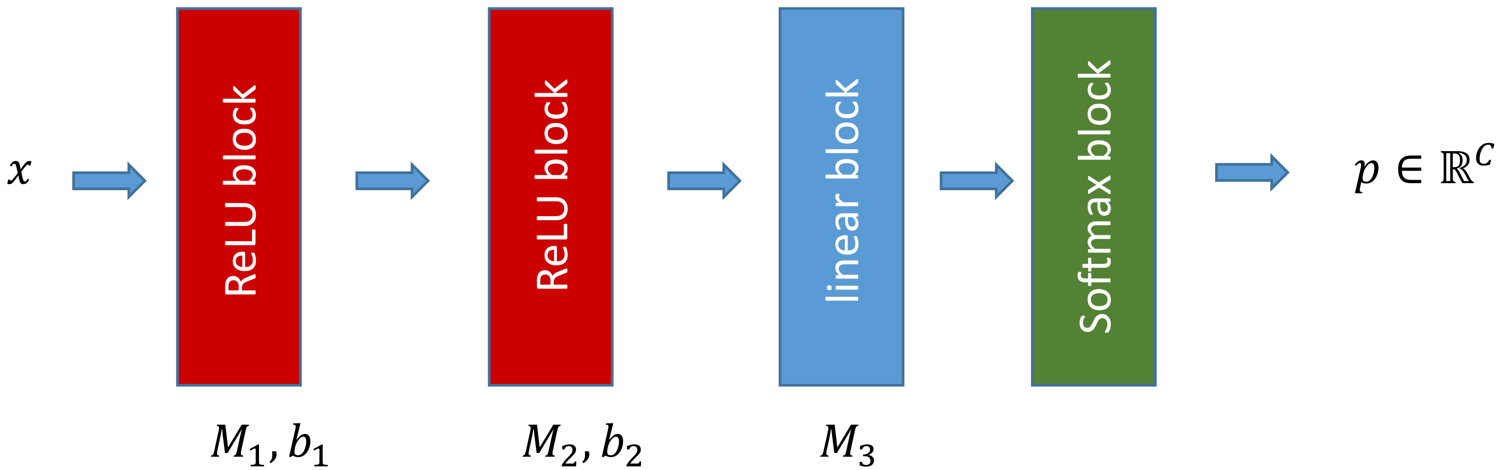
- KL-Divergence:

$$D(\pi||p) = \sum_i \pi_i \log\left(\frac{\pi_i}{p_i}\right)$$

- $D(\pi||p) \geq 0$, and $D(\pi||p) = 0$ only if $\pi = p$.
- Think of $y = \pi$ as a distribution. Then

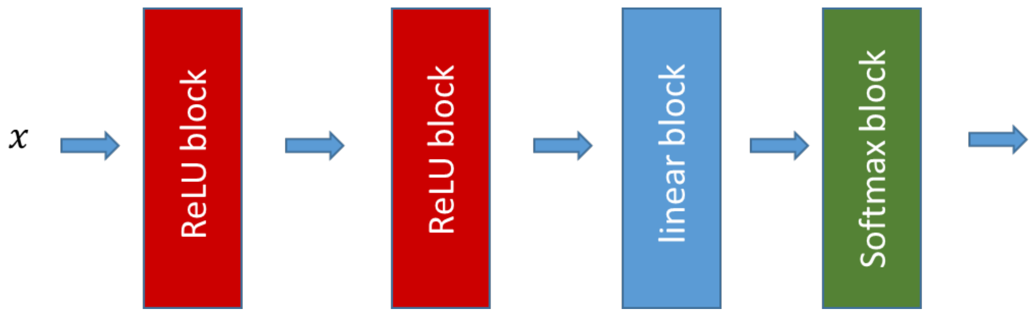
$$crossentropy(p, y) = H(y) + D(y||p)$$

Summary



- Most common architecture: ReLU blocks, followed by a linear block with softmax (for classification)
- What about regression?
- Other tasks?

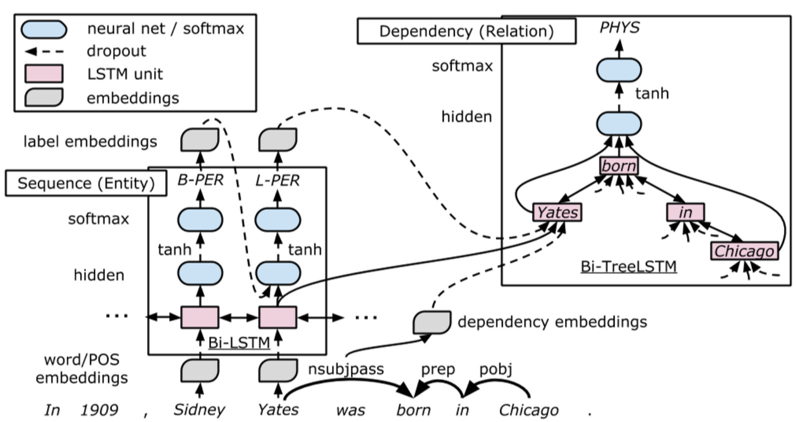
Deep Learning is modular



PyTorch

Why do we need deep learning libraries?

- Deep learning architectures can get very complicated!
- Instead of writing specialized code, use “auto-grad” library
- Easy to specify computation graph, find gradients



Popular DL libraries

PYTORCH

TensorFlow

Caffe2

dmlc mxnet

Microsoft CNTK

Chainer

K

theano

torch

Caffe

Some Pytorch

```python
class TwoLayerNet(torch.nn.Module):
    def __init__(self, input_size, layer_size, output_size):
        super(TwoLayerNet, self).__init__()

        self.Linear1 = torch.nn.Linear(input_size, layer_size)
        self.activation1 = torch.nn.ReLU()

        self.Linear2 = torch.nn.Linear(layer_size, output_size)
        self.activation2 = torch.nn.ReLU()
        self.softmax = torch.nn.Softmax(-1)


    def forward(self, x):
        out = self.Linear1(x)
        out = self.activation1(out)

        out = self.Linear2(out)
        out = self.activation2(out)
        out = self.softmax(out)

        return out
```

Some Pytorch

```
x = torch.tensor([[1.0, 1.0],
                  [1.0, 0.0],
                  [-1.0, 2.0],
                  [2.0, 4.0]])

y = torch.tensor([[0,0,1],
                  [1,0,0],
                  [1,0,0],
                  [0,1,0]])
```

"batch size" = 4

- A "tensor" is an n-dimensional matrix.
- By convention, the first dimension of a tensor of datapoints is the "batch dimension"
- In this example, x is a batch of 4 datapoints, each of which is 2-dimensional.

Some Pytorch

```
x = torch.tensor([[1.0, 1.0],
                  [1.0, 0.0],
                  [-1.0, 2.0],
                  [2.0, 4.0]])

y = torch.tensor([[0,0,1],
                  [1,0,0],
                  [1,0,0],
                  [0,1,0]])
```

```
net = TwoLayerNet(2, 10, 3)
scores = net(x)
loss_func = torch.nn.CrossEntropyLoss()

label_indices = np.argmax(y,axis=1)

loss = loss_func(scores, label_indices)

print("scores: ", scores, "\n\nlabels: ", label_indices, "\n\nloss: ", loss)
```

```
scores:  tensor([[0.3275, 0.4101, 0.2624],
        [0.3724, 0.3478, 0.2798],
        [0.2729, 0.4804, 0.2467],
        [0.2259, 0.5482, 0.2259]], grad_fn=<SoftmaxBackward>)

labels:  tensor([2, 0, 0, 1])

loss:  tensor(1.0730, grad_fn=<NllLossBackward>)
```

Some Pytorch

```
class MultiLayerNet(torch.nn.Module):
    def __init__(self, input_size, layer_sizes, output_size):
        super(MultiLayerNet, self).__init__()

        layer_sizes = [input_size] + layer_sizes
        self.linear_layers = [torch.nn.Linear(layer_sizes[i], layer_sizes[i+1]) for i in range(len(layer_sizes)-1)]
        self.activations = [torch.nn.ReLU() for _ in self.linear_layers]

        self.final_linear = torch.nn.Linear(layer_sizes[-1], output_size)

        self.softmax = torch.nn.Softmax(-1)


    def forward(self, x):
        out = x
        for linear_layer, activation in zip(self.linear_layers, self.activations):
            out = linear_layer(out)
            out = activation(out)
        out = self.final_linear(out)

        out = self.softmax(out)

        return out
```

Next Time

- Backpropogation algorithm
  - How to build an automatic differentiation package.
  - What are these weird “graph” and “leaf” things my tensorflow/pytorch code keeps complaining about?