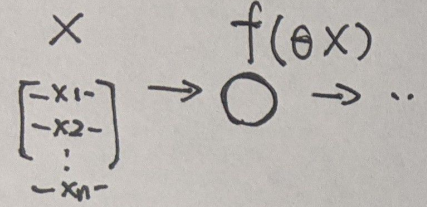
Deep Learning

- Neuron:



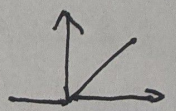
X : data matrix

θ : $\begin{bmatrix} 1 \\ \theta_1 \\ \theta_2 \\ \vdots \end{bmatrix}$ params

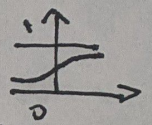
θX : linear comb of param * data

f : activation function:

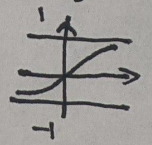
Relu:



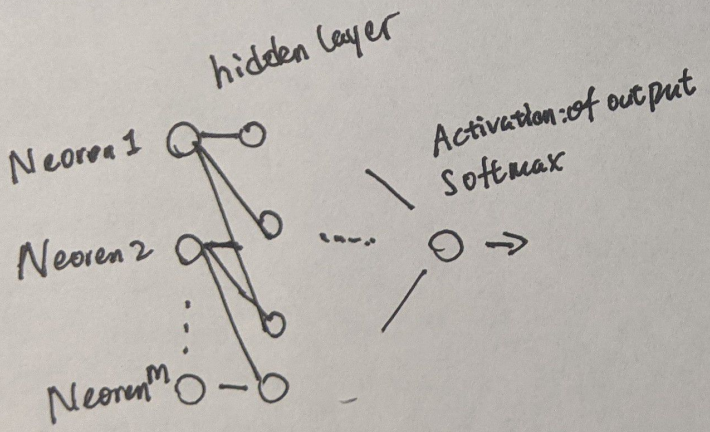
Sigmoid :



tanh:



-



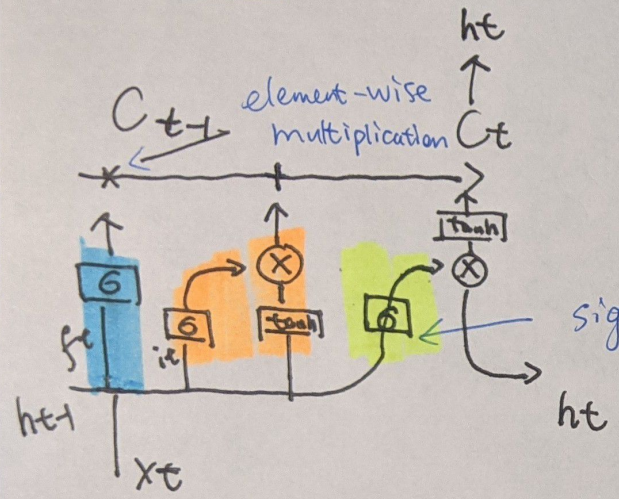
Softmax:

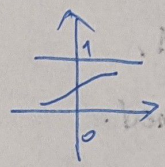
output a array of prob,

Sum of these is 1

- Backpropergation: ~~update~~ update params by gradient descent & chain rule.

LSTM





Computation: Foget Gate Input Gate

$$\tanh\left[\sigma\left(W_f\begin{bmatrix}h_{t-1}\\x_t\end{bmatrix}\right)\sigma\left(W_i\begin{bmatrix}h_{t-1}\\x_t\end{bmatrix}\right)*\tanh\left(W_c\begin{bmatrix}h_{t-1}\\x_t\end{bmatrix}\right)\right] + \sigma\left[W_o\begin{bmatrix}h_{t-1}\\x_t\end{bmatrix}\right]$$

new Value

Output gate

params:

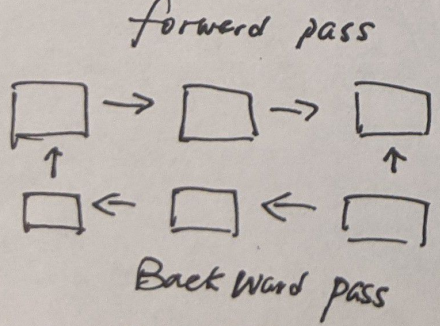
$4 \times \text{Shape}(h) \times (\text{Shape}(h) + \text{Shape}(x))$

W_f, W_i, W_c, W_o

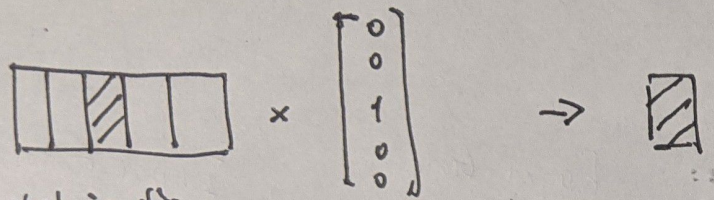
$h\begin{bmatrix}\quad\\\quad\end{bmatrix}$

$h + x$

Bidirectional Lstm:



Word Embedding

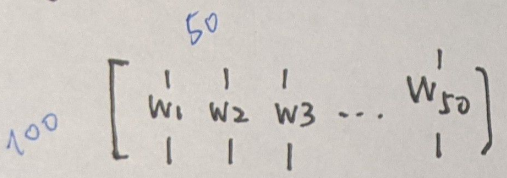


Veb in dic　　one hot encoding　　encoding for one word,
length is customed.

Model From Internship,

text classification for cleaning text.

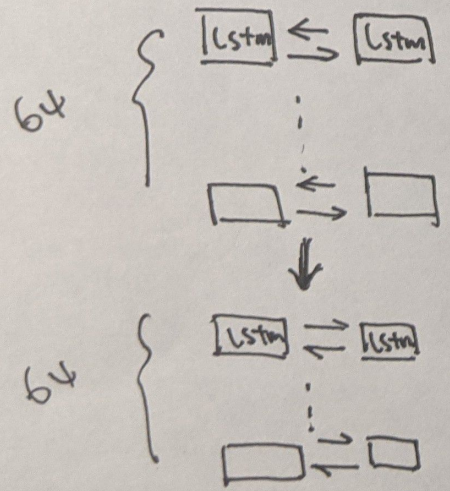
- each piece of doc are sep by " " (triple space)
- 0 as need to keep
  1 as need to remove
-



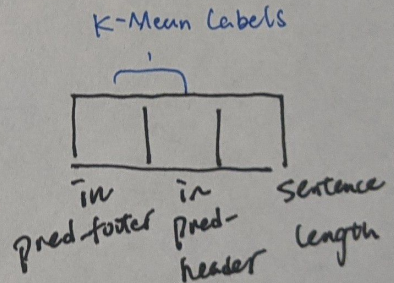
---- Embeding layer

100 for embedding vector length

50 for max length of each doc.



64 Bidirection
---- Lstm units

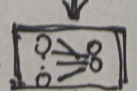


... # params:

$2 \times 4 \times [(64+100) \times 64 + 64] \approx 80k$ (bias)

use layer regularizer to prevent overfitting.

Concatenate



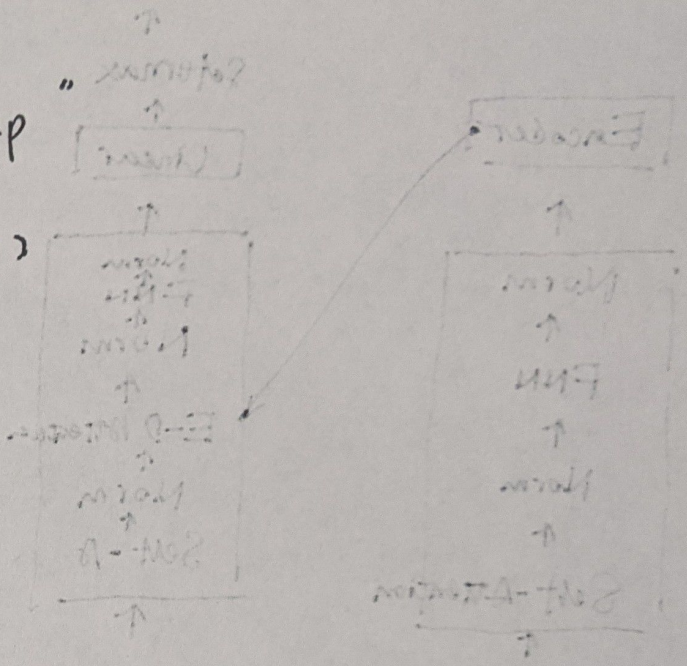
DNN, Relu

Sigmoid

Cross Entropy Loss

Transformer Networks

- Vector - Seq Model : Dog img → "Dog in a cup"
- Seq - vec Model : "Text..." → [0~1]
- Seq - seq Model : Translation



RNN ↑ x^t x^{t+1}

Transformer ↑ x_1 x_2 x_3 ... — Positional Encoder:

vector that gives context based on position of word in sentence

Seq. model

- Attention Layer. What part of the input should we focus?

E-C Attention Structure:

Enc: $[x_1 \; x_2 \; x_3 \; \cdots \; x_m]$

- $k_i = W_k x_i$ ...
- $v_i = W_v x_i$ ...

Dec: $[x_1' \; x_2' \; \cdots \; x_m']$

- $q_j = W_q x_j'$ ...

Formulas $C = \text{Attn}(X, X')$

$\alpha_1 = \text{Softmax}(K^T q_{:1}) \in \mathbb{R}^m$

$C_1 = \alpha_{11} v_1 + \cdots \alpha_{m1} v_m = V \alpha_1$

$\alpha_2 = \cdots$

$C_2 = \cdots$

$\vdots$

C_m

Notes:

C can be used to generate next word (Translation Task)

Self-Attention Layer

- Query $q_i = W_Q x_i$

  Key $k_i = W_K x_i$

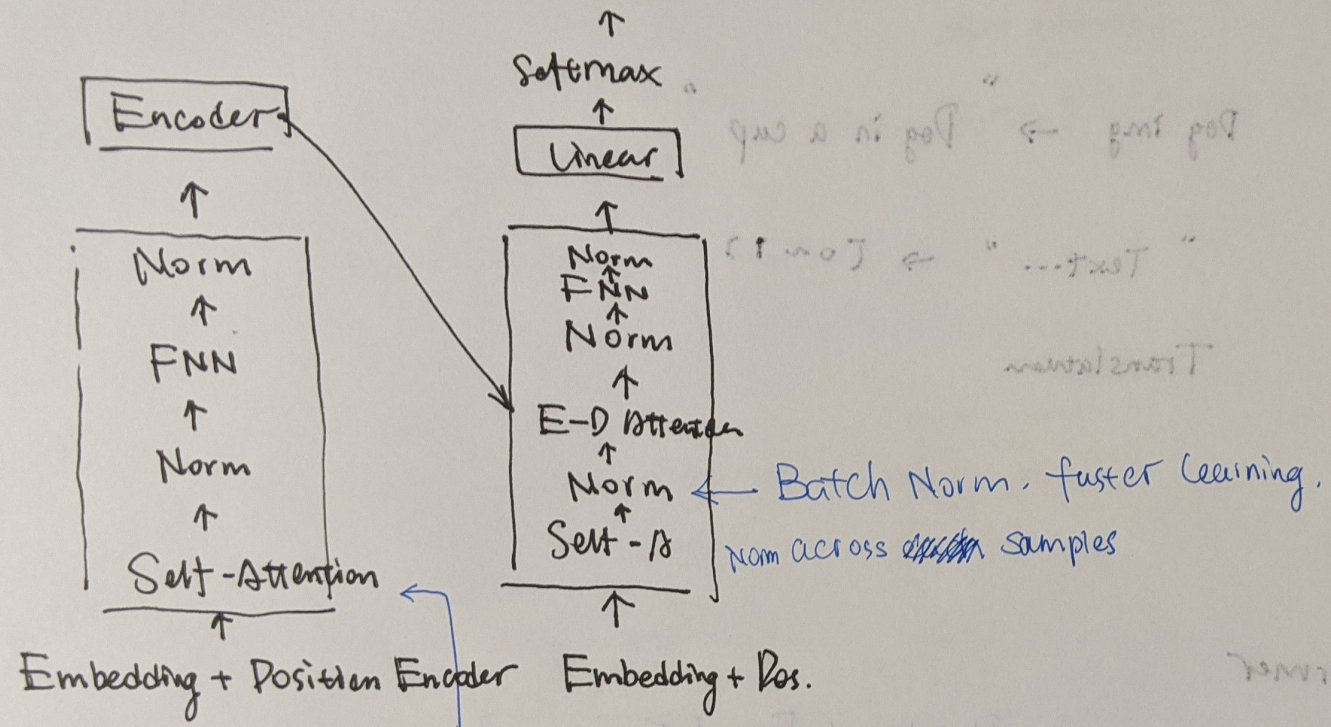
  Value $v_i = W_V x_i$

- $\alpha_j = \text{Softmax}(K^T q_{:j})$

  $c_1 = \alpha_{11} V_{:1} + \cdots \alpha_{m1} v_m = V\alpha_{:2}$ $\Big\} \Rightarrow c_{:j} = V \cdot \text{Softmax}(K^T q_{:j})$

- Self-attention layer: $C = \text{Attn}(X, X)$

- Encoder - Decoder



BERT

- is for pre-training Transformer's encoders
- Predict ① masked word ② next sentence.

① mask 15% words randomly.

output a prob @ masked position.

Calculate loss = CrossEntropy(e, p)

ground-truth ↑ e, ↑ Prob

Perform GD to update parameters

② use 50% true linked 2 sentences, 50% faked

[CLS] Sent 1 [SEP] Sent 2

↑ token for classification ↑ for seperating sentences

- Combing Two Tasks

input

example: "[CLS] Calculus is a [MASK] of math.

[SEP] it [MASK] developed by newton and leibniz."

Target: true. "branch", "was"

loss:

next-sent: binary clf: cross Entropy

masked word: multi-class clf

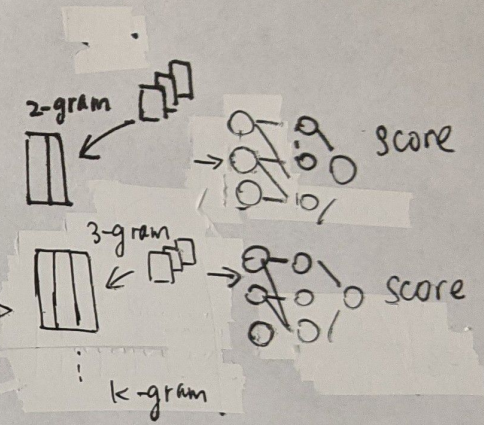
Sum of three loss

Model From Internship. , keyphrase extraction task

- Unsupervised Keyphrase Extraction:

n-grams → Pretrained Bert → Word Embedding ↑
doc → .. → Word Embedding ↓
Cosine similarity to select keywords.

- Supervised Keyphrase Extraction. (BERT-KPE)



- n-grams → Pretrained Bert (Base/SpanBert/RoBert) →

- Loss:

Hingle Loss: pairwise ranking loss $\sum \max(0, 1 - f^*(p_+, D) + f^*(p_-, D))$ phase, doc

CE Loss: is or not true keyphrase $\mathcal{L}_{chunk} = \text{CrossEntropy}(p(c_i^k = y_i^k))$

Multi-task training $\mathcal{L}_{oss} = \mathcal{L}_{Rank} + \mathcal{L}_{chunk}$.

Pytorch:

Pros : many things are modulizable

Cons : may not good for launching model into production

Code example for train model (Stats 507)

```
for epoch in range(n_epochs):
    model.train()
    for bdx, batch in enumerate(train_loader):
        # forward passing
        Logits, probs = model(...)
        #          ↑ input   ↑ output # softmax
        Cost = F.cross_entropy(Logit, true_counts)
        optimizer.zero_grad()
        Cost.backward()
        # update model parameters
        optimizer.step()
    # evaluation
    model.eval()
    Cost = Compute_epoch_loss(model, train_loader)
    :
```

More codes from lecture —>

PyTorch Usage: Step 1 (Definition)

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_feat, num_h1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_h1, num_h2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_h2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Define model parameters that will be instantiated when created an object of this class

Define how and it what order the model parameters should be used in the forward pass

PyTorch Usage: Step 2 (Creation)

```
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                             num_classes=num_classes)

model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
```

Instantiate model
(creates the model parameters)

Optionally move model to GPU, where
device e.g. torch.device('cuda:0')

Define an optimization method

PyTorch Usage: Step 3 (Training)

Run for a specified number of epochs

Iterate over minibatches in epoch

If your model is on the GPU, data should also be on the GPU

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(probas, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

model.eval()
with torch.no_grad():
    # compute accuracy
```

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

model.eval()
with torch.no_grad():
    # compute accuracy
```

This will run the `forward()` method

Define a loss function to optimize

Set the gradient to zero
(could be non-zero from a previous forward pass)

Compute the gradients, the backward is automatically constructed by "autograd" based on the forward() method and the loss function

Use the gradients to update the weights according to the optimization method (defined on the previous slide)
E.g., for SGD, $w := w + \text{learning\_rate} \times \text{gradient}$

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

model.eval()
with torch.no_grad():
    # compute accuracy
```

For evaluation, set the model to eval mode (will be relevant later when we use DropOut or BatchNorm)

This prevents the computation graph for backpropagation from automatically being build in the background to save memory

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

model.eval()
with torch.no_grad():
    # compute accuracy
```

logits because of computational efficiency.
Basically, it internally uses a log_softmax(logits) function that is more stable than log(softmax(logits)).
More on logits coming up.